

MODELING GRANULARITY IN DATA FLOW PROGRAMS

Daniel R. Greening

**February 1988
CSD-880009**

UNIVERSITY OF CALIFORNIA

Los Angeles

Modeling Granularity in Data Flow Programs

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in Computer Science

by

Daniel Rex Greening

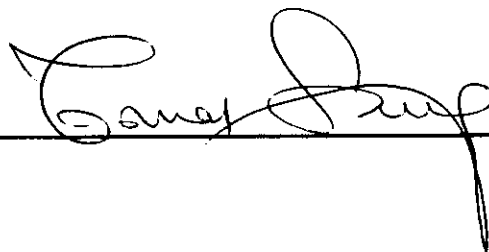
1988

© Copyright by

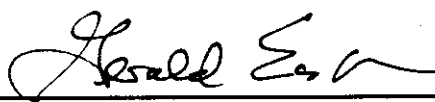
Daniel Rex Greening

1988

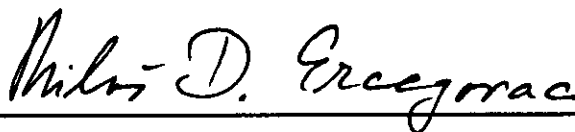
The thesis of Daniel Rex Greening is approved.

A handwritten signature in cursive script, appearing to read "Tomas Lang", written above a horizontal line.

Tomas Lang

A handwritten signature in cursive script, appearing to read "Gerald Estrin", written above a horizontal line.

Gerald Estrin

A handwritten signature in cursive script, appearing to read "Miloš Ercegovac", written above a horizontal line.

Miloš Ercegovac, Committee Chair

University of California, Los Angeles

1988

DEDICATION

My grandmother, Violette Mae Teeters, was born to a farming family in Ohio. She taught rural elementary school from 1916 to 1919. She moved to Michigan, married my grandfather in 1920, and began a long career of academic and civic activism.

Because she and my grandfather refused to join, the Ku Klux Klan burned a cross in their front yard. She wrote free lance articles for the Detroit Free Press. She raised three successful children.

In 1949, enlisting the help of other community members and overcoming many obstacles, she built the first public library in Durand, Michigan. In 1969 and 1970, she coordinated a Senior Citizens' group in Clifton, Texas, as a member of VISTA, the Volunteers in Service to America.

In 1972, at the age of 74, she received a Bachelor of Arts degree in History from Central Michigan University. In 1985, at the age of 87, she received a Master of Arts degree in History, again from CMU. She is the oldest known graduate of CMU.

Her interests are wide, her abilities diverse, her will strong, and her love of fellow humans steadfast.



TABLE OF CONTENTS

1	Introduction	1
1.1	Background	2
1.2	Data Flow Notation and Terminology	3
1.3	The Problem	8
1.4	Thesis Contributions	10
2	Previous Research	13
2.1	Research on the Effects of Granularity	13
2.1.1	Gaudiot's High Granularity Data Flow Ring	14
2.1.2	Serial Combinators	15
2.2	Performance Modelling	16
2.2.1	Kapelnikov's System Model	16
2.2.2	Thomasian and Bay Task System Performance Analysis	19
3	Modeling Data Flow Program Performance	21
3.1	Probabilistic Data Flow Graphs	22
3.1.1	Introduction	22
3.1.2	Formal Description	24
3.1.3	PDFG Execution	29
3.1.4	Transition Probabilities	33
3.1.5	Fitting the Model to the System	34
3.1.6	Obtaining Probability Estimates	35
3.2	Interpreting the Markov Model	36
3.2.1	Generating a Markov Chain	38
3.2.2	Closed Subsets in Markov Chains	42
3.2.3	Obtaining Expected Execution Time	47
3.2.4	Limitations of the Stochastic Model	47
4	Partitioning Data Flow Programs	49
4.1	Naturally Sequential Blocks	49
4.2	Execution Times for Different Partitionings	54
4.3	Obtaining a Solution	57
5	Implementation	58
5.1	The Analysis Program	59
5.2	Input Format	60
5.3	Example Programs	63

5.3.1	Program 1: INTEGRATE	64
5.3.2	Program 2: RECURSIVE_AQ	67
5.3.3	Discussion of Example Programs	71
6	Conclusion	73
6.1	Work Remaining	74
A	Notation	76
B	RECURSIVE_AQ: Data Flow Program Listing	77
C	RECURSIVE_AQ: Identified Partitions	80
D	The Simulator	89
D.1	Instructions Provided	90
	References	94

LIST OF FIGURES

1.1	Pictorial Conventions	5
1.2	Merge Actor and Our Representation	6
1.3	Program Fragment, Three Partitions	7
2.1	Manchester Communication Ring	17
3.1	Probabilistic Data Flow Graph	36
3.2	Markov Chain for Probabilistic Data Flow Graph	39
3.3	Deleting Closed Subsets from Markov Chain	42
3.4	Markov Chain for PDFG After Algorithm FMCS	45
4.1	Two Outcomes of Algorithm PSB	54
4.2	Optimal Partition Depends on Input	56
5.1	RECURSIVE_AQ: Probabilistic Data Flow Graph	69

LIST OF TABLES

3.1	Transitions for Example PDFG	40
3.2	Transitions Minus Closed Subset for PDFG	46
5.1	INTEGRATE: Analysis vs. Simulation	66
5.2	RECURSIVE_AQ: Analysis vs. Simulation	70

ACKNOWLEDGEMENTS

I must first acknowledge my advisor, who reminded me of the "big picture" while I was mired in the details of this thesis. He supported me with money and words, at times when I really needed both. He believes in my abilities. Thank you very much, Miloš.

Dr. Sheila Greibach reviewed Chapter 3 and suggested numerous improvements. Steve Skedzielewski of Lawrence Livermore National Laboratory supplied the SISAL compiler. Nancy Greening, my mother, supplied the drafting software used for several figures.

I owe a great debt to several individuals whose encouragement and love has meant much to me, and who helped me finish this work through their moral support:

Dan J. Greening, my father, has been a consistent friend, a hiking buddy, a repossession partner in a hair-raising truck recovery, a telephone auto-mechanic, and an education financier. He stuck by me when the going got tough. Verra Morgan, student affairs officer in the UCLA Computer Science Department, served as a great coach throughout this project. Ron Lussier and Ed Arnes, two good friends, pushed me to keep going and cheered me on.

The Office of Naval Research generously sponsored this work under grant number N00014-85-K-0159.

ABSTRACT OF THE THESIS

Modeling Granularity in Data Flow Programs

by

Daniel Rex Greening

Master of Science in Computer Science

University of California, Los Angeles, 1988

Professor Miloš Ercegovic, Chair

The execution time of a data flow program in a system depends intrinsically on the degree of parallelism available, the resource requirements of the program, and the resources provided by the system. Communication delays between actors in a data flow graph present a significant performance degradation factor. We can reduce these delays by partitioning actors into large sequential blocks. This thesis provides a method for optimally partitioning static cyclic data flow graphs into sequential blocks, when we know transition probabilities and communication delays, to reduce overall execution time.

A structural model of data flow programs, called a “probabilistic data flow graph,” provides a mathematical base for our analysis. We provide a method for converting probabilistic data flow graphs to Markov chains.

We provide an algorithm to give the set of all maximal sequential partitionings for a data flow graph. Selecting an optimal partitioning from this set, when tran-

sition probabilities are not fixed, is incomputable. When transition probabilities are fixed and known, we use Markov analysis to select the optimal partitioning. We discover that suboptimal partitionings provide a nearly optimal speedup.

We show two sample data flow programs, apply our algorithms, and discuss the advantages and disadvantages of the method based on the examples.

CHAPTER 1

Introduction

Communication and synchronization delays often dominate the execution time of dynamic data flow programs. We can reduce execution time by identifying naturally sequential blocks in dynamic data flow programs which require no synchronization. Generally, one can find several ways to partition a data flow program, each with a different execution time.

We present an algorithm for enumerating all maximal sequential block partitions of a dynamic data flow program. We construct the notion of a “probabilistic data flow graph,” which allows us to describe a data flow program’s behavior stochastically. Using a Markov modeling heuristic, we can estimate the execution speed of each partition. A software modeling tool we wrote applies these concepts to select a good partitioning for a data flow program.

To judge the results of our modeling heuristic, we also wrote a tagged-token data flow simulator, based loosely on the principles of the Manchester [Gurd85] and the MIT Tagged-Token [Arvi87] data flow machines. We present two example programs, and the results of applying our modeling system and simulation.

We discover that the modeling system chooses good partitions, but it requires a large amount of CPU time. For large programs whose actual execution time

is small, simulation will provide partitioning information faster. However, when a small program executes for a long time, our modeling system has merit over simulation.

In the introductory sections that follow, we provide some background on data flow programs, present some definitions and formalisms used later in the text, discuss the notion of data flow granularity, and elaborate on our thesis contributions.

1.1 Background

Traditional computers conform to the control flow computing model (sometimes called the von Neumann model), which attaches a single storage-unit to a sequential control flow instruction processor.

Many researchers have criticized the control flow model's inherent difficulties in executing concurrent programs [Back78]. Pipelining, multiprocessing, and layered storage-units have improved the von Neumann model. But the so-called "von Neumann bottleneck," the data path between the instruction processors and memory, caps the maximum parallelism that a von Neumann architecture can exploit.

Researchers have proposed several alternative models to solve the problems inherent in control flow machines [Vegd84] [Burt81] [Chan84] [Denn79] [Denn80] [Erce84] [Grna80] [Gurd78] [Mago80]. All attempt to avoid the von Neumann bottleneck. Many conform to the "data flow" computing model.

Data flow programs use no variables. Instead, one expresses all intermediate results as “tokens” traveling along the directed edges of the graph, which establish precedence relations between operators. When an operation’s predecessors have all produced their results, the operation “consumes” its input values and begins execution. Parallelism occurs implicitly.

Data flow execution requires a mechanism to detect when an instruction’s operands are ready. Data flow machines often include a separate “matching unit” to synchronize input operands. Unfortunately, matching time, in this thesis considered a part of “communication time,” can dominate overall execution time. This may be the major reason data flow machines have not replaced control flow machines: the gains reaped from increased parallelism in data flow programs have been offset by increased communication time.

1.2 Data Flow Notation and Terminology

When we describe data flow programs, we use directed positional graphs, where the source and sink of each edge carry an integer position attribute. [Even79]. Each vertex represents an operation, and each edge represents the flow of data from the producing operation to the consuming operation.

We call each operation (vertex) in a data flow graph an “actor.” A data flow program can be *coarse-grained* or *fine-grained*, depending on the average “size” of its actors. A *large* actor performs many functions in one indivisible action, while a *small* actor performs few. When we have a program with many large actors,

actors, we call it “coarse-grained.” When only small actors comprise a program, we call it “fine-grained.”

For the purposes of our discussion, small actors contain only one machine primitive operation. Large actors contain more than one primitive. Within a large actor, primitives execute in sequence, even if the encapsulated data flow fragment expresses parallelism. In a sense, primitives within a large actor execute as if they were running on a von Neumann machine.

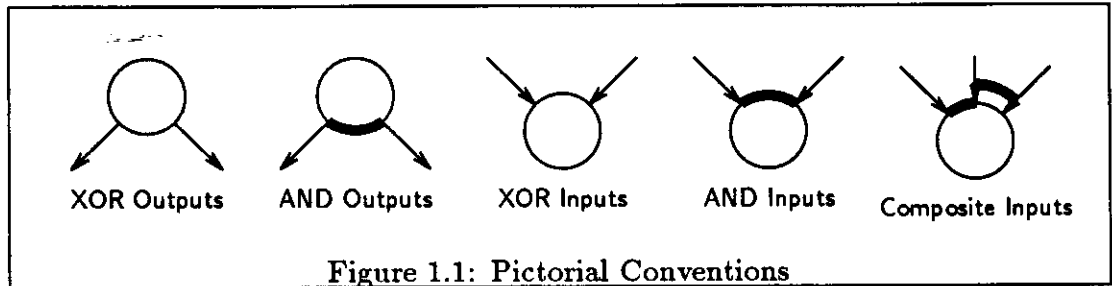
We illustrate partitioning concepts in this thesis by pictorial data flow graphs. Solid-lined circles represent primitive actors. Incoming edges represent the flow of information to an actor, from another actor or from a constant.

We group edges into “enabling groups” and “production groups.” Enabling groups are sets of incoming edges. In one enabling group, all edges share a common sink actor. When tokens rest on all edges in an enabling group, that actor can consume the tokens and begin execution. Several enabling groups can be associated with a given actor.

Production groups are sets of outgoing edges. In one production group, all edges share a common source actor. When an actor completes execution, it sends tokens out on only one production group, one per edge.

We use thick arcs to connect members of enabling groups and production groups.

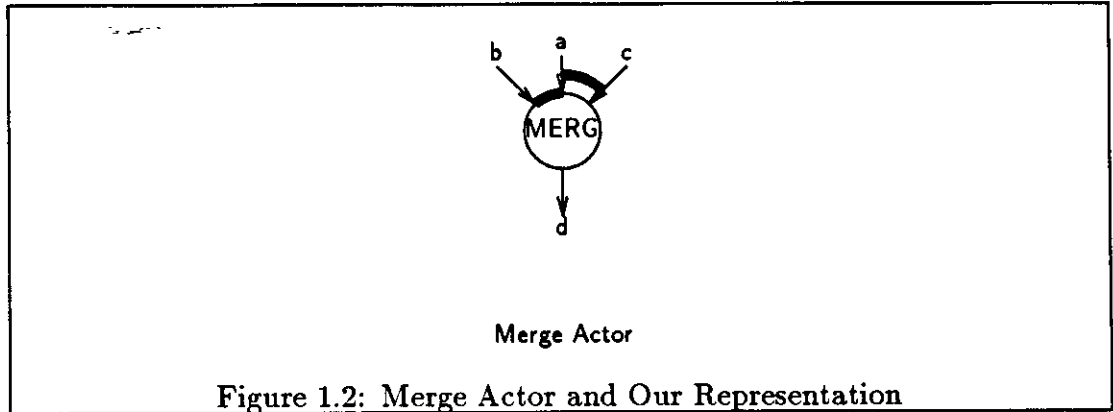
Consider two simple actor input types, AND inputs and XOR inputs. Figure 1.1 shows how these appear in our examples. An actor with AND inputs requires



tokens on both input edges before it will fire. When the actor fires, it consumes a token from each edge, and begins operating on the data items associated with the tokens. An actor with XOR inputs will fire when a token appears on any input edge. When there are tokens on more than one input edge, an actor with XOR inputs selects the input edge at random (in our model, selection probabilities are specified). XOR inputs can be ANDed in our model. We show this by merging two edges to one. Figure 1.1 shows this in the Composite Inputs example.

Consider two simple actor output types, AND outputs and XOR outputs. Refer to Figure 1.1 for pictorial representations. An actor with AND outputs will always produce a single token on each output edge when it completes execution. An actor with XOR outputs will produce a single token on only *one* output edge when it completes. The edge on which an output token is placed depends on the input values to the actor.

Although AND and XOR inputs and outputs account for all actor types in many data flow machines, including the Manchester machine, through these simple constructs we can represent higher level data flow operations. Figure 1.2



shows a typical data flow construct, the MERG operation. Note that the name of the operation is written inside the node. The MERG operation first accepts a boolean input token on input edge *a*. If that value is true, the operation consumes a value from input edge *b*. If false, the operation consumes a value from input edge *c*. The operation merely transmits the consumed value to its output edge *d*.

Constants appear as an edge with a numeric value at the edge's source. In our modeled machine, an actor will encapsulate all input constants, because communication and synchronization time is reduced at no cost. Dynamic data flow machines commonly retain this property [Arvi87] [Gurd85]. For visual clarity, our graphs make it appear that physical tokens transmit constants along an edge, but they don't. Constants are embedded with the actor and are immediately available when the actor needs them. Constants 0.02 and 1 appear in Figure 1.3.

When we show small actors combined into large partitioned actors, we demar-

cate each partition with a dashed box. The small actors within the box execute sequentially, while separate boxes may execute in parallel. Within a box, numbers adjoining each small actor indicate the sequential progression of the actors. The first small actor in the sequence is "1," the second "2," etc.

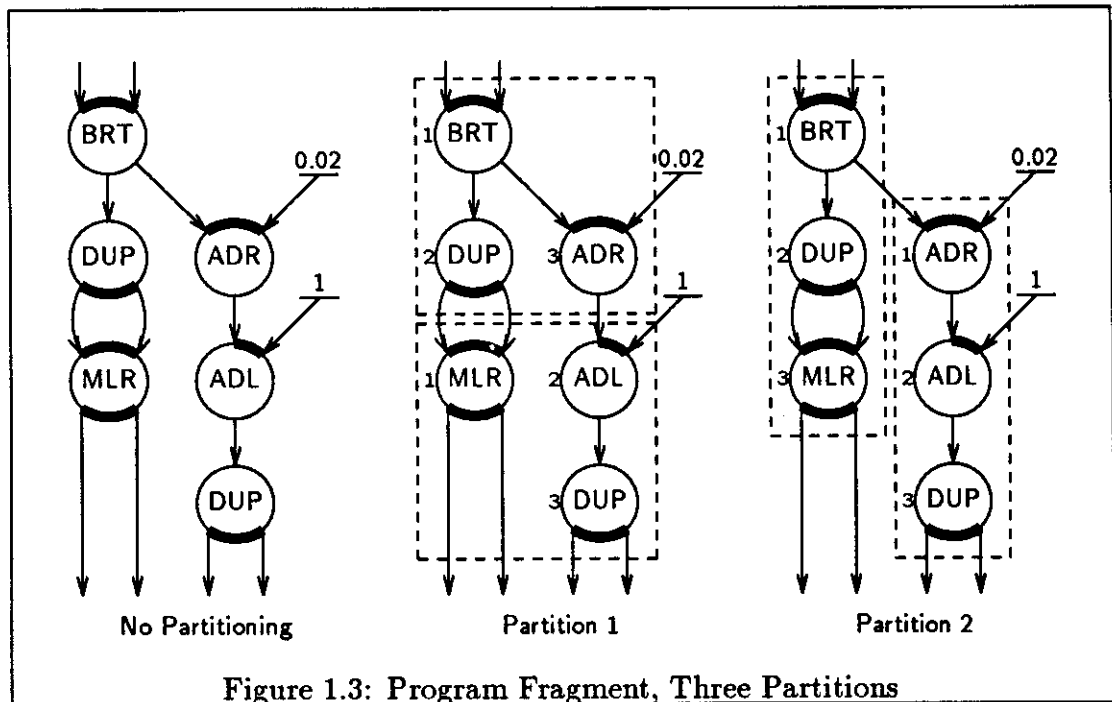


Figure 1.3: Program Fragment, Three Partitions

Figure 1.3 shows the conventions we use to diagram coarse-grained data flow programs. Examine Partition 1. Each dashed box defines a large actor. Three machine-level operations comprise each large actor. In the topmost box, execution proceeds in the order specified by the number at the left of each node: BRT, DUP, ADR. Tokens must appear on each incoming edge of BRT before the actor can begin. Note that the constant 0.02 is embedded inside the ADR. The large actor enclosing it requires only the two input tokens for BRT to begin operation.

Tokens exit the large actor and travel to the matching unit immediately after they are calculated. Thus, execution of our actor proceeds as follows: BRT, DUP, send left result of DUP, send right result of DUP, ADR, send result of ADR.

The bottom block, enclosing MLR, ADL, and DUP, must wait for three incoming tokens before it can begin: the two input tokens for MLR and the single input token for ADL. The constant 1 is embedded in the ADL actor. Note how encapsulation of small actors that would normally run in parallel has severely penalized us: the program fragment in Partition 1 runs completely sequentially, due to non-optimal partitioning.

Small actors in Partition 1 must run in sequence because the bottom actor must wait for all incoming tokens before proceeding. ADR produces its result when the top actor completes. The bottom actor requires that result.

Partition 2, on the other hand, causes no loss of parallelism. The left actor begins with two entering tokens, operation BRT executes, sending its right output token out, then operation DUP executes, et cetera. Immediately after BRT produces its right output token, the rightmost coarse actor can commence.

1.3 The Problem

Tagged-token data flow computers tag each data token with an “instantiation identifier” and a target actor. These instantiation identifiers allow the program to express recursion and iteration easily. When an actor requires multiple input tokens, a “matching unit” assembles the tokens based on equal instantiation

identifiers and target actor address. For more information on tagged-token data flow computers, please see [Arvi87] and [Gurd85].

As we have mentioned, data transmission from one actor to another incurs a large time cost when operands must pass through token matching or queuing units. Thus, coarse-grained data flow programs—where most intermediate data incurs little communication and synchronization delays—often run faster than equivalent fine-grained programs, even though fine-grained programs can introduce greater parallelism. Machines with long transmission or matching delays exacerbate the effect [Chan81] [Leco81], as shown by both simulations [Gaud82] [Huda85] and mathematical constructions [Gaud84].

In experiments on 29 different numerical analysis programs, Manchester machine researchers discovered that unary instructions comprised between 56% and 70% of the total instructions executed [Gurd85]. They modified the Manchester machine to bypass the matching store when an instruction with a unary output followed an instruction with a unary input, essentially creating a coarse-grained actor, to reduce overall execution time.

The problem of identifying “unary-output followed by unary-input” constitutes a special case of the problem of identifying “natural sequences in a data flow program.” Precedence relations can force data flow program fragments that include n -ary (not just unary) operations to run sequentially.

We extend Gurd and Watson’s work to identify and combine naturally sequential, small, n -ary actors into large actor blocks, avoiding unnecessary com-

munication and matching delays.

Using unfolded acyclic execution graphs, [Gaud84] shows that combining small actors into large sequential blocks, even when that parallelism is reduced, can improve the execution time of a data flow program. But to obtain an execution graph, one must evaluate an entire program—flattening the loops and conditional branches that occur in the unevaluated data flow graph. Gaudiot’s analysis does not extend to cyclic data flow program graphs, a standard representation of data flow programs. By analyzing execution graphs he limits the usefulness of his results: execution graphs show the nature of the program *after* input values have been specified and the program has been run. And for long-running, highly-parallel programs, execution graphs become prohibitively large.

1.4 Thesis Contributions

We base our work on a hypothetical data flow machine where each indivisible (i.e., uninterruptible) actor can be a single primitive operation or a composition of several operations. The Manchester machine, since it is microprogrammable, provides a concrete example.

We identify sequential subsections of fine-grained data flow graphs and encapsulate them into maximal sequential blocks or “partitions,” which form large actors. In doing this, we preserve inherent parallelism (unlike Gaudiot), while we attempt to reduce overall communication and matching delays to a mini-

mum. We show how to obtain the set of *all* maximal sequential partitionings of a data flow graph. We often find several alternatives to choose from, each with a different execution time.

To help us select a good alternative, we create a stochastic model of data flow program execution, called “probabilistic data flow graphs.” We provide a method for transforming probabilistic data flow graphs to Markov chains, allowing us to evaluate and compare the performance of different partitions without actually executing the program. We show that we must deal with closed subsets in the resulting Markov chain, an unfortunate consequence of modeling cyclic data flow programs as if they ran stochastically.

Finally, after obtaining a set of partitionings, converting them to a Markov representation, removing closed Markov subsets, and evaluating the resulting Markov chains, we choose a good partitioning by finding the one which has the lowest expected execution time.

To help us judge the model’s usefulness, we wrote a suite of programs:

1. A T-language [Slad87] program which reads a probabilistic data flow graph, obtains the set of all maximal sequential partitions, converts each to a Markov chain, removes closed subsets according to our algorithm, and produces the resulting Markov transition probability matrix.
2. A general purpose Markov chain solving program, which produces stationary probabilities and mean return times. The mean return time of a

program's start state is its estimated execution time.

3. A tagged-token data flow machine simulator, based loosely on the philosophy of the Arvind and Manchester machines. The simulator provides true execution times for comparison.

We ran two example data flow programs through our suite. One called "Integrate" is adapted from a program which runs on the Manchester machine [Gurd85]. The other, called "Recursive_AQ" is the data flow object of a SISAL program provided by Lawrence Livermore National Laboratory [McGr85].

As expected, execution time estimates provided by the modeling system diverged from the actual execution time. However, in each example case, the modeling system chose the best partitioning, and substantially reduced overall execution time.

CHAPTER 2

Previous Research

Several researchers have discussed and developed ideas about data flow program granularity. In this chapter we document and discuss several papers that specifically address granularity issues. We then outline related work in performance modeling.

2.1 Research on the Effects of Granularity

Gaudiot and Ercegovac study the granularity of acyclic binary-tree data flow graphs [Gaud84] [Gaud85]. They show that reductions in parallelism can speed up a data flow program. Their work did not consider cyclic graphs, a difficult problem. But they reduced parallel constructs to sequential blocks, a topic we will not explore. Our work deals exclusively with the recognition and selection of naturally sequential blocks in cyclic data flow programs.

Hudak and Goldberg discuss “serial combinators,” which explores ways to identify and combine sequential sections of lambda expressions [Huda85]. Their method fails to recognize a few non-obvious serial combinations. Our discussion explores the nature of a “heuristic,” for selecting “serial combinators,” which their work mentions but does not explain.

Others made sketchy descriptions of machines that handled high granularity data flow programs, but no performance results (or even estimates) were given [Chan81] [Leco81].

2.1.1 Gaudiot's High Granularity Data Flow Ring

In [Gaud85], Gaudiot and Ercegovic present a well-analyzed argument for using coarse-grained data flow programs. They explain that large data flow actors can consolidate several low-level operations that would otherwise execute sequentially. This consolidation helps decrease communication and matching store delays without eliminating parallelism. In addition, one *can* combine low-level operations that would otherwise execute in parallel, and still decrease the overall execution time through the elimination of matching store operations and the transmission of data.

[Gaud84] constructs an analytic model to predict the performance of a variable resolution data flow machine. They restrict their analysis to unfolded execution graphs. That is, they use a data flow graph with no loops or conditional operations. The model has limited practical value to data flow compiler writers, due to this restriction. Unfolded execution graphs can only be formed from the execution of a folded data flow graph. Thus, a compiler writer would have to execute the program before he could optimize it. This may be impractical when considering programs that execute many operations.

This paper first constructs a ring-based variable resolution data flow machine

[Thom78], [Gost80]. Using a binary tree data graph they obtain execution time improvements of about 40% by reducing communication costs through elimination of parallelism.

However, the research does not develop techniques necessary to partition data flow programs into optimally large actors. Nor does it apply to the problem of partitioning unevaluated data flow graphs with embedded conditionals and loops.

2.1.2 Serial Combinators

Hudak and Goldberg [Huda85] discuss *serial combinators* for evaluation of functional programs. Serial combinators are large actors which combine small actors which would execute sequentially. Thus, a program with serial combinators avoids the communication delays that would occur between small actors, but retains the parallelism inherent in the program.

They present an algorithm that partitions a lambda expression to serial combinators. They assert this partitioning is complete, that is, that one cannot create larger serial combinators without sacrificing parallelism.

But the algorithm presented in [Huda85] does not optimally partition sequential sections of the program. One part of the algorithm makes an arbitrary choice for the subexpression to retain in a combinator. The selection of one over another will affect overall execution time, which they do not account for. We present an extended algorithm which chooses the best path, given accurate statistical information.

2.2 Performance Modeling

Besides the theoretical aspects of granularity specifically addressed by the research discussed above, other works discuss the performance modeling of data flow programs and the machines they run on. In choosing the optimal set of sequential partitions for a data flow graph, we require performance modeling tools. Thus, we present here a brief overview of previous work in modeling parallel programs.

2.2.1 Kapelnikov's System Model

[Kape86] develops a complex analytic model for predicting the performance of specific programs on general multiprocessing machines. Using Markov analysis, he predicts the performance of smaller Markov chains. Then to analyze larger systems, he forms aggregates of smaller Markov chains into large exponential server systems. These large servers approximate the original chain. Their exponential nature makes it possible to use them as nodes in more comprehensive continuous time Markov systems.

Kapelnikov divides a machine-plus-program system into two subnetworks, the P/C (Processing and Communication) subnetwork and the M/U (Matching and Updating) subnetwork. The P/C subnetwork performs the following tasks: receiving operands and program segments from the M/U subnetwork, selecting processing units for execution, executing program segments, and sending results

to the M/U subnetwork. For example, the P/C subnetwork in a Manchester machine handles token transfer from the matching unit, fetches instructions from the instruction store, sends them to the processing unit, processes them, and sends them to the token queue [Gurd85]. See Figure 2.1.

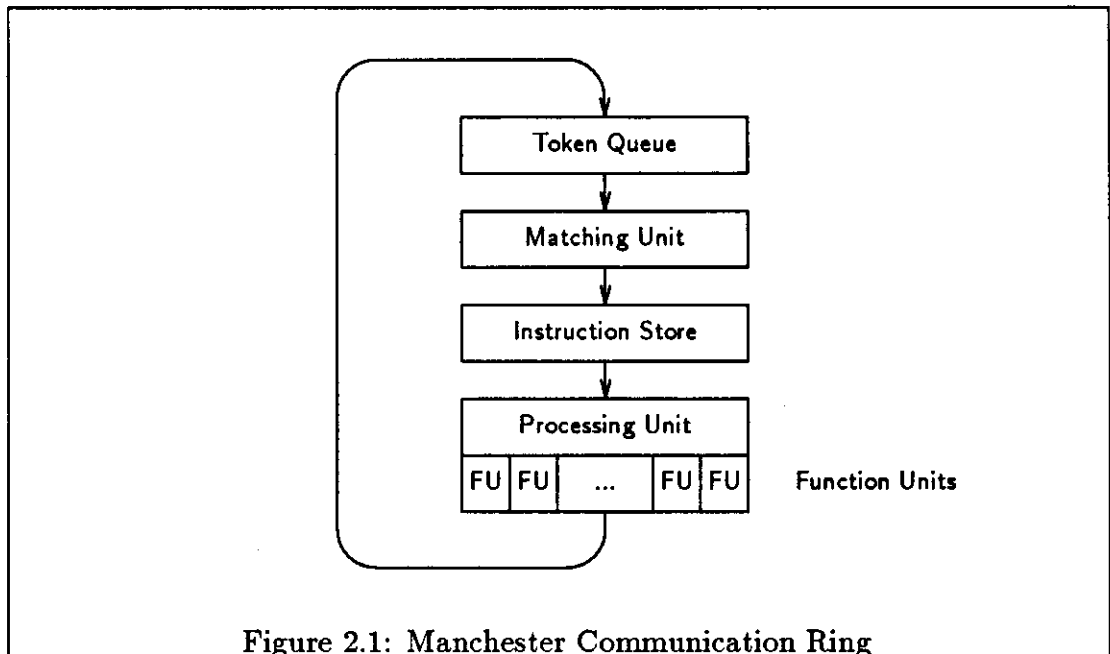


Figure 2.1: Manchester Communication Ring

The M/U subnetwork service centers account for delays in contention, access, use and update of the “synchronization subsystem.” The synchronization subsystem in a Manchester machine is the token queue, the matching unit and its associated overflow unit.

Kapelnikov points out that when the system contains a shared, central synchronization facility, one need not analyze the M/U subnetwork separately. It can be merged into the P/C subnetwork. In fact, our Manchester-like machine

uses a shared, central matching unit. We can avoid the extra complexity of analyzing two distinct units by merging our analysis of the matching store and token queues with the remainder of the machine.

Kapelnikov constructed “computational control graphs,” which stochastically model program behavior. We unknowingly and independently constructed a similar stochastic graph structure in this thesis. We called them “stochastic” or “probabilistic” data flow graphs. They are non-deterministic data flow graphs where the edge taken from an “OR output” is determined strictly by fixed probabilities.

Kapelnikov allows a great deal more flexibility in constructing graph models than we do. He dealt with a broader topic—generating performance models for all classes of multiprocessing systems, whereas we use our model specifically to analyze data flow graph running on a Manchester machine.

Kapelnikov does not consider one problem with computational control graphs. It is possible for the Markov chain model to form proper closed subsets that do not contain the start state. These proper closed subsets might better be called “deadlocks,” for once the system enters a state within the proper closed subset, it cannot complete. When a Markov chain contains a proper closed subset, the expected completion time will be infinite.

To handle this problem we provide an algorithm for deleting proper closed subsets from a Markov chain. We believe this will more accurately model the behavior of a *correct* data flow program.

Proper closed subsets appear because stochastic data flow graphs are only approximations of actual data flow programs. Typical data flow operators program are wholly deterministic—the input values to an operation determine which output edges will produce values. When we adopt a stochastic model, we must estimate the probability that an operator will produce a value on a particular output edge. Using a stochastic model makes the problem of estimating performance computable, but it must be made clear that this is only an approximation.

2.2.2 Thomasian and Bay Task System Performance Analysis

Like Kapelnikov, Thomasian and Bay [Thom85] [Thom86] construct a model for evaluating the performance and behavior of parallel computations. Their work is limited to analysis of systems with acyclic task precedence relationships. Thomasian’s work provides methods for hierarchical decomposition.

In contrast, this thesis will analyze systems with cyclic task relationships, and does not discuss hierarchical decomposition. We discuss Thomasian’s work, in part, to provide motivation for future work in hierarchical decomposition based on the discrete time model developed in this thesis.

Because Thomasian deals with continuous-time Markov approximations, he can represent complex relationships between probabilistic and deterministic precedence by introducing dummy tasks, with zero execution time. Our model, however, uses discrete-time Markov chains—zero execution time vertices cannot be used. Therefore, we will provide explicit representations for these complex

relationships.

Thomasian considers a simple scheduler which preassigns tasks to host computers, and activates tasks as precedence and passive resource constraints are satisfied. That satisfies our model as well.

2.2.2.1 Task Decomposition

Thomasian assumes a closed product-form queueing network, and shows that one can construct an exact low-cost solution through convolution or mean value analysis [Lave83].

In considering only acyclic graphs, Thomasian's work is constrained to task systems that do not loop. With loop-free systems, the resulting Markov chain is also acyclic, and can be expressed as a triangular matrix. Triangular matrices can be solved in linear time.

We consider cyclic graphs in this thesis, thus the solution time for our model is of the order $O(n^3)$, where n is the total number of vertices and edges in a data flow graph. Closed proper Markov chain subsets cannot occur with acyclic data flow graphs, but they can with cyclic data flow graphs. Thus, we will develop a scheme for trimming closed proper subsets from the resulting Markov chain.

Thomasian's work in decomposition may also apply to cyclic discrete-time graphs, but we leave that topic to future research.

CHAPTER 3

Modeling Data Flow Program Performance

In this chapter, we formally describe probabilistic data flow graphs, and their relation to data flow programs. We construct a mapping from probabilistic data flow graphs to discrete-time discrete-space Markov chains.

We show that the constructed Markov chains may contain proper closed subsets. These subsets produce catastrophic failure in the model. We describe a method for removing these from the Markov chain.

In later chapters we will use the performance estimates obtained from this model to improve how we partition actors into sequential blocks. We must point out that our stochastic system is a *heuristic* for approximating the execution time of a system. Several inaccuracies are introduced by this approximation—we discuss them at the end of this chapter and in Chapter 5.

3.1 Probabilistic Data Flow Graphs

In this section, we develop a formal descriptive model to characterize the statistical behavior of data flow programs. We add transition probabilities to data flow graphs [Karp69] [Karp66] [Camp87], calling them *probabilistic data flow graphs* (PDFGs).

3.1.1 Introduction

In our graphs, each vertex corresponds to a machine operation, and each edge corresponds to the communication of a value from its source vertex to its sink vertex. Each edge has two ordinal positions, at the source vertex and the sink vertex. If edge e has sink position i , edge e supplies the value for the i th operand of edge e 's sink vertex. If edge e has source position j , edge e receives its value from the j th output of edge e 's source vertex.

Each edge or vertex holds a set of *tokens*. Like Petri nets [Pete77], probabilistic data flow graphs can be *bounded* or *unbounded*. A bounded probabilistic data flow graph enforces a maximum number of tokens per edge or vertex, while an unbounded PDFG places no limit on the number of tokens per edge or vertex. As with unbounded Petri nets, unbounded PDFGs make analysis of most systems impossible.

We further restrict our analysis to *safe probabilistic data flow graphs*. In a safe PDFG, an edge may contain at most one token. This restriction simplifies

our modeling task.

A PDFG executes as follows: Before a vertex may begin executing, there must be one token available on each of a set of *enabling edges*, which we call an *enabling set*. When tokens rest on each edge in an enabling set, we say the set is *ready*. A vertex may have several enabling sets. The vertex consumes a token from each edge in one enabling set.

After a delay it puts a token on each edge in set of *production edges* called a *production set*. One vertex may have several production sets, but a vertex may place tokens on only one production set at a time. A production set is called *ready* if none of its edges carry tokens. When a vertex wants to place tokens on a not-ready production set, the vertex halts until the production set is ready.

Each enabling set and production set carries a rational weight value. The probability that a ready enabling set will fire is the ratio of its weight over the total weight of all *ready* enabling sets on the same vertex. The probability that tokens will be produced on a production set after its source vertex fires is the ratio of its weight over the total weight of all production sets on the same vertex. We require neither enabling sets nor production sets to be disjoint.

For example, suppose vertex v has three enabling sets $\bar{E}_1 = \{a, b, c\}$, $\bar{E}_2 = \{c, d\}$, and $\bar{E}_3 = \{e, f\}$ with weights $w(\bar{E}_1) = 0.3$, $w(\bar{E}_2) = 0.5$, and $w(\bar{E}_3) = 0.2$. Suppose tokens rest on edges a, b, c, e and f . Then enabling sets \bar{E}_1 and \bar{E}_3 enable vertex v . We compute the probability that vertex v will consume tokens from

\overline{E}_1 in Equation 3.1.

$$p(\overline{E}_1) = \frac{w(\overline{E}_1)}{w(\overline{E}_1) + w(\overline{E}_3)} \quad (3.1)$$

Unlike enabling set probabilities, production set probabilities are independent of which sets are ready. Thus, if $\overline{E}_1, \dots, \overline{E}_n$ are production sets for vertex v , the probability that production set \overline{E}_i will receive tokens after vertex v fires is given by Equation 3.2.

$$p(\overline{E}_i) = \frac{w(\overline{E}_i)}{\sum_{j=1}^n w(\overline{E}_j)} \quad (3.2)$$

We must point out that a probabilistic model can only be an approximation to a deterministic system (like a data flow program). Our model occasionally results in catastrophic modeling failures, namely closed proper subsets in the generated Markov chain. We provide a mechanism for correcting this problem (See Section 3.2.2).

3.1.2 Formal Description

Here, we formalize the notions expressed loosely in the previous section. This will help us examine partitioning algorithms.

We characterize a data flow program by a directed positional graph, $G = (V, C, E, \overline{P}_s, \overline{P}_t, \tau, F)$, called a *probabilistic data flow graph*. Graph G consists of the following:

1. A finite set of *vertices* V . Each $v \in V$ represents a single primitive actor in a static data flow graph.

2. A finite set of *constant generators* $C \subseteq V$. Each $c \in C$ represents a primitive actor that place an output token whenever its production set is ready. A constant generator produces tokens in zero time.
3. A finite set of *edges* $E \subset V \times \mathbf{Z}^+ \times (V \setminus C) \times \mathbf{Z}^+$. Each edge represents a communication path from one actor to another. If $e \in E, e = \langle v, i, v', i' \rangle$, then we call v the *source vertex* of edge e , i the *source vertex position* of edge e , v' the *sink vertex* of edge e , i' the *sink vertex position*, ordered pair $\langle v, i \rangle$ the *source point* of edge e , and $\langle v', i' \rangle$ the *sink point* of edge e .

Our definition of E disallows edges with sinks which are constant generators. In other words, constant generators cannot have input edges.

Equations 3.3 and 3.4 ensure that all edges $e \in E$ have unique source points and unique sink points.

$$\forall(\langle v, i, v', i' \rangle \in E), \langle v, i, v'', i'' \rangle \in E \Rightarrow (v' = v'') \wedge (i' = i'') \quad (3.3)$$

$$\forall(\langle v', i', v, i \rangle \in E), \langle v'', i'', v, i \rangle \in E \Rightarrow (v' = v'') \wedge (i' = i'') \quad (3.4)$$

Equation 3.5 restricts constant generators to a single output edge.

$$\forall(i \in \mathbf{Z}^+ \setminus \{1\}, c \in C), \langle c, i, v', i' \rangle \notin E \quad (3.5)$$

4. A finite set of *enabling groups* $\bar{P}_s \subset 2^E \times \mathbf{Z}^+$. Each enabling group $w \in \bar{P}_s$ designates a set of input edges to a single vertex, and the weight associated with that set.

Equation 3.6 constrains subsets of E marked with a single enabling group to those sharing a common sink vertex.

$$\forall(\langle \bar{E}, r \rangle \in \bar{P}_s), (\langle v_1, i_1, v'_1, i'_1 \rangle \in \bar{E} \wedge \langle v_2, i_2, v'_2, i'_2 \rangle \in \bar{E}) \Rightarrow (v'_1 = v'_2) \quad (3.6)$$

Equations 3.7 and 3.8 ensure that a unique weight marks every enabling set, and every edge $e \in E$ is marked by at least one enabling group.

$$\forall(\langle \bar{E}, r \rangle, \langle \bar{E}', r' \rangle \in \bar{P}_s), \langle \bar{E}, r \rangle \neq \langle \bar{E}', r' \rangle \Rightarrow (\bar{E} \neq \bar{E}') \quad (3.7)$$

$$\forall(e \in E), \exists(\langle \bar{E}, r \rangle \in \bar{P}_s), e \in \bar{E} \quad (3.8)$$

5. A finite set of *production groups* $\bar{P}_t \subset 2^E \times (0, 1]$. Each production group in \bar{P}_t designates a set of output edges from a single vertex, and the weight associated with that set. Equation 3.9 constrains subsets of E marked with a single production group to those sharing a common source vertex.

$$\forall(\langle \bar{E}, r \rangle \in \bar{P}_t), (\langle v_1, i_1, v'_1, i'_1 \rangle \in \bar{E} \wedge \langle v_2, i_2, v'_2, i'_2 \rangle \in \bar{E}) \Rightarrow (v_1 = v_2) \quad (3.9)$$

Equations 3.10 and 3.11 ensure that a unique weight marks every production set, and every edge $e \in E$ is marked by at least one production group.

$$\forall(\langle \bar{E}, r \rangle, \langle \bar{E}', r' \rangle \in \bar{P}_t), \langle \bar{E}, r \rangle \neq \langle \bar{E}', r' \rangle \Rightarrow (\bar{E} \neq \bar{E}') \quad (3.10)$$

$$\forall(e \in E), \exists(\langle \bar{E}, r \rangle \in \bar{P}_t), e \in \bar{E} \quad (3.11)$$

6. A time delay function $\tau: ((V \setminus C) \cup E) \rightarrow \mathbf{Z}^+$. For any $v \in (V \setminus C)$, $\tau(v)$ is the time cost of performing the operation. Note that actual primitive operations occasionally require different execution times for different operand

values. In that situation, we must decompose the operation into smaller fixed-time primitives.

7. A set of termination vertices $F \subseteq (V \setminus C)$. When any of these vertices are enabled by δ (see below), the PDFG is said to have terminated.

Execution of a PDFG is represented by a series of *instantaneous descriptions* $I_i = (\delta_i, H_i)$ which consists of the following:

1. A *residual firing time function* $\delta_i: (V \setminus C) \cup E \rightarrow \mathbf{Z}^{0+} \cup \{-1\}$. We designate δ_0 the *start instantaneous description* of the PDFG.

For any $v \in (V \setminus C)$, $\delta(v)$ is the remaining time before v will fire. $\delta(v) = -1$ indicates that the vertex is in a quiescent state, not processing any data.

For any $e \in E$, $\delta(e)$ is the remaining time before a token on edge e will be made available to target vertices. $\delta(e) = -1$ indicates the edge does not contain a token.

Equation 3.12 ensures that the values of the residual firing time function do not exceed those of the time delay function.

$$\forall(a \in (V \setminus C) \cup E), \delta(a) \leq \tau(a) \quad (3.12)$$

Our definition precludes storing more than one token on an edge.

2. A *holding set* $H_i \subseteq \overline{P}_i$ of production groups, which satisfies Equation 3.13.

$$\overline{p} = \langle \overline{E}, r \rangle \in H_i \Rightarrow \delta_i(\text{vert}(\overline{p})) = 0 \wedge \exists(e \in \overline{E})\delta_i(e) \neq -1 \quad (3.13)$$

The holding set retains the production sets which should receive tokens from vertices which are ready to fire. That is, when a vertex completes, a production set is selected. If that production set is not ready to receive tokens (because at least one of its edges is not quiescent), the vertex must “remember” the production set. It cannot “forget” and choose a different production set because this would not correspond to the way deterministic data flow programs work.

3.1.2.1 Additional Definitions

Let $\overline{P}_s^v = \{(\overline{E}, r) \in \overline{P}_s \mid \forall (a, i, b, j) \in \overline{E}, b = v\}$ denote the enabling sets for vertex v . Let $\overline{P}_t^v = \{(\overline{E}, r) \in \overline{P}_t \mid \forall (a, i, b, j) \in \overline{E}, b = v\}$ denote the production sets for vertex v .

We define the *source vertex function* $source: E \rightarrow V$ for $\langle v, i, v', i' \rangle \in E$ such that $source(\langle v, i, v', i' \rangle) = v$. We define the *sink vertex function* $sink: E \rightarrow V$, such that $sink(\langle v, i, v', i' \rangle) = v'$. We define the *source position function* $sourcep: E \rightarrow V$ for $\langle v, i, v', i' \rangle \in E$ such that $sourcep(\langle v, i, v', i' \rangle) = i$. We define the *sink position function* $sinkp: E \rightarrow V$ for $\langle v, i, v', i' \rangle \in E$ such that $sinkp(\langle v, i, v', i' \rangle) = i'$.

We define the input edge function $I: V \rightarrow 2^E$ and the output edge function $O: V \rightarrow 2^E$ in Equations 3.14 and 3.15, respectively.

$$I(v) = \{e \in E \mid \exists (v' \in V), \exists (i, i' \in \mathbf{Z}^+), \langle v', i, v, i' \rangle \in E \wedge e \in E\} \quad (3.14)$$

$$O(v) = \{e \in E \mid \exists(v' \in V), \exists(i, i' \in \mathbf{Z}^+), \langle v, i, v', i' \rangle \in E \wedge e \in E\} \quad (3.15)$$

Equations 3.16 and 3.17 constrain the source vertex positions and the sink vertex positions, respectively, for a given vertex to a sequential numbering starting at 1.

$$\forall(v \in V), \forall(i \in \mathbf{Z}^+, 1 \leq i \leq |I(v)|), \exists(j \in \mathbf{Z}^+, v' \in V), \langle v', j, v, i \rangle \in E \quad (3.16)$$

$$\forall(v \in V), \forall(i \in \mathbf{Z}^+, 1 \leq i \leq |O(v)|), \exists(j \in \mathbf{Z}^+, v' \in V), \langle v, i, v', j \rangle \in E \quad (3.17)$$

Each vertex $v \in V$, designates a node in the program graph. We define functions $pred: V \rightarrow 2^V$ and $succ: V \rightarrow 2^V$ in Equations 3.18 and 3.19, respectively.

$$pred(v) = \{\bar{v} \in V \mid \exists(e \in E), e \in I(v) \wedge e \in O(\bar{v})\} \quad (3.18)$$

$$succ(v) = \{\bar{v} \in V \mid \exists(e \in E), e \in O(v) \wedge e \in I(\bar{v})\} \quad (3.19)$$

If $a = (\bar{E}, r) \in \bar{P}_s \cup \bar{P}_t$, Equation 3.20 defines a function $vert: \bar{P}_s \cup \bar{P}_t \rightarrow V$.

$$vert(a) = \begin{cases} v_1 & \text{if } a \in \bar{P}_s \wedge \forall(e \in \bar{E}) sink(e) = v_1 \\ v_2 & \text{if } a \in \bar{P}_t \wedge \forall(e \in \bar{E}) source(e) = v_2 \end{cases} \quad (3.20)$$

By Equations 3.6 and 3.9, when $a \in \bar{P}_s \cup \bar{P}_t$, $vert(a)$ has a unique, defined value. If $a \in \bar{P}_s$, then $vert(a)$ returns the unique sink vertex associated with edges in enabling group a . If $a \in \bar{P}_t$, then $vert(a)$ returns the unique source vertex associated with edges in production group a .

3.1.3 PDFG Execution

We characterize the operation of a probabilistic data flow graph with a sequence of instantaneous descriptions derived from the initial state δ_0 . Here we

describe the generation of successive instantaneous descriptions.

We say an instantaneous description $I = (\delta, H)$ *directly yields* $I' = (\delta', H')$ iff Equations 3.21 through 3.31 hold. These equations will derive δ' from δ as follows: We first construct a sequence of instantaneous descriptions starting from I where edges and vertices fire, but where no time passes. The last instantaneous description in the sequence, I_n , will be in a state where no edge or vertex may fire until time passes. We then derive I' from I_n by incrementing our “clock” by one time unit.

Equation 3.21 disallows a direct yield operation when I is a terminating instantaneous description.

$$\forall(f \in F), \delta(f) = -1 \quad (3.21)$$

There exist a sequence of groups $P_* = (P_1, P_2, \dots, P_{n-1})$ with $P_i \in \overline{P}_s \cup \overline{P}_t$, a sequence of potential enabling groups $R_{s,*} = (R_{s,1}, \dots, R_{s,n-1})$ with $R_{s,i} \subseteq \overline{P}_s$, a sequence of potential production groups $R_{t,*} = (R_{t,1}, \dots, R_{t,n-1})$ with $R_{t,i} \subseteq \overline{P}_t$ and a corresponding sequence of instantaneous descriptions $I_* = (I_1, I_2, \dots, I_n, I')$ such that $I_1 = I$ and for each $i \in \{1, \dots, n-1\}$, Equations 3.23 through 3.27 hold.

Equations 3.22 through 3.24 fire edges in a single enabling set. Equation 3.22 constructs a set, $R_{s,i}$, of enabling groups which can fire. Equation 3.23 selects an enabling group $P_i = \langle \overline{E}_i, r_i \rangle$ from that set. Each edge $e \in \overline{E}_i$ is ready to fire

$(\delta_i(e) = 0)$ and the sink vertex for enabling group P_i is quiescent.

$$R_{s,i} = \{ \langle \overline{E}, r \rangle \in \overline{P}_s \mid \overline{E} \subseteq \delta_i^{-1}[[0]] \wedge \delta_i(\text{vert}(P_i)) = -1 \} \quad (3.22)$$

$$P_i = \langle \overline{E}_i, r_i \rangle \in \overline{P}_s \Rightarrow P_i \in R_{s,i} \quad (3.23)$$

Equation 3.24 defines the state of the PDFG after the edges in \overline{E}_i have fired.

The sink vertex takes on its starting time delay, the edges in the enabling set lose their tokens, and the state does not change for other edges and vertices. For

$$\delta_{i+1}: (V \setminus C) \cup E \rightarrow \mathbf{Z}^{0+} \cup \{-1\},$$

$$\delta_{i+1}(a) = \begin{cases} \tau(a) & \text{if } a = \text{vert}(P_i) \\ -1 & \text{if } a \in \overline{E}_i \\ \delta_i(a) & \text{otherwise} \end{cases} \quad (3.24)$$

Equations 3.25 through 3.27 fire a single vertex. Equation 3.25 constructs a set, $R_{t,i}$, of production groups that can fire. Equation 3.26 selects a production group $P_i = \langle \overline{E}_i, r_i \rangle$. The source vertex for \overline{E}_i is ready to fire, and no holding production set has been designate for this vertex in H_i .

$$R_{t,i} = \{ \langle \overline{E}, r \rangle \in \overline{P}_t \mid \text{vert}(\langle \overline{E}_i, r_i \rangle) \in \delta_i^{-1}[[0]] \wedge \forall (\overline{p} \in H_i) \text{vert}(\overline{p}) \neq \text{vert}(P_i) \} \quad (3.25)$$

$$P_i = \langle \overline{E}_i, r_i \rangle \in \overline{P}_t \Rightarrow P_i \in R_{t,i} \quad (3.26)$$

Equation 3.27 defines the instantaneous description of the PDFG after vertex $\text{vert}(P_i)$ has fired or P_i has been placed in the holding set. Each edge in the production set takes on its starting time delay, the vertex loses its token or P_i is

added to H_{i+1} , and the state does not change for other edges and vertices. Let $\widehat{H}_i = H_i \cup P_i$. We define $\delta_{i+1}: (V \setminus C) \cup E \rightarrow \mathbf{Z}^{0+} \cup \{-1\}$ by Equation 3.27.

$$\delta_{i+1}(a) = \begin{cases} \tau(a) & \text{if } \exists (\langle \overline{E}, r \rangle \in \widehat{H}_i) a \in \overline{E} \wedge \overline{E} \subseteq \delta_i^{-1}[-1] \\ -1 & \text{if } \exists (\langle \overline{E}, r \rangle \in \widehat{H}_i) a = \text{vert}(\langle \overline{E}, r \rangle) \wedge \overline{E} \subseteq \delta_i^{-1}[-1] \\ \delta_i(a) & \text{otherwise} \end{cases} \quad (3.27)$$

We define the next holding set, $H_{i+1} \subseteq \overline{P}_s \cup \overline{P}_t$ by Equation 3.28.

$$H_{i+1} = \{ \langle \overline{E}, r \rangle \in \widehat{H}_i \mid \exists (e \in \overline{E}) e \notin \delta_i^{-1}[-1] \} \quad (3.28)$$

Finally, Equations 3.29, 3.30 and 3.31 must hold for the terminating instantaneous description in the sequence. Equation 3.29 states that there is no enabling set which can fire in δ_n .

$$\forall (\langle \overline{E}, r \rangle \in \overline{P}_s) \overline{E} \not\subseteq \delta_n^{-1}[0] \vee \delta_n(\text{vert}(\langle \overline{E}, r \rangle)) \neq -1 \quad (3.29)$$

Equation 3.30 states that there is no vertex which can fire in δ_n .

$$\forall (\langle \overline{E}, r \rangle \in \overline{P}_t) \text{vert}(\langle \overline{E}, r \rangle) \notin \delta_n^{-1}[0] \vee \overline{E} \not\subseteq \delta_n^{-1}[-1] \quad (3.30)$$

Equation 3.31 completes one time step. Define $\delta': (V \setminus C) \cup E \rightarrow \mathbf{Z}^{0+} \cup \{-1\}$ such that Equation 3.31 holds.

$$\delta'(a) = \begin{cases} -1 & \text{if } \delta_n(a) = -1 \\ \max(0, \delta_n(a) - 1) & \text{otherwise} \end{cases} \quad (3.31)$$

Finally, Equation 3.32 completes the construction of I' .

$$I' = (\delta', H_n) \quad (3.32)$$

3.1.4 Transition Probabilities

For any yield operation $I \rightarrow I'$ we can compute a probability $p(I, I')$. In general, we compute this using the yield sequences we constructed in the previous section.

For a yield operation $I \rightarrow I'$, we can construct a set of *yield sequences*, $Y(I, I')$, satisfying Equation 3.33.

$$Y(I, I') = \{I_* | \exists (n \in \mathbf{Z}^+) I_* = (I, \dots, I_n, I')\} \quad (3.33)$$

This is the set of all possible sequences from I which directly yield I' .

Pick any sequence $I_* = (I, \dots, I_n, I') \in Y(I, I')$. Equation 3.34 gives the probability, $p(I_i, I_{i+1})$, that a particular transition from I_i to I_{i+1} occurs in I_* .

For $1 \leq i < n$,

$$p(I_i, I_{i+1}) = \frac{w(P_i)}{\sum_{P \in (R_{s,i} \cup R_{t,i})} w(P)} \quad (3.34)$$

Equation 3.35 gives the probability that sequence I_* will occur from starting instantaneous description I .

$$p(I_*) = \prod_{i=1}^{n-1} p(I_i, I_{i+1}) \quad (3.35)$$

Finally, Equation 3.36 gives the probability that a particular yield operation $I \rightarrow I'$ will occur.

$$p(I \rightarrow I') = \sum_{I_* \in Y(I, I')} p(I_*) \quad (3.36)$$

We note that the \rightarrow operation and the probability function p describe a Markov chain. We can make use of this to predict the execution time of a data flow

program.

3.1.5 Fitting the Model to the System

Since each edge must have two endpoints, we establish these conventions: Where the program receives a value from the external environment, we create a source vertex with zero residual firing time (i.e., with a resident token). When the program sends a value to the external environment, we create a sink vertex, where edges enter the vertex but do not leave.

The model cannot deal with multiple tokens per edge or vertex because it is safe. One cannot accommodate this by simply multiplying the number of edges in the model by the maximum number of expected tokens per edge in the data flow machine. However, replications of the system can model a fixed maximum number of tokens on each edge.

Unfortunately, this can increase the number of states exponentially with the number of tokens accommodated, and does not significantly increase the power of the model. In this thesis, we impose a maximum of one token per edge.

We must also point out that recursion cannot be handled adequately by our model. We can approximate it by assigning a fixed time cost to each recursive call. In a later section, we will show a recursive example program, Recursive_AQ. While the modeled execution-times were extremely inaccurate for Recursive_AQ, the relative ordering of the execution-time estimates and the partitioning results were correct.

Transition probabilities of distinct data flow actors are assumed independent by the Markov construction. Dependencies in the original data flow program can cause the Markov construction to produce inaccurate execution time predictions.

3.1.6 Obtaining Probability Estimates

A detailed exposition on how to obtain the probability estimates for a data flow program is beyond the scope of this thesis. However, we will briefly mention two methods, the first based on extrapolated empirical data, the second based on analysis.

Using simulations of low execution-time sample data sets, one can obtain transition frequencies. Extrapolating these transition frequencies to normal execution-time data sets, one can obtain probabilities.

A second method involves probability estimates made by the programmer. Often a cursory analysis of a program can produce reasonable transition probability estimates. Those estimates can be included as pragmatic remarks in source code for the program.

3.2 Interpreting the Markov Model

Figure 3.1 shows an example PDFG with weights assigned to each enabling and production group. The decimal fractions shown in Figure 3.1 refer either to the relative weight of an enabling set, if the numbers appear at the sink of a set of edges, or the weight of a production set, if the numbers appear at the source of a set of edges. Where no weight accompanies a group, the weight is 1.

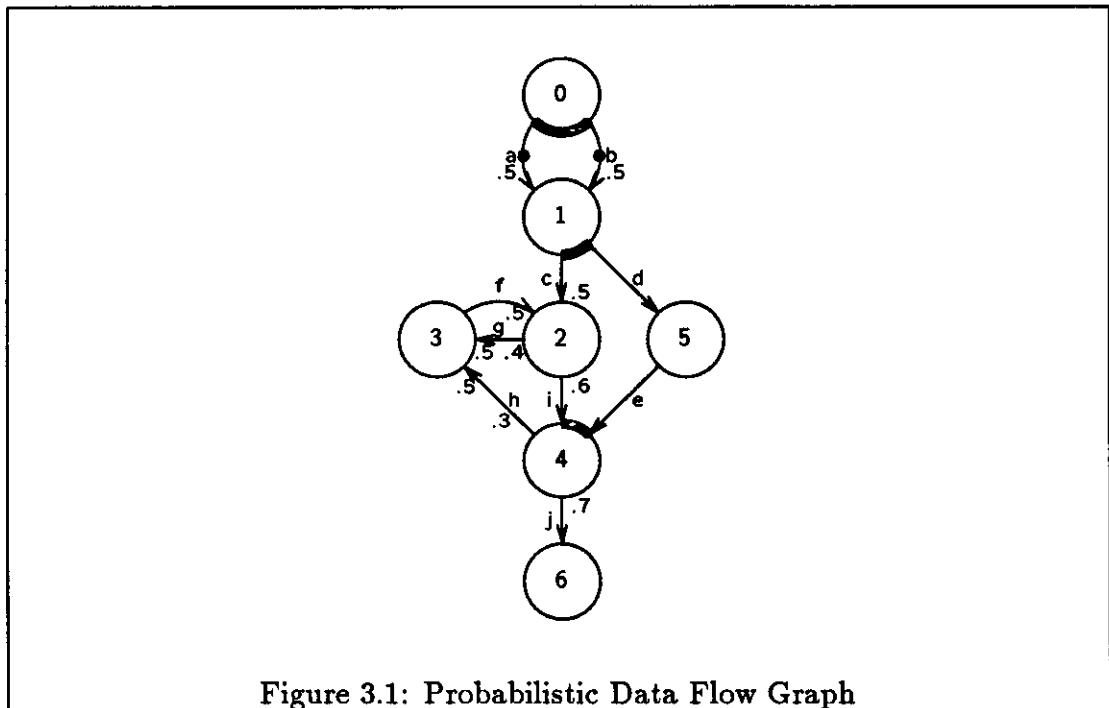


Figure 3.1: Probabilistic Data Flow Graph

For convenience, we assume that edges and vertices in this PDFG have a fixed delay time of 1, so Equation 3.37 holds. This gives us a simpler and more easily explained Markov chain.

$$\forall(a, b \in (V \setminus C) \cup E) \tau(a) = \tau(b) = 1 \quad (3.37)$$

Vertex (0) represents the “start node” in Figure 3.1. Two initial tokens reside on edges (a) and (b). In another modeling system, both initial tokens might be shown residing on the same edge. As mentioned before, we disallow more than one token per edge in our safe PDFG model.

Vertex (1) can absorb a token from either edge (a) or edge (b), with equal probability. After it operates on the input token it produces two output tokens on edges (c) and (d). Vertex (2) absorbs an input token from either edge (c) or edge (f), operates on the input token, and either produces an output token on edge (g) with probability 0.4, or produces an output token on edge (i) with probability 0.6.

Vertex (3) absorbs a token from either edge (h) or edge (g), operates on the input token, and produces an output token on edge (f). Vertex (4) requires tokens resident on both edge (i) and edge (e) before it can begin. It absorbs the two input tokens, operates on them, and either produces an output token on edge (h) with probability 0.3 or produces an output token on edge (j) with probability 0.7.

Vertex (5) simply absorbs a token from edge (d), operates on it, and produces an output token on edge (e).

Vertex (6) represents a termination state. When it absorbs a token from edge (j), the program has completed.

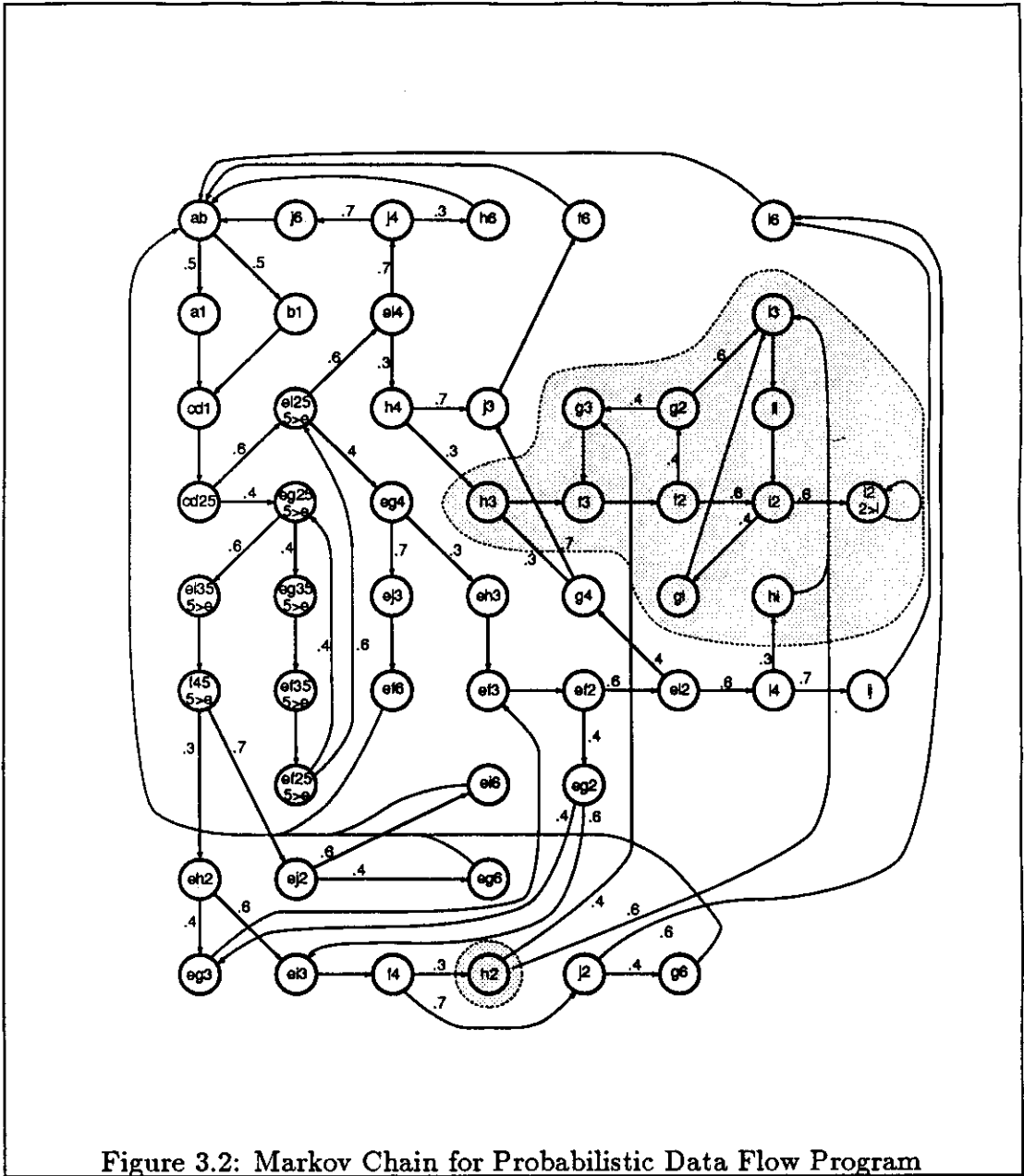
3.2.1 Generating a Markov Chain

Figure 3.2 shows a graphical representation of the Markov chain produced by the probabilistic data flow graph in Figure 3.1. This chain can be obtained by transforming the PDFG to a Markov chain, as discussed in Section 3.1.4. We have a program which performs this transformation. The transitions produced by the program appear in Table 3.1.

We will discuss a few highlights from our example chain to elucidate the general method.

The chain was generating assuming that each operation (vertex) takes a single cycle and each data communication plus matching operation (edge) consumes one cycle.

In the upper left corner of Figure 3.2, we see state (ab), the starting state for the system. Vertex (1) can absorb a token from either edge (a) or edge (b) at the next cycle, with equal probability. The 0.5 probability transitions to states (a1) and (b1) model the first cycle of the data flow program's execution, an OR input.



<i>Source</i>	<i>→Sink</i>	<i>Probability</i>	<i>Source</i>	<i>→Sink</i>	<i>Probability</i>
ab	→b1	0.5	i3	→fi	1.0
ab	→a1	0.5	fi	→i2	1.0
a1	→cd1	1.0	j2	→i6	0.6
cd1	→cd25	1.0	j2	→g6	0.4
cd25	→ei25	0.6	g6	→ab	1.0
cd25	→eg25	0.4	i6	→ab	1.0
eg25	→ei35 [5→(e)]	0.6	ei2	→i4	0.6
eg25	→eg35 [5→(e)]	0.4	ei2	→g4	0.4
eg35 [5→(e)]	→ef35 [5→(e)]	1.0	g4	→j3	0.7
ef35 [5→(e)]	→ef25 [5→(e)]	1.0	g4	→h3	0.3
ef25 [5→(e)]	→ei25 [5→(e)]	0.6	h3	→f3	1.0
ef25 [5→(e)]	→eg25 [5→(e)]	0.4	j3	→f6	1.0
eg25 [5→(e)]	→ei35 [5→(e)]	0.6	f6	→ab	1.0
eg25 [5→(e)]	→eg35 [5→(e)]	0.4	i4	→ij	0.7
ei25 [5→(e)]	→ei4	0.6	i4	→hi	0.3
ei25 [5→(e)]	→eg4	0.4	hi	→i3	1.0
eg4	→ej3	0.7	ij	→i6	1.0
eg4	→eh3	0.3	ej3	→ef6	1.0
eh3	→ef3	1.0	ef6	→ab	1.0
ef3	→ef2	1.0	ei4	→j4	0.7
ef2	→ei2	0.6	ei4	→h4	0.3
ef2	→eg2	0.4	h4	→j3	0.7
eg2	→ei3	0.6	h4	→h3	0.3
eg2	→eg3	0.4	j4	→j6	0.7
eg3	→ef3	1.0	j4	→h6	0.3
ei3	→f4	1.0	h6	→ab	1.0
f4	→j2	0.7	j6	→ab	1.0
f4	→h2	0.3	ei35 [5→(e)]	→f45 [5→(e)]	1.0
h2	→i3	0.6	f45 [5→(e)]	→ej2	0.7
h2	→g3	0.4	f45 [5→(e)]	→eh2	0.3
g3	→f3	1.0	eh2	→ei3	0.6
f3	→f2	1.0	eh2	→eg3	0.4
f2	→i2	0.6	ej2	→ei6	0.6
f2	→g2	0.4	ej2	→eg6	0.4
g2	→i3	0.6	eg6	→ab	1.0
g2	→g3	0.4	ei6	→ab	1.0
i2	→i2 [2→(i)]	0.6	ei25	→ei4	0.6
i2	→gi	0.4	ei25	→eg4	0.4
gi	→i3	1.0	b1	→cd1	1.0
i2 [2→(i)]	→i2 [2→(i)]	1.0			

Table 3.1: Transitions for Example PDFG

From state (cd25), vertex (2) can produce a token on edge (g) or edge (i), with probabilities 0.4 and 0.6 respectively. Simultaneously, vertex (5) fires and must place edge (e) into the holding set (see page 28). The transitions to states (ei25,5→e) and (eg25,5→e) model this XOR output example.

From state (cd1), vertex (1) produces two output tokens on edges (c) and (d). Simultaneously, the tokens previously on edges (c) and (d) are absorbed by vertices (2) and (5). The transitions from (cd1) to (cd25) model this AND output example.

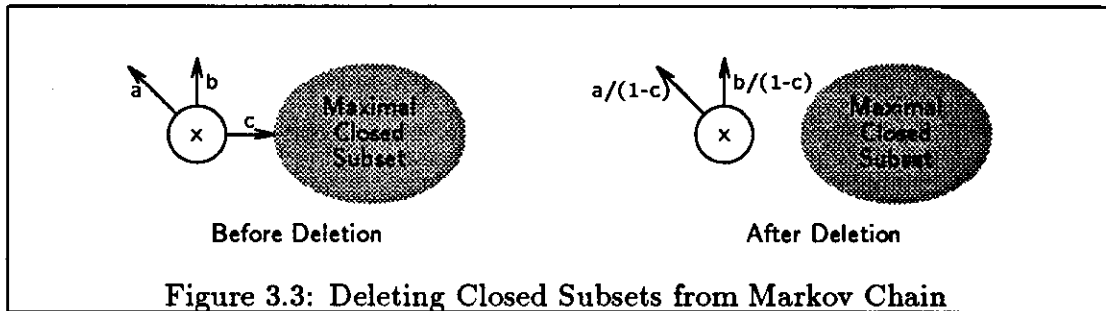
From state (ef2), vertex (4) cannot begin because no token resides on edge i. Vertex (4) has AND inputs. However, the state immediately following (ef2), (ei2), has tokens on edges (i) and (e). Thus, from state (ei2), we can go to states (i4) or (g4).

State (f6), (g6), (h6), (i6) and (j6) are termination states. At these states, the designated "final vertex" has been reached and the data flow program halts. We insert transitions from all final states in the Markov chain to the start state, (ab). This allows us to obtain the estimated execution time for the program.

At states (h3) and (hi), the program enters an unrecoverable infinite loop. The set of states { (h3), (f3), (f2), (i2), (gi), (fi), (i3), (g2), (g3), (hi), (i2,2→i), (h2) } form a proper closed subset of the state space (the shaded region in the graph). Neither the start state nor the end state occupy this state space, thus we have a useless infinite loop. Should a real data flow program enter state (3h) or (hi), the program has a bug.

3.2.2 Closed Subsets in Markov Chains

We define a *maximal closed subset* of a Markov chain as a closed proper subset of the chain which is not itself a proper subset of a closed (not necessarily proper) subset of the Markov chain. For any Markov chain, there is either one maximal closed subset or none.



For our analysis, we assume that the programmer will not intentionally program an infinite loop. We cut infinite loops out of the Markov chain by removing maximal closed subsets and adjusting transition probabilities appropriately. If we do not remove these sections, our expected execution time will be infinite.

Figure 3.3 demonstrates the process of cutting state transitions to maximal closed subsets. We must remove the state transition with probability c from the Markov chain. We set its probability to zero, eliminating the transition. We then normalize the other state transition probabilities associated with this node, by multiplying them by $1/(1 - c)$.

Theorem 1 *Let T be a Markov chain with state space V_i , and let C be a maximal*

closed subset of V_t . Then $\forall v \in (V_t \setminus C), 0 \leq \sum_{i \in C} \mathbf{T}_{v,i} < 1$.

PROOF. We know $0 \leq \sum_{i \in C} \mathbf{T}_{s,i} \leq 1$, since \mathbf{T} is Markovian. Suppose $\exists v \in (V_t \setminus C), \sum_{i \in C} \mathbf{T}_{v,i} = 1$. Then $C \cup \{v\}$ is a closed subset of the Markov chain, and C cannot be a maximal closed subset of V_t , a contradiction. ■

Theorem 1 shows that we can legally cut all transitions into a maximal closed subset. Suppose that Theorem 1 was false. If $\exists v \in (V_t \setminus C), \sum_{i \in C} \mathbf{T}_{v,i} = 1$, then all transitions from state v go to states in maximal closed subset C . When we applied our transition deletion algorithm, we would encounter an indeterminate situation in deleting the last remaining transition to C .

Let directed graph $G_t = (V_t, E_t, \rho_t, s_t)$ represent $n \times n$ stationary transition matrix \mathbf{T} . Graph G_t consists of a finite set of vertices V_t , a finite set of edges E_t , transition probability function $\rho_t: E_t \rightarrow [0, 1]$ and a single start state $s \in V_t$.

1	$S \leftarrow \{s_t\}, F \leftarrow \emptyset$;	S is unvisited nodes not in MCS.
2	while $S \neq \emptyset$ do	;	Find vertices that can reach any $x \in S$.
3	select any $x \in S$;	Visit one vertex.
4	$F \leftarrow F \cup \{x\}$;	Add x to list of found vertices.
5	$S \leftarrow S \setminus \{x\}$;	Remove x from list of unvisited.
6	$S \leftarrow S \cup \{v \in V_t \mid \langle y, x \rangle \in E_t\} \setminus F\}$;	Get vertices that can start this one.
7	end while	;	
8	$M \leftarrow V_t \setminus F$;	Set M to Maximal Closed Subset.

Algorithm FMCS (Find Maximal Closed Subset)

Theorem 2 Algorithm FMCS completes in $O(|V_t| + |E_t|)$ time.

PROOF. Statements (1) and (8) complete in constant time. Statements (2), (3), and (4) repeat at most once per element of V_t . Statement (6) performs at

most, over the complete execution of the program, $|E_t|$ set inclusion operations and $|V_t|$ set-difference operations.

Thus FMCS completes in $O(|V_t| + |E_t|)$ time. ■

There are 52 vertices and 79 edges in Figure 3.2. So the FMCS algorithm applied to that graph would take $O(52 + 79)$ time. The graph resulting from the application of Algorithm FMCS is shown in Table 3.2 and Figure 3.4.

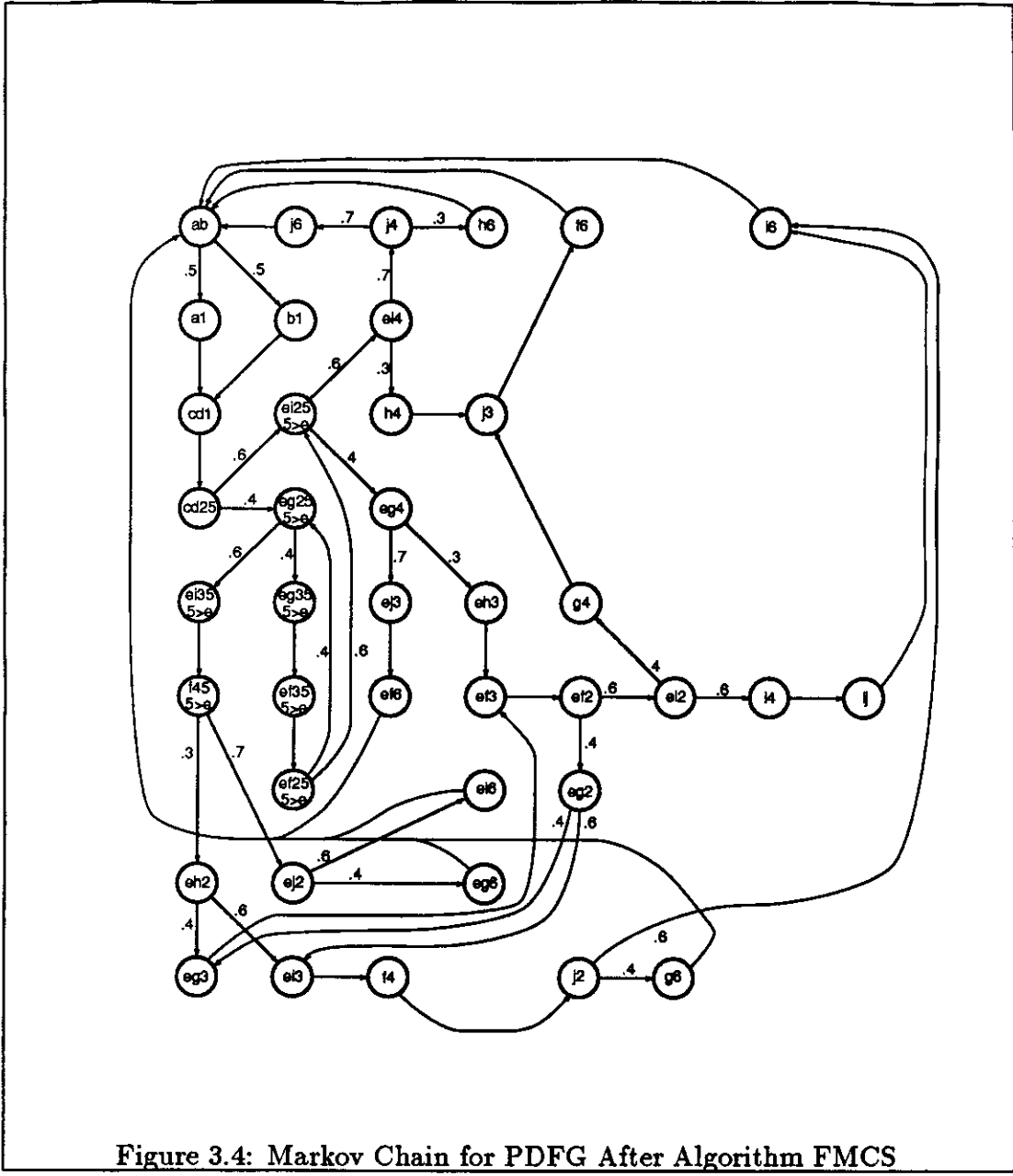


Figure 3.4: Markov Chain for PDFG After Algorithm FMCS

<i>Source</i>	<i>→Sink</i>	<i>Probability</i>	<i>Source</i>	<i>→Sink</i>	<i>Probability</i>
b1	→cd1	1.0	j2	→g6	0.4
ei25	→eg4	0.4	j2	→i6	0.6
ei25	→ei4	0.6	f4	→j2	1.0
ei6	→ab	1.0	ei3	→f4	1.0
eg6	→ab	1.0	eg3	→ef3	1.0
ej2	→eg6	0.4	eg2	→eg3	0.4
ej2	→ei6	0.6	eg2	→ei3	0.6
eh2	→eg3	0.4	ef2	→eg2	0.4
eh2	→ei3	0.6	ef2	→ei2	0.6
f45 [5→(e)]	→eh2	0.3	ef3	→ef2	1.0
f45 [5→(e)]	→ej2	0.7	eh3	→ef3	1.0
ei35 [5→(e)]	→f45 [5→(e)]	1.0	eg4	→eh3	0.3
j6	→ab	1.0	eg4	→ej3	0.7
h6	→ab	1.0	ei25 [5→(e)]	→eg4	0.4
j4	→h6	0.3	ei25 [5→(e)]	→ei4	0.6
j4	→j6	0.7	eg25 [5→(e)]	→eg35 [5→(e)]	0.4
h4	→j3	1.0	eg25 [5→(e)]	→ei35 [5→(e)]	0.6
ei4	→h4	0.3	ef25 [5→(e)]	→eg25 [5→(e)]	0.4
ei4	→j4	0.7	ef25 [5→(e)]	→ei25 [5→(e)]	0.6
ef6	→ab	1.0	ef35 [5→(e)]	→ef25 [5→(e)]	1.0
ej3	→ef6	1.0	eg35 [5→(e)]	→ef35 [5→(e)]	1.0
ij	→i6	1.0	eg25	→eg35 [5→(e)]	0.4
i4	→ij	1.0	eg25	→ei35 [5→(e)]	0.6
f6	→ab	1.0	cd25	→eg25	0.4
j3	→f6	1.0	cd25	→ei25	0.6
g4	→j3	1.0	cd1	→cd25	1.0
ei2	→g4	0.4	a1	→cd1	1.0
ei2	→i4	0.6	ab	→a1	0.5
i6	→ab	1.0	ab	→b1	0.5
g6	→ab	1.0			

Table 3.2: Transitions Minus Closed Subset for PDFG

The chain resulting from application of FMCS is *irreducible* and *homogeneous* [Klei75]. It may not be aperiodic.

3.2.3 Obtaining Expected Execution Time

We determine the expected execution time for the data flow program by obtaining the *mean recurrence time* of the Markov chain start state. Each Markov chain is characterized by a transition probability matrix \mathbf{T} . To obtain the mean recurrence time, we must obtain the stationary probability vector $\mu = \lim_{i \rightarrow \infty} \mu^{(i)}$.

We solve this equation by noting that $\mu = \mu\mathbf{T}$, and if μ is a $1 \times n$ matrix, that $\sum_{i=1}^n \mu_i = 1$. Several algorithms have been produced which solve the resulting system of linear equations in order $\mathcal{O}(t^3)$ time [Heym87] [Harr84].

If state ab is the start state for our data flow program, and μ_1 is the stationary transition probability for state ab , then $1/\mu_1$ is the mean recurrence time for the start state and the expected execution time for the data flow program.

3.2.4 Limitations of the Stochastic Model

Three problems arise from our stochastic model. First, finding production set probabilities is an incomputable problem (discussed later in Theorem 7). With reasonable probability and starting state *estimates*, we can obtain reasonable partitionings. Methods for accurately predicting state transition probabilities exceed the scope of this manuscript, but they normally use program analysis or

the statistical results of simulation.

Second, data flow operations do not operate stochastically. Predicting execution times from a statistical analysis occasionally produces strange artifacts. Often two branches will always fire in the same direction. Our model assumes that each branch will select its output edge with some fixed, independent probability. As a result, programs which include dependent branches cause errors in the model.

The third major limitation of our model results from our prohibition of more than one token residing on an edge or within a vertex. Many present machines allow many token to rest on an edge, waiting for other matching tokens. Likewise, in some machines, many copies of the same actor or vertex can be operating on different sets of tokens simultaneously.

In Chapter 5 we will apply the stochastic model to a dynamic data flow program (which allows multiple tokens per edge). We will compare our execution time estimates with actual execution times. While the problems we discuss above cause errors, for our partitioning examples the model chooses the correct partitions.

CHAPTER 4

Partitioning Data Flow Programs

Using the methods outlined in Chapter 3 we can obtain performance estimates for data flow programs. Our original goal was to find an automatic method for partitioning data flow programs into naturally sequential blocks, or “sequences.” As described in this chapter, one can find several alternative partitionings. We will use the modeling method of Chapter 3 to select a good one, from the set of all possible partitionings.

4.1 Naturally Sequential Blocks

One can partition data flow graphs in several ways. We choose to restrict ourselves to partitioning in a way that preserves all parallelism, but which incorporates as many actors into each sequence as possible. To ensure this, we adopt the following set of rules:

All input tokens coming into a sequence must enter before the sequence starts. If a sequence could partially complete and then wait for a token, deadlock conditions could occur.

Since input tokens must be available before a sequence starts, inherent parallelism in the system could be destroyed by interior actors that require tokens

1	$S \leftarrow \{v \in V \mid (\exists e \in E, \delta_0(e) \neq -1 \wedge \text{sink}(e) = v) \vee \delta_0(v) \neq -1\}$; Establish starting nodes.
2	$S \leftarrow S \cup \{v \in V \mid \overline{P}_s^v > 1\}$; If a $\text{pred}(v)$ may not start v , include v .
3	$M \leftarrow S, j \leftarrow 0$; Initialize.
4	while $S \neq \emptyset$ do	; As long as there remains a start node,
5	select any $s \in S$... select one as this partition's start.
6	$S \leftarrow S - \{s\}, j \leftarrow j + 1, k \leftarrow 0, Q \leftarrow \emptyset, \text{Found} \leftarrow \text{true}$; Update S , get next P , initialize.
7	while Found do	; If found a good successor, continue.
8	$\text{Found} \leftarrow \text{false}, k \leftarrow k + 1, P_{(j,k)} \leftarrow s, Q \leftarrow Q \cup \{s\}$; Go to next vertex, add s to P and Q .
9	for all $v \in V$ such that $v \in \text{succ}(s) \wedge v \notin M$ do	; Use only "good" successor nodes.
10	if $\text{pred}(v) \subseteq M$; Have predecessors been placed?
11	$M \leftarrow M \cup \{v\}$; Yes, this one will be too.
12	if $\text{pred}(v) \subseteq Q \wedge \neg \text{Found}$; Are all predecessors in partition?
13	$\text{Found} \leftarrow \text{true}, n \leftarrow v$; Yes. Put v in partition.
14	else	;
15	$S \leftarrow S \cup \{v\}$; No. Put v in start nodes.
16	end if	;
17	end if	;
18	end for	;
19	$s \leftarrow n$; Set up to put our successor in P .
20	end while	;
21	$P_{(j,k+1)} \leftarrow \epsilon$; Mark end of this partition.
22	end while	;

Algorithm PSB (Partition Sequential Blocks)

from outside the sequence. Therefore, we will *not* incorporate an actor into a sequence if it requires tokens from another sequence and it is not the first actor in the sequence.

However, we do allow any actor (i.e., not necessarily the last actor in a sequence) to produce output tokens which are sent to another sequence. This is easy to implement in hardware. When an actor produces a result, it could be immediately sent to the matching store, for example, rather than wait for the entire sequence to complete.

Algorithm PSB partitions a program graph into the largest possible sequences according to the above rules.

Algorithm PSB first places all vertices that must begin sequential blocks in

set S (statement 1). It uses this criteria: If an *initial* token rests on a vertex's incoming edge (i.e., if residual time $\delta_0(e) \neq -1$) or the vertex itself is processing a token ($\delta_0(v) \neq -1$), that vertex is placed in S . Sequences then grow from these "starting nodes" in the loop beginning at statement 7. If an immediate successor to the current vertex, s , *requires* a token from s to begin execution, and if that successor accepts tokens *solely* from the vertices preceding it in the current sequence (set Q), it is selected as a successor within the sequence.

Any other successor vertices, whose predecessors are entirely in $Q \cup M$, will be added to the set S . These will serve as additional sequence starting nodes.

Upon completion of Algorithm PSB, each P_i , where $1 \leq i \leq j$, comprises one sequential block. If we form large actors from these blocks, we may eliminate some communication delay while preserving all parallelism in the program.

Theorem 3 *Upon completion of Algorithm PSB, two-dimensional array P contains all vertices in V , except unexecutable vertices.*

PROOF. First, we assume that $\forall v \in V, [c \in C \wedge v = \text{end}(c)] \Rightarrow [\exists e \in E, \text{end}(e) = v]$. By this assumption, we disallow vertices executable solely by input constants. Because vertices driven solely by constants have no clear interpretation, and in some machines they could generate an infinite number of output tokens, we exclude them.

By statement 1, we know that the labels of all vertices that might execute immediately after initiation of the program are in S . Statement 7 executes at

least once for every $s \in S$, by statements 4 to 6. No statement in Algorithm PSB removes an element of S , except statement 6. Thus, if at any time during Algorithm PSB's operation, $s \in S$, then by the algorithm's termination $\exists i \in \{1, \dots, k\}, P_{(i,1)} = s$.

Assume that $a \in V$, and a is executable. We want to show that for any $b \in succ(a)$, where b is executable, that $\exists i, j \in \mathbf{Z}^+, P_{(i,j)} = b$.

Suppose $b \in succ(a)$. Then $\exists e \in E, start(e) = a \wedge end(e) = b$. By statements 3 and 11, M contains all vertices that at some point were members of S , or those included in some previously constructed sequence. We know by the above argument that all such elements of S will be in a sequence. Therefore, if $b \in M$, b is in a sequence.

Now consider statement 7. If a can receive a token, at some point the program's s equals a at statement 9. If $b \notin M$, we execute the body of the loop. If $pred(b) \subseteq M$, that is, all predecessors of b are executable and have been placed in P , then by statement 11, b will be added to M , and thus b will be placed in a sequence (either immediately at statement 19, or later through statement 15).

We know that all vertices that can execute immediately upon program initiation are present in P . We know that if a vertex is present in P , and successor vertices will also be present in P . So by induction, the labels of all executable vertices are in P . ■

Theorem 4 *Algorithm PSB preserves the ordering of directed graph G .*

PROOF. To prove this, we have to show that no sequence P_i in P internally violates the ordering of G . Then we must show that externally, no ordering violations can occur.

By statement 12, a vertex v can only be included in P_i if all $\bar{v} \in \text{pred}(v)$ precede it in P_i . Therefore, internally all sequences P_i preserve the ordering of G .

Externally, according to the machine's operation, all incoming tokens must be present before a sequence begins. Therefore, P_i cannot begin until the external ordering is satisfied. ■

Theorem 5 *Algorithm PSB preserves all parallelism present in the original data flow graph.*

PROOF. Assume otherwise. Then $\exists i, j, k \in \mathbf{Z}^+, j < k$, such that $P_{(i,k)}$ precedes $P_{(i,j)}$ in graph G or no ordering exists between $P_{(i,k)}$ and $P_{(i,j)}$.

Theorem 4 implies that no ordering exists between $P_{(i,k)}$ and $P_{(i,j)}$. But by line (9) of PSB, the *succ* relation orders all nodes in the string P_i . With this contradiction, we proved our theorem. ■

Theorem 6 *Algorithm PSB's maximum execution time is $O(|V| + |E|)$.*

PROOF. Clearly statement 1 executes in $O(|E|)$ time, and statement 2 in $O(|E|)$ time. The body of loop 9 iterates, at most, once per $e \in E$ before Algorithm PSB terminates. The body of loop 7 iterates, at most, once per $v \in V$ before Algorithm PSB terminates.

So Algorithm PSB completes in $O(|V| + |E|)$ time. ■

4.2 Execution Times for Different Partitionings

While the use of Algorithm PSB will reduce communication delays (see Chapter 5), the indeterminacy of statement 9 allows several different partitionings of most graphs. In some cases, selecting one partitioning over another will substantially accelerate an algorithm. In this section, we examine criteria for selecting one partitioning over another.

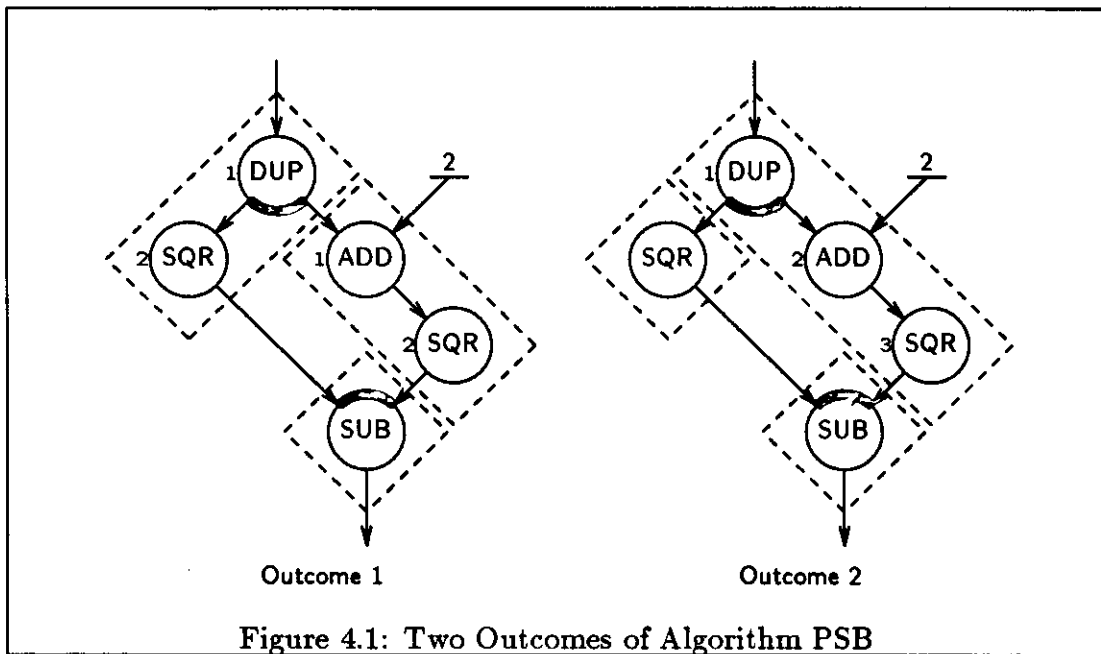


Figure 4.1: Two Outcomes of Algorithm PSB

In Figure 4.1, we see two possible outcomes, depending on which $v \in succ(s)$ statement 9 chooses when $s = DUP$. In Outcome 1, a result token will appear on the left input of SUB earlier than on the right input, because the left path requires

fewer vertices (machine operations) and fewer exposed edges (communication delays) than the right path. In Outcome 2, the two result tokens will appear at SUB at more nearly the same time.

Remark 1 *Outcome 2 of Figure 4.1 is faster than Outcome 1.*

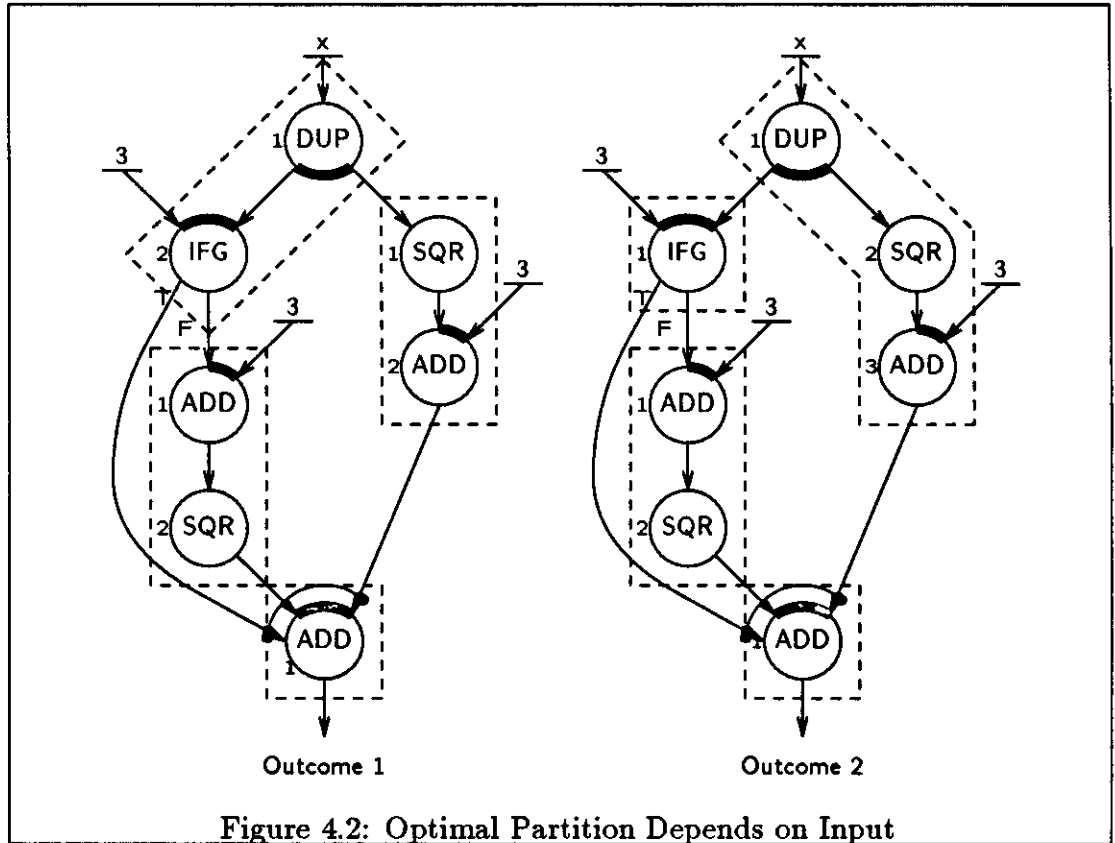
PROOF. Let $t: C \rightarrow \mathbf{R}$, where C is the set of all operation codes and \mathbf{R} is the set of real numbers. If $c \in C$, then $t(c)$ is the execution time of that operation. t_1 is the average time to process 1-input communications. t_2 is the average time to process 2-input communications. $t_1 \leq t_2$ because the latter must go through the matching store, while the former need not [Wats82].

The Outcome 1 program takes $T_1 = t(\text{DUP}) + \max(t(\text{SQR}), t_1 + t(\text{ADD}) + t(\text{SQR})) + t_2 + t(\text{SUB})$ time units. The Outcome 2 program takes $T_2 = t(\text{DUP}) + \max(t_1 + t(\text{SQR}), t(\text{ADD}) + t(\text{SQR})) + t_2 + t(\text{SUB})$ time units. We easily compute that $T_2 = T_1 - t_1 - t(\text{ADD}) + \max(t_1, t(\text{ADD}))$. Since $t_1 > 0 \wedge t(\text{ADD}) > 0$, we see that $T_2 < T_1$. ■

Ideally, we would generalize this concept, generating an algorithm to choose the optimal path. We shall discover this task difficult.

Theorem 7 *Partitioning an unevaluated data flow graph into optimal sequential blocks is incomputable.*

PROOF. We prove this theorem by example. Observe the data flow program fragment in Figure 4.2. There are two possible partitionings under algorithm



PSB. Assume the incoming value $x > 3$, and the execution time of each node is the same. Then the leftmost partitioning completes earlier. Conversely, with $x \leq 3$, the rightmost partitioning completes earlier.

Thus, the optimal graph partitioning depends solely on the value of incoming value x . But the algorithm used to calculate x can be any data flow algorithm. By [Turi36], predicting the resulting value of x is incomputable. Therefore, partitioning a data flow graph into optimal sequential blocks is incomputable.

■

4.3 Obtaining a Solution

Since partitioning is incomputable by Theorem 7, we must rely on approximations. Applying non-deterministic Algorithm PSB to a data flow graph will result in a set of different sequential partitionings.

Execution times will differ between partitionings because intra-sequence communication times will be less than inter-sequence communication times. Using the method described in Chapter 3, we can obtain an estimated completion time for each partitioning, and choose the partitioning which completes in the least amount of time. We describe our experiences with an automatic system we developed, in Chapter 5.

CHAPTER 5

Implementation

In Chapter 3 we gave a notation for describing data flow programs probabilistically. We showed how to convert those to discrete-time, discrete-space Markov systems. We noted that these Markov systems can have proper closed subsets which do not include the start state, and explained why this is a model failure. We gave an algorithm to delete these closed proper subsets from the Markov chain.

We showed in Chapter 4 how one can find all maximal partitionings for a data flow graph preserving parallelism. We gave a high-level description of the algorithm. We presented an example program which would run with different execution times depending on the partitioning used. We showed that finding the *optimal* partitioning is an incomputable problem, and the best we can do is find a good heuristic. We suggested that the PDFG/Markov model provided in Chapter 3 might provide a reasonable answer.

Here, we merge the ideas of the previous two chapters. We built a program that accepts a probabilistic data flow graph specification and analyzes different partitionings for it. To compare our results with the execution times on a real machine, we wrote a tagged-token data flow machine simulator. In this chapter,

we describe both systems in detail and present the results from two example data flow programs.

5.1 The Analysis Program

Written in the Lisp language variant called “T” [Slad87], our analysis program reads a PDFG description and produces the transition probability matrix for an equivalent Markov chain. It performs the following operations:

1. Computes the set of all maximal partitionings for the program according to Algorithm PSB.
2. For each partitioning created, it
 - (a) Sets the execution time of edges internal to a sequence to zero. This corresponds with zeroing the communication time between actors which share a common sequence, in an actual data flow machine.
 - (b) Writes the names of the zeroed edges to a file.
 - (c) Converts the resulting PDFG to a Markov chain according to the algorithm described in Chapter 3.
 - (d) Applies Algorithm FMCS to the Markov chain.
 - (e) Outputs the resulting modified Markov chain to a file.
3. Lastly, the analysis program performs operations 2c through 2e on the original PDFG.

To complete the analysis, we wrote a program in the C language which computes the stationary probability vector and the state return times for the Markov chain output in step 2e. This program uses the method described in [Heym87] to perform these calculations directly (rather than by successive approximation).

5.2 Input Format

The analysis program described above, and the simulator program described in Appendix D, use the same input format to describe a data flow program. A description of that input format follows:

The input format is designed to carry all information required by both the analysis program and the simulator. Therefore, some of the information in the common description format is superfluous to the analyzer (such as the actual *value* of the initial token on an edge).

Likewise, some information is superfluous to the simulator. For example, the ability to state that a vertex is in a partial state of completion (with $\delta_0(v) \neq -1$) is completely ignored by the simulator. The concept of a residual time is inherent in the PDFG description for both edges and vertices, but implementing it as part of the initial state of the *simulator* would not be worth the effort. None of the examples documented in this thesis specify vertices in partial states of completion, as initial states.

The input consists of a series of Lisp-style *s-expressions*, followed by the keyword “end”. Each specifies a particular component of the data flow program.

(*edge name time residual*)

(*edge name time residual value*)

The **edge** command specifies an edge in the graph. It has a label, *name*, which can consist of any non-blank characters.

Time must be an integer greater than or equal to 0. It specifies the time required to send a token from the source vertex of this edge to the sink vertex. When the edge carries a value between vertices contained in the same sequence, we will set *time* to zero.

Residual must be an integer greater than or equal to -1. If it is -1, no *value* parameter should be specified. A *residual* of -1 means there is no token resting on the edge in its initial state. If *residual* \neq -1, it indicates that a token is resting on the edge in its initial state, and that *residual* cycles remain before the *value* will reach the edge's sink vertex.

Value must be specified if *residual* \neq -1. If *value* is a sequence of digits, it is tagged as an integer. If *value* is a sequence of digits with a decimal point, it is tagged as a real number. If *value* is **TRUE** or **FALSE**, it is given the corresponding boolean value. If *value* begins and ends with the character "'", it is considered a string. Otherwise, it is invalid.

(*vertex name instruction time residual enabling producing*)

Name specifies a unique label. When *instruction* field is **SUBR**, the vertex may be called as a subroutine, using the label given in *name*.

Instruction is used only by the simulator, and indicates the instruction to be executed when this vertex fires.

Time and *residual* correspond the execution time and residual execution time of the vertex (see Chapter 3).

Enabling specifies the enabling groups for the vertex, in Lisp *s-expression* format. Each enabling set is a list. The first value is the floating-point weight associated with the enabling set. The remaining values are the names of input edges. *Enabling* is a list of enabling groups. So, for example, if edges **a** through **d** have been defined, “((1 **a b c**) (3.5 **d**))” specifies two enabling groups. The first has three input edges, **a** through **c**, with weight 1. The second has one input edge, **d**, with weight 3.5. The weights are ignored by the simulator.

Producing specifies the production groups for the vertex, in the same format as *enabling*.

The system requires all edges to be defined, before they are referenced in a **vertex** command.

(**constantvertex** *name value producing*)

This command defines a special vertex, which produces constants on demand. It is like the **vertex** command, with an implied *time* of 0, a permanent residual time of zero, and a null enabling set.

Value has the format outlined above in the description of the **edge** com-

mand.

Producing must contain only one production group, with only one edge.

The weight specified for the producing group is ignored by both the simulator and the analyzer.

The tokens produced by this vertex will match anything targeted for the same instruction, in the simulator program's matching store. In the analyzer program, a constant vertex is "absorbed" into its sink (constants are assumed immediately available to target vertices).

(finalvertex name enabling)

This command defines a special vertex with an implied *time* of 0, a *residual* of -1, and a null *producing* value.

When this vertex is enabled, both the simulator and the analyzer will assume program termination.

5.3 Example Programs

Both Program 1 and Program 2 were originally written in SISAL, a stream-oriented, Pascal-like applicative language. We used the retargetable SISAL compiler developed by Lawrence Livermore National Laboratory [McGr85] to generate data flow program object, and converted the object by hand to our input format.

5.3.1 Program 1: INTEGRATE

The INTEGRATE program is derived from an example discussed in [Gurd85], and converted to SISAL version 1.2. The source follows:

```
define Integrate
function Integrate (returns real)
  for initial
    int := 0.0;
    y   := 0.0;
    x   := 0.02
  while
    x < 1.0
  repeat
    int := 0.01 * (old y + old y);
    y   := old x * old x;
    x   := old x + 0.02
  returns
    value of sum int
  end for
end function
```

The resulting data flow input for our analysis and simulation programs follows:

```
(edge =a 1 0 0.0)
(edge =b 1 0 0.0)
(edge =c 1 0 0.02)
(edge =d 1 0 0.02)
(edge =e 1 -1)
(edge =f 1 -1)
(edge =g 1 -1)
(edge =h 1 -1)
(edge =i 1 -1)
(edge =j 1 -1)
(edge =k 1 -1)
(edge =l 1 -1)
(edge =m 1 -1)
(edge =n 1 -1)
(edge =o 1 -1)
(edge =p 1 -1)
(edge =q 1 -1)
(edge =r 1 -1)
(edge =s 1 -1)
(edge =t 1 -1)
(edge =u 1 -1)
(edge =v 1 -1)
(edge =w 1 -1)
```

```

(edge =x 1 -1)
(edge =y 1 -1)
(edge =z 1 -1)
(edge =aa 1 -1)
(edge =ab 1 -1)
(edge =ac 1 -1)
(edge =ad 1 -1)
(edge =ae 1 -1)
(edge =af 1 -1)
(edge =c0 0 -1)
(edge =c1 0 -1)
(edge =c2 0 -1)
(edge =c3 0 -1)
(edge =c4 0 -1)
(edge =c5 0 -1)
(edge =c6 0 -1)
(edge =s0 0 -1)
(finalvertex *F ((1 =af)))
(vertex *32 NOP 0 -1 () ((1 =a)))
(vertex *33 NOP 0 -1 () ((1 =b)))
(vertex *34 NOP 0 -1 () ((1 =c)))
(vertex *35 NOP 0 -1 () ((1 =d)))
(constantvertex *C0 1.0 ((1 =c0)))
(vertex *0 CGR 1 -1 ((1 =c0 =g)) ((1 =h)))
(vertex *1 DUP 1 -1 ((1 =h)) ((1 =m =o)))
(vertex *2 BRR 1 -1 ((1 =n =f)) ((0.02 =s0)(0.98 =l)))
(vertex *3 BRR 1 -1 ((1 =m =e)) ((0.02 =j)(0.98 =k)))
(vertex *4 DUP 1 -1 ((1 =o)) ((1 =n =p)))
(vertex *5 BRRdt 1 -1 ((1 =p =i)) ((0.02 )(0.98 =q =r)))
(vertex *6 DUP 1 -1 ((1 =q)) ((1 =s =t)))
(constantvertex *C1 0.02 ((1 =c1)))
(vertex *7 ADR 1 -1 ((1 =r =c1)) ((1 =u)))
(vertex *8 MLRd 1 -1 ((1 =s =t)) ((1 =v =w)))
(constantvertex *C2 1 ((1 =c2)))
(vertex *9 ADL 1 -1 ((1 =u =c2)) ((1 =x)))
(vertex *10 ADR 1 -1 ((1 =l =v)) ((1 =y)))
(vertex *11 DUP 1 -1 ((1 =x)) ((1 =ad =ac)))
(constantvertex *C3 0.01 ((1 =c3)))
(vertex *12 MLR 1 -1 ((1 =y =c3)) ((1 =z)))
(constantvertex *C6 1 ((1 =c6)))
(vertex *13 ADL 1 -1 ((1 =w =c6)) ((1 =ab)))
(vertex *14 ADR 1 -1 ((1 =k =z)) ((1 =ae)))
(constantvertex *C4 1 ((1 =c4)))
(vertex *15 ADL 1 -1 ((1 =ae =c4)) ((1 =aa)))
(constantvertex *C5 0 ((1 =c5)))
(vertex *16 SIL 1 -1 ((1 =j =c5)) ((1 =af)))
(vertex *20 MERG 0 -1 ((1 =a)(1 =aa)) ((1 =e)))
(vertex *17 MERG 0 -1 ((1 =b)(1 =ab)) ((1 =f)))
(vertex *18 MERG 0 -1 ((1 =c)(1 =ac)) ((1 =g)))
(vertex *19 MERG 0 -1 ((1 =d)(1 =ad)) ((1 =i)))
(vertex *S0 STUB 0 -1 ((1 =s0)) ())
end

```

<i>Version</i>	<i>Analysis</i>	<i>Simulation</i>
No partitioning	602.1 cycles	761 cycles
Partitioning # 1	286.0 cycles	459 cycles
Partitioning # 2	301.3 cycles	507 cycles

Table 5.1: INTEGRATE: Analysis vs. Simulation

The analysis program identified two partitionings for this program:

Partition 1

(*19) (*6 *8 *13) (*10 *12) (*14 *15)
 (*5 *7 *9 *11) (*18 *0 *1 *4) (*2 *S0)
 (*3 *16 *F) (*17) (*20)

Zeroing edges:

=W =T =S =Y =AE =X =U =R =0 =H =G =S0 =AF =J

Partition 2

(*19) (*7 *9 *11) (*5 *6 *8 *13) (*10 *12)
 (*14 *15) (*18 *0 *1 *4) (*2 *S0)
 (*3 *16 *F) (*17) (*20)

Zeroing edges:

=X =U =W =T =S =Q =Y =AE =0 =H =G =S0 =AF =J

Results of the analysis and simulation programs are shown in Table 5.1

We find that the analysis program estimated the execution time of INTEGRATE with an error between 20.8% and 40.6%. The error was kept relatively low because INTEGRATE is not recursive.

Note that the best-case partitioning (partition # 1) improves execution speed by only 9.4% over the worst-case partitioning (partition # 2), while the worst-case beats no partitioning by 18.5%.

5.3.2 Program 2: RECURSIVE_AQ

This program is taken from an example shown in [McGr85], a recursive adaptive quadrature program which integrates an arbitrary function, using a supplied stop condition.

We inserted the subroutines necessary to integrate the function $x^2 + 3x - 8$ from $x = 0$ to $x = 10$. It should be noted that the actual input values, which to a great degree determine the actual running time of this program, are not known to the analyzer.

```
define Recursive_AQ

type Interval =
  record [ X_Low, Fx_Low, X_High, Fx_High : real ];
type Interval_List = array[ Interval ]

function Evaluate_Function( X: real returns real )
  (X * X) + (3.0 * X) - 8.0
end function

function Stop_Condition( Area_1, Area_2,
  Interval_Width: real returns boolean )
  (abs(Area_1 - Area_2) < 2.5) & (Interval_Width < 1.0)
end function

function Recursive_AQ( L, Leftv, R, Rightv: real
  returns real, boolean )
  let
    Mid := (L + R) * 0.5;
    Midv := Evaluate_Function(Mid);
    Prev_area := (R - L) * (Rightv + Leftv) * 0.5;
    New_Area := (R - Mid) * (Rightv + Midv) * 0.5
              + (Mid - L) * (Midv + Leftv) * 0.5;
    Done := Stop_Condition(Prev_Area, New_Area, R-L );
    Abort := is error(New_Area) | is error(Done)
  in
    if Abort then Prev_Area, true
    elseif Done then New_Area, false
    else
      let
        Left_Area, Abt_Left
          := Recursive_AQ( L, Leftv, Mid, Midv);
        Rgt_Area, Abt_Rgt
```

```

:= Recursive_AQ( Mid, Midv, R, Rightv);
in
Left_Area + Rgt_Area , Abt_Left | Abt_Rgt
end let
end if
end let
end function

```

Rather than show the 234 line data flow input for this SISAL program here, we provide a graphic description in Figure 5.1. A full listing of the data flow program is supplied in Appendix B.

The results of our analysis and simulation programs appear in Table 5.2. The detailed listing of the partitions is supplied in Appendix C.

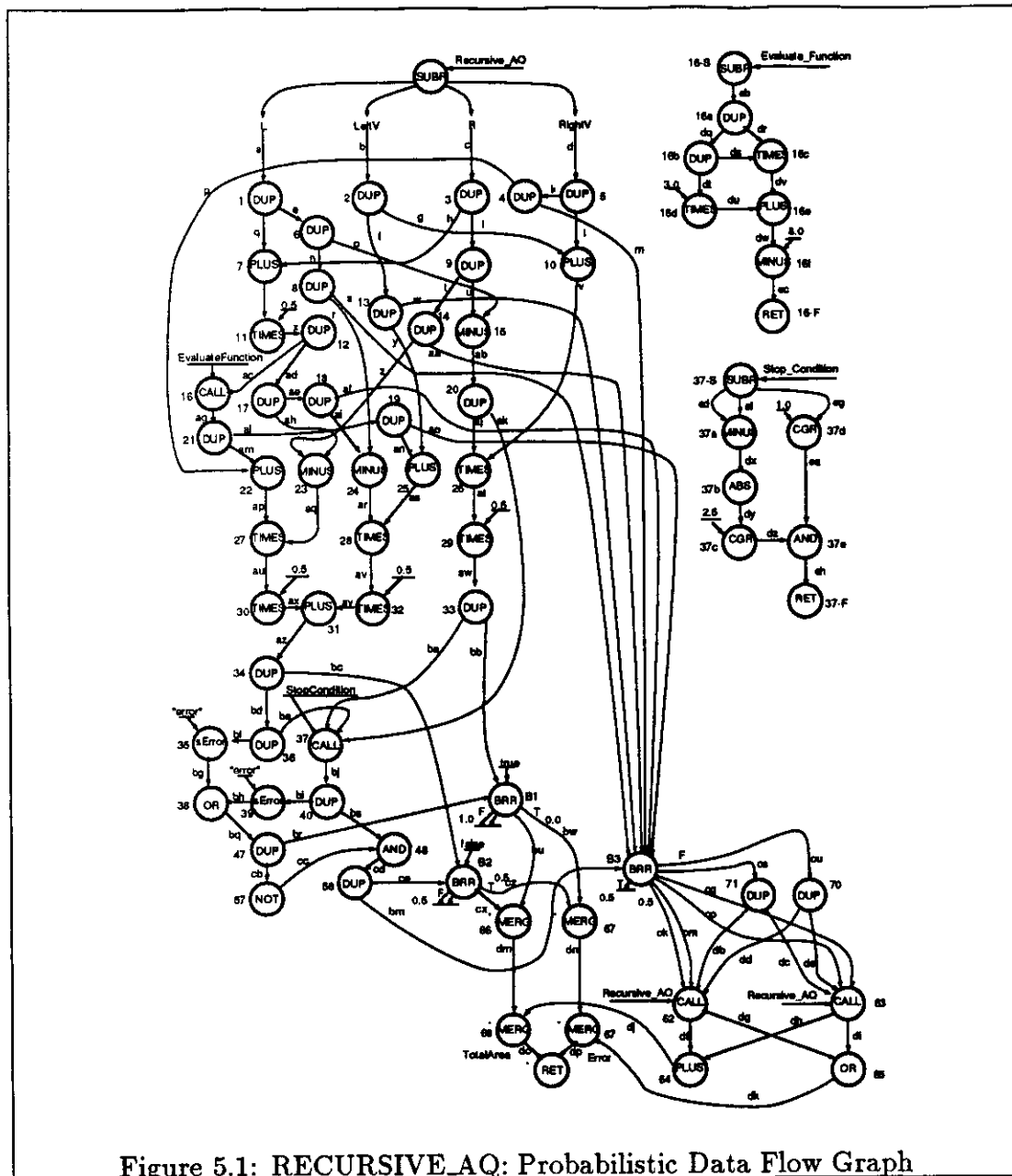


Figure 5.1: RECURSIVE_AQ: Probabilistic Data Flow Graph

<i>Version</i>	<i>Analysis</i>	<i>Simulation</i>	<i>Version</i>	<i>Analysis</i>	<i>Simulation</i>
No partitioning	65.3 cycles	324 cycles	Partitioning # 33	52.3 cycles	259 cycles
Partitioning # 01	52.3 cycles	259 cycles	Partitioning # 34	53.3 cycles	264 cycles
Partitioning # 02	53.3 cycles	264 cycles	Partitioning # 35	52.3 cycles	259 cycles
Partitioning # 03	52.3 cycles	259 cycles	Partitioning # 36	53.3 cycles	264 cycles
Partitioning # 04	53.3 cycles	264 cycles	Partitioning # 37	52.3 cycles	259 cycles
Partitioning # 05	52.3 cycles	259 cycles	Partitioning # 38	53.3 cycles	264 cycles
Partitioning # 06	53.3 cycles	264 cycles	Partitioning # 39	52.3 cycles	259 cycles
Partitioning # 07	52.3 cycles	259 cycles	Partitioning # 40	53.3 cycles	264 cycles
Partitioning # 08	53.3 cycles	264 cycles	Partitioning # 41	52.3 cycles	259 cycles
Partitioning # 09	52.3 cycles	259 cycles	Partitioning # 42	53.3 cycles	264 cycles
Partitioning # 10	53.3 cycles	264 cycles	Partitioning # 43	52.3 cycles	259 cycles
Partitioning # 11	52.3 cycles	259 cycles	Partitioning # 44	53.3 cycles	264 cycles
Partitioning # 12	53.3 cycles	264 cycles	Partitioning # 45	52.3 cycles	259 cycles
Partitioning # 13	52.3 cycles	259 cycles	Partitioning # 46	53.3 cycles	264 cycles
Partitioning # 14	53.3 cycles	264 cycles	Partitioning # 47	52.3 cycles	259 cycles
Partitioning # 15	52.3 cycles	259 cycles	Partitioning # 48	53.3 cycles	264 cycles
Partitioning # 16	53.3 cycles	264 cycles	Partitioning # 49	52.3 cycles	259 cycles
Partitioning # 17	52.3 cycles	259 cycles	Partitioning # 50	53.3 cycles	264 cycles
Partitioning # 18	53.3 cycles	264 cycles	Partitioning # 51	52.3 cycles	259 cycles
Partitioning # 19	52.3 cycles	259 cycles	Partitioning # 52	53.3 cycles	264 cycles
Partitioning # 20	53.3 cycles	264 cycles	Partitioning # 53	52.3 cycles	259 cycles
Partitioning # 21	52.3 cycles	259 cycles	Partitioning # 54	53.3 cycles	264 cycles
Partitioning # 22	53.3 cycles	264 cycles	Partitioning # 55	52.3 cycles	259 cycles
Partitioning # 23	52.3 cycles	259 cycles	Partitioning # 56	53.3 cycles	264 cycles
Partitioning # 24	53.3 cycles	264 cycles	Partitioning # 57	52.3 cycles	259 cycles
Partitioning # 25	52.3 cycles	259 cycles	Partitioning # 58	53.3 cycles	264 cycles
Partitioning # 26	53.3 cycles	264 cycles	Partitioning # 59	52.3 cycles	259 cycles
Partitioning # 27	52.3 cycles	259 cycles	Partitioning # 60	53.3 cycles	264 cycles
Partitioning # 28	53.3 cycles	264 cycles	Partitioning # 61	52.3 cycles	259 cycles
Partitioning # 29	52.3 cycles	259 cycles	Partitioning # 62	53.3 cycles	264 cycles
Partitioning # 30	53.3 cycles	264 cycles	Partitioning # 63	52.3 cycles	259 cycles
Partitioning # 31	52.3 cycles	259 cycles	Partitioning # 64	53.3 cycles	264 cycles
Partitioning # 32	53.3 cycles	264 cycles			

Table 5.2: RECURSIVE_AQ: Analysis vs. Simulation

The analysis program estimated the execution time of `RECURSIVE_AQ` with an error between 78.2% and 79.8%. In large part, recursion introduces this large error. Instead of making an accurate estimate for the cost of the two `RECURSIVE_AQ CALL` instructions, we chose the fixed cost of 1 cycle per call. Supplying an accurate estimate for the recursive call would presume information not available to a compiler or a program designer.

The difference between simulated execution time of particular partitionings was not significant (1.5% maximum). However, the speedup of the worst-case partitioning over no partitioning was approximately 18.5%.

5.3.3 Discussion of Example Programs

A cursory glance at Tables 5.1 and 5.2 makes it obvious that our modeling system does not handle recursion well. The analysis program estimated the execution time of `RECURSIVE_AQ` with an error of about 80%, whereas the analysis program estimated execution time of `INTEGRATE` with an error between 20.8% and 40.6%.

However, that did not invalidate the usefulness of the model in our partitioning problem. In both `RECURSIVE_AQ` and `INTEGRATE`, the relative ordering of the simulation and analyzer times were the same. The analyzer would have chosen a good partition in both cases.

We are concerned that the analyzer program is expensive to run. Analysis of `INTEGRATE` took approximately 10 minutes of CPU time on an unloaded

HP 9000/350, while simulation took 30 seconds. Analysis of RECURSIVE_AQ took 14 hours, while simulation took 1 hour. This discrepancy may be explained partially by the fact that the analyzer is written in T, and the simulator is written in Modula-2.

Our experiences lead us to believe that Markov analysis is useful as a partitioning heuristic when the data flow program is relatively small and the expected execution time is large.

At least for these examples, selecting a *particular* partitioning makes only a small difference in the execution time. For INTEGRATE and RECURSIVE_AQ, the best-case partitioning execution time beats the worst-case partitioning by 9.4% and 1.8% respectively.

Even the worst-case partitioning beats no partitioning by a substantial margin, 33.3% and 18.5% for INTEGRATE and RECURSIVE_AQ. The best-case partition beats no partitioning by 39.6% and 20.0%.

We note that the number of partitionings generated by Algorithm PSB grows exponentially with the number of multiple-successor nodes. The decomposition methods of [Kape87] [Kape88] can address this problem to some extent. However, our experience seems to indicate that (since worst-case and best case times do not differ substantially) using a cheaper heuristic to reduce the number of partitions generated by Algorithm PSB would provide an efficient data flow optimization scheme.

CHAPTER 6

Conclusion

We have shown that optimally partitioning a data flow program is intractable, but that approximate solutions can be obtained if the system is purely stochastic, and transition probabilities are known and fixed.

We described a model for describing data flow program behavior stochastically, the “Probabilistic Data Flow Graph.” Probabilistic data flow graphs can be transformed into discrete-time, discrete-space Markov chains.

However, Markov chains created from these PDFGs may contain proper closed subsets, which destroy the predictive power of the model. We provide a linear-time algorithm for deleting these closed subsets from Markov chains.

We described a linear-time, non-deterministic algorithm for partitioning a data flow graph into sequential blocks. Several partitionings may exist, and selecting the fastest is an intractable problem. Our stochastic model provides an approximate value for the execution time of a partitioning, and allows us to compare different partitionings. From this comparison, we can choose the partitioning which results in the lowest estimated execution time.

We described our implementation of an analysis program and a simulation program. We showed two example programs, INTEGRATE and RECUR-

SIVE_AQ, and gave the analytical and simulated execution times of both.

We discovered that, in these cases, the modeling method chose the correct partitioning. Its inaccuracy in estimating execution time, however, could cause a non-optimal partitioning to be selected in other programs.

On the other hand, we know that an optimal solution is incomputable by Theorem 7, and with the example programs we gave, our modeling system appears to be a reasonable heuristic for selecting a good partitioning.

We also discovered that, in our examples, selecting a *particular* partitioning made little difference. We feel that a cheap heuristic might be more appropriate for this partitioning algorithm, since the performance of the worst-case and best-case partitionings differed by only a small margin.

6.1 Work Remaining

The work developed in this thesis gives rise to many unanswered questions, which could provide fuel for further research. Among the unaddressed issues remaining are:

1. We suspect that selecting the optimal partitioning, even with fixed known transition probabilities, is an *NP*-complete problem. Algorithm PSB and Theorem 6 provide proof that selecting the optimal partitioning is in *NP*. Several *NP*-complete problems, including the knapsack problem, linear programming, etc., are known to be *NP*-complete, but they belong to a

class of problems where either

(a) in most cases, tractable solutions exist, or

(b) in most cases, obtaining a close approximate solution is tractable.

2. A method for decomposing a discrete-time system into an simpler, higher-level problem is needed.

3. Using a cheaper heuristic seems to be indicated by our results. Other heuristics should be investigated and compared to the Markov heuristic we developed.

APPENDIX A

Notation

- Q** The set of rational numbers.
- R** The set of real numbers.
- Z** The set of integers.
- Q⁺** The set of positive rationals ($0 \notin \mathbf{Q}^+$).
- R⁺** The set of positive reals ($0 \notin \mathbf{R}^+$).
- Z⁺** The set of positive integers ($0 \notin \mathbf{Z}^+$).
- Q⁰⁺** The set of non-negative rationals ($\mathbf{Q}^+ \cup \{0\}$).
- R⁰⁺** The set of non-negative reals ($\mathbf{R}^+ \cup \{0\}$).
- Z⁰⁺** The set of non-negative integers ($\mathbf{Z}^+ \cup \{0\}$).
- $f[A]$** The image of function f under the set A .
- 2^A** The power set of set A .
- { }** Set constructors.
- $\exists!$** “There exists a unique element ...”

APPENDIX B

RECURSIVE_AQ: Data Flow Program Listing

```
(edge =a 1 0 0.0)
(edge =b 1 0 -8.0)
(edge =c 1 0 10.0)
(edge =d 1 0 122.0)
(edge =e 1 -1)
(edge =f 1 -1)
(edge =g 1 -1)
(edge =h 1 -1)
(edge =i 1 -1)
(edge =k 1 -1)
(edge =l 1 -1)
(edge =m 1 -1)
(edge =n 1 -1)
(edge =o 1 -1)
(edge =p 1 -1)
(edge =q 1 -1)
(edge =r 1 -1)
(edge =s 1 -1)
(edge =t 1 -1)
(edge =u 1 -1)
(edge =v 1 -1)
(edge =w 1 -1)
(edge =x 1 -1)
(edge =y 1 -1)
(edge =z 1 -1)
(edge =aa 1 -1)
(edge =ab 1 -1)
(edge =ac 1 -1)
(edge =ad 1 -1)
(edge =ae 1 -1)
(edge =af 1 -1)
(edge =ag 1 -1)
(edge =ah 1 -1)
(edge =ai 1 -1)
(edge =aj 1 -1)
(edge =ak 1 -1)
(edge =al 1 -1)
(edge =am 1 -1)
(edge =an 1 -1)
(edge =ao 1 -1)
(edge =ap 1 -1)
(edge =aq 1 -1)
(edge =ar 1 -1)
(edge =as 1 -1)
(edge =at 1 -1)
(edge =au 1 -1)
(edge =av 1 -1)
(edge =aw 1 -1)
(edge =ax 1 -1)
(edge =ay 1 -1)
(edge =az 1 -1)
(edge =ba 1 -1)
(edge =bb 1 -1)
(edge =bc 1 -1)
(edge =bd 1 -1)
(edge =be 1 -1)
(edge =bf 1 -1)
(edge =bg 1 -1)
(edge =bh 1 -1)
(edge =bi 1 -1)
```

```

(edge =bj 1 -1)
(edge =bm 1 -1)
(edge =bq 1 -1)
(edge =br 1 -1)
(edge =bs 1 -1)
(edge =bt 1 -1)
(edge =bu 1 -1)
(edge =bv 1 -1)
(edge =bw 1 -1)
(edge =cb 1 -1)
(edge =cc 1 -1)
(edge =cd 1 -1)
(edge =ce 1 -1)
(edge =cj 1 -1)
(edge =ck 1 -1)
(edge =cl 1 -1)
(edge =cm 1 -1)
(edge =cn 1 -1)
(edge =co 1 -1)
(edge =cp 1 -1)
(edge =cq 1 -1)
(edge =cr 1 -1)
(edge =cs 1 -1)
(edge =ct 1 -1)
(edge =cu 1 -1)
(edge =cw 1 -1)
(edge =cx 1 -1)
(edge =cy 1 -1)
(edge =cz 1 -1)
(edge =db 1 -1)
(edge =dc 1 -1)
(edge =dd 1 -1)
(edge =de 1 -1)
(edge =df 1 -1)
(edge =dg 1 -1)
(edge =dh 1 -1)
(edge =di 1 -1)
(edge =dj 1 -1)
(edge =dk 1 -1)
(edge =dl 1 -1)
(edge =dm 1 -1)
(edge =dn 1 -1)
(edge =do 1 -1)
(edge =dp 1 -1)
(vertex RecursiveAQ NOP 1 -1 () ((1 =a =b =c =d)))
(vertex *1 DUP 1 -1 ((1 =a)) ((1 =dl =e)))
(vertex *2 DUP 1 -1 ((1 =b)) ((1 =f =g)))
(vertex *3 DUP 1 -1 ((1 =c)) ((1 =h =i)))
(vertex *4 DUP 1 -1 ((1 =k)) ((1 =p =m)))
(vertex *5 DUP 1 -1 ((1 =d)) ((1 =k =l)))
(vertex *6 DUP 1 -1 ((1 =e)) ((1 =n =o)))
(vertex *7 PLUS 1 -1 ((1 =dl =h)) ((1 =q)))
(vertex *8 DUP 1 -1 ((1 =u)) ((1 =r =s)))
(vertex *9 DUP 1 -1 ((1 =i)) ((1 =t =u)))
(vertex *10 PLUS 1 -1 ((1 =g =l)) ((1 =v)))
(edge =c0 0 -1)
(constantvertex *C0 0.5 ((1 =c0)))
(vertex *11 TIMES 1 -1 ((1 =q =c0)) ((1 =x)))
(vertex *12 DUP 1 -1 ((1 =x)) ((1 =ac =ad)))
(vertex *13 DUP 1 -1 ((1 =f)) ((1 =w =y)))
(vertex *14 DUP 1 -1 ((1 =t)) ((1 =z =aa)))
(vertex *15 MINUS 1 -1 ((1 =u =o)) ((1 =ab)))
(edge =c13 0 -1)
(constantvertex *C13 "Evaluate_Function" ((1 =c13)))
(vertex *16 CALL 9 -1 ((1 =c13 =ac)) ((1 =ag)))
(vertex *17 DUP 1 -1 ((1 =ad)) ((1 =ae =ah)))
(vertex *18 DUP 1 -1 ((1 =ae)) ((1 =af =ai)))
(vertex *19 DUP 1 -1 ((1 =al)) ((1 =an =ao)))
(vertex *20 DUP 1 -1 ((1 =ab)) ((1 =aj =ak)))
(vertex *21 DUP 1 -1 ((1 =ag)) ((1 =al =am)))
(vertex *22 PLUS 1 -1 ((1 =p =am)) ((1 =ap)))
(vertex *23 MINUS 1 -1 ((1 =z =ah)) ((1 =aq)))
(vertex *24 MINUS 1 -1 ((1 =ai =r)) ((1 =ar)))
(vertex *25 PLUS 1 -1 ((1 =an =y)) ((1 =as)))
(vertex *26 TIMES 1 -1 ((1 =aj =v)) ((1 =at)))
(vertex *27 TIMES 1 -1 ((1 =ap =aq)) ((1 =au)))

```

```

(vertex *28 TIMES 1 -1 ((1 =ax =as)) ((1 =av)))
(edge =c1 0 -1)
(constantvertex *C1 0.5 ((1 =c1)))
(vertex *29 TIMES 1 -1 ((1 =at =c1)) ((1 =av)))
(edge =c7 0 -1)
(constantvertex *C7 0.5 ((1 =c7)))
(vertex *30 TIMES 1 -1 ((1 =au =c7)) ((1 =ax)))
(vertex *31 PLUS 1 -1 ((1 =ax =ay)) ((1 =az)))
(edge =c2 0 -1)
(constantvertex *C2 0.5 ((1 =c2)))
(vertex *32 TIMES 1 -1 ((1 =av =c2)) ((1 =ay)))
(vertex *33 DUP 1 -1 ((1 =aw)) ((1 =ba =bb)))
(vertex *34 DUP 1 -1 ((1 =az)) ((1 =bc =bd)))
(vertex *35 ISERROR 1 -1 ((1 =bf)) ((1 =bg)))
(vertex *36 DUP 1 -1 ((1 =bd)) ((1 =bf =be)))
(edge =c12 0 -1)
(constantvertex *C12 "Stop_Condition" ((1 =c12)))
(vertex *37 CALL 7 -1 ((1 =c12 =ba =be =ak)) ((1 =bj)))
(vertex *38 OR 1 -1 ((1 =bg =bh)) ((1 =bq)))
(vertex *39 ISERROR 1 -1 ((1 =bi)) ((1 =bh)))
(vertex *40 DUP 1 -1 ((1 =bj)) ((1 =bi =bs)))
(edge =c3 0 -1)
(constantvertex *C3 TRUE ((1 =c3)))
(vertex *B1 BRR 1 -1 ((1 =br =bb =c3)) ((9999 =bt =bv) (1 =bu =bw)))
(edge =c4 0 -1)
(constantvertex *C4 FALSE ((1 =c4)))
(vertex *B2 BRR 1 -1 ((1 =ce =bc =c4)) ((1 =cw =cy) (1 =cx =cz)))
(vertex *B3 BRR 1 -1 ((1 =bm =a =w =aa =m =af =ao))
((1 =ck =cm =co =cq =cs =cu) (1 =cj =cl =cn =cp =cr =ct)))
(vertex *47 DUP 1 -1 ((1 =bq)) ((1 =br =cb)))
(vertex *48 AND 1 -1 ((1 =ba =cc)) ((1 =cd)))
(vertex *57 NOT 1 -1 ((1 =cb)) ((1 =cc)))
(vertex *58 DUP 1 -1 ((1 =cd)) ((1 =bm =ce)))
(edge =c10 0 -1)
(constantvertex *C10 "RecursiveAQ" ((1 =c10)))
(vertex *62 CALL 1 -1 ((1 =c10 =ck =cm =db =dd)) ((1 =df =dg)))
(edge =c11 0 -1)
(constantvertex *C11 "RecursiveAQ" ((1 =c11)))
(vertex *63 CALL 1 -1 ((1 =c11 =dc =de =co =cq)) ((1 =dh =di)))
(vertex *64 PLUS 1 -1 ((1 =df =dh)) ((1 =dj)))
(vertex *65 OR 1 -1 ((1 =dg =di)) ((1 =dk)))
(vertex *66 MERG 0 -1 ((1 =cx) (1 =bu)) ((1 =dm)))
(vertex *67 MERG 0 -1 ((1 =cz) (1 =bv)) ((1 =dn)))
(vertex *68 MERG 0 -1 ((1 =dm) (1 =dj)) ((1 =do)))
(vertex *69 MERG 0 -1 ((1 =dn) (1 =dk)) ((1 =dp)))
(vertex *70 DUP 1 -1 ((1 =cu)) ((1 =dd =de)))
(vertex *71 DUP 1 -1 ((1 =cs)) ((1 =db =dc)))
(vertex *S1 STUB 0 -1 ((1 =cj)) ())
(vertex *S2 STUB 0 -1 ((1 =cl)) ())
(vertex *S3 STUB 0 -1 ((1 =cn)) ())
(vertex *S4 STUB 0 -1 ((1 =cp)) ())
(vertex *S5 STUB 0 -1 ((1 =cr)) ())
(vertex *S6 STUB 0 -1 ((1 =ct)) ())
(vertex *S7 STUB 0 -1 ((1 =cw)) ())
(vertex *S8 STUB 0 -1 ((1 =cy)) ())
(vertex *S9 STUB 0 -1 ((1 =bt)) ())
(vertex *S10 STUB 0 -1 ((1 =bv)) ())
(finalvertex *F ((1 =do =dp)))
end

```

APPENDIX C

RECURSIVE_AQ: Identified Partitions

Partition 1

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3)
(*S2) (*S1) (*70) (*B3 *71) (*63) (*62) (*65) (*64) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CS =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 2

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3)
(*S2) (*S1) (*70) (*B3 *71) (*63) (*62) (*65) (*64) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CS =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 3

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3)
(*S2) (*S1) (*70) (*B3 *71) (*63) (*62) (*65) (*64) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CS =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 4

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3)
(*S2) (*S1) (*70) (*B3 *71) (*63) (*62) (*65) (*64) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CS =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 5

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3)
(*S2) (*S1) (*71) (*B3 *70) (*63) (*62) (*65) (*64) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CU =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 6

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3)
(*S2) (*S1) (*71) (*B3 *70) (*63) (*62) (*65) (*64) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CU =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 7

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3)
(*S2) (*S1) (*71) (*B3 *70) (*63) (*62) (*65) (*64) (*S10) (*B1 *S9)

(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:
 =K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CU =BT =T =I =AB
 =AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 8

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*71) (*B3 *70) (*63) (*62) (*65) (*64) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:
 =K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CU =BT =T =I =AB
 =AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 9

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S2) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S1) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:
 =K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CJ =BV =T =I =AB
 =AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 10

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S2) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S1) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:
 =K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CJ =BV =T =I =AB
 =AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 11

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S2) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S1) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:
 =K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CJ =BT =T =I =AB
 =AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 12

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S2) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S1) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:
 =K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CJ =BT =T =I =AB
 =AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 13

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S2) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:
 =K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CL =BV =T =I =AB
 =AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 14

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S2) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:
 =K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CL =BV =T =I =AB
 =AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 15

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40

*39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S2) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CL =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 16

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S3) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S2) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CL =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 17

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S3) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CH =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 18

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S3) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CH =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 19

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S3) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CH =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 20

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S4) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S3) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CH =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 21

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S4) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CP =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 22

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S6) (*S5) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S4) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CP =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 23

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S6) (*S5) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S4) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CP =BT =T =I =AB =AE =AV =AL =AG =AC =X =Q =F =W =E

Partition 24

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S6) (*S5) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S4) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CP =BT =T =I =AB =AL =AG =AV =AE =AD =X =Q =F =W =E

Partition 25

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S6) (*S4) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S5) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CR =BV =T =I =AB =AE =AV =AL =AG =AC =X =Q =F =W =E

Partition 26

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S6) (*S4) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S5) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CR =BV =T =I =AB =AL =AG =AV =AE =AD =X =Q =F =W =E

Partition 27

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S6) (*S4) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S5) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CR =BT =T =I =AB =AE =AV =AL =AG =AC =X =Q =F =W =E

Partition 28

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S6) (*S4) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S5) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CR =BT =T =I =AB =AL =AG =AV =AE =AD =X =Q =F =W =E

Partition 29

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S5) (*S4) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S6) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CT =BV =T =I =AB =AE =AV =AL =AG =AC =X =Q =F =W =E

Partition 30

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S5) (*S4) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S6) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CT =BV =T =I =AB =AL =AG =AV =AE =AD =X =Q =F =W =E

Partition 31

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*58) (*B2 *S7) (*S5) (*S4) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S6) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=I =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CT =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 32

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S8) (*B2 *S7) (*S5) (*S4) (*S3) (*S2) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S6) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CW =CT =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 33

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*70) (*B3 *71) (*63) (*62) (*65) (*64) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CS =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 34

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*70) (*B3 *71) (*63) (*62) (*65) (*64) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CS =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 35

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*70) (*B3 *71) (*63) (*62) (*65) (*64) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CS =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 36

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*70) (*B3 *71) (*63) (*62) (*65) (*64) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CS =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 37

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*71) (*B3 *70) (*63) (*62) (*65) (*64) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CU =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 38

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*71) (*B3 *70) (*63) (*62) (*65) (*64) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CU =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 39

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*71) (*B3 *70) (*63) (*62) (*65) (*64) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CU =BT =T =I =AB =AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 40

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*S1) (*71) (*B3 *70) (*63) (*62) (*65) (*64) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CU =BT =T =I =AB =AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 41

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S1) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CJ =BV =T =I =AB =AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 42

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S1) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CJ =BV =T =I =AB =AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 43

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S1) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CJ =BT =T =I =AB =AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 44

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S2) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S1) (*S10) (*B1 *S9) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CJ =BT =T =I =AB =AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 45

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S2) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16 *21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CL =BV =T =I =AB =AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 46

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40 *39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3) (*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S2) (*S9) (*B1 *S10) (*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17 *18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CL =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 47

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S2) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CL =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 48

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S3)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S2) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CL =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 49

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S3) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CN =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 50

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S3) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CN =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 51

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S3) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CN =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 52

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S4) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S3) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CN =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 53

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S4) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CP =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 54

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S4) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CP =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 55

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S4) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CP =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 56

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S5) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S4) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CP =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 57

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S4) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S5) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CR =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 58

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S4) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S5) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CR =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 59

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S4) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S5) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CR =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 60

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S6) (*S4) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S5) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CR =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 61

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S5) (*S4) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S6) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CT =BV =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 62

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S5) (*S4) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S6) (*S9) (*B1 *S10)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17

*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CT =BV =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

Partition 63

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S5) (*S4) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S6) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*17 *18) (*24) (*28 *32) (*23) (*7 *11 *12 *16
*21 *19) (*25) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CT =BT =T =I =AB
=AE =AV =AL =AG =AC =X =Q =F =H =E

Partition 64

(*5 *4) (*22) (*27 *30) (*31 *34 *36 *35) (*10) (*26 *29 *33) (*37 *40
*39) (*38 *47 *57) (*48 *58) (*S7) (*B2 *S8) (*S5) (*S4) (*S3) (*S2)
(*S1) (*70) (*71) (*63) (*62) (*65) (*64) (*B3 *S6) (*S10) (*B1 *S9)
(*3 *9 *14) (*15 *20) (*16 *21 *19) (*25) (*28 *32) (*7 *11 *12 *17
*18) (*24) (*23) (*2 *13) (*1 *6 *8) (*69) (*F) (*68) (*67) (*66)

Zeroed edges:

=K =AU =BF =BD =AZ =AW =AT =BI =BJ =CB =BQ =CD =CY =CT =BT =T =I =AB
=AL =AG =AV =AE =AD =X =Q =F =H =E

APPENDIX D

The Simulator

We wrote a tagged-token data flow machine simulator in Modula-2, using the SIMON simulation package. Our machine corresponds roughly to the Manchester machine [Gurd85]. It differs from the Manchester machine in the following ways:

1. Communication plus matching time is given a fixed value for each edge.
On a real machine, this time will vary depending on the machine load, and the size of the data tokens.
2. The matching unit can handle operations which require more than two input operands. The Manchester machine matching unit imposes a maximum on the number of input operands: two.
3. The matching unit handles arbitrary multiple “enabling groups” according to the PDFG specification given on page 23. The Manchester machine allows only one enabling group per instruction.
4. Our system does not allow two edges to share the same sink vertex and sink position. We provide an equivalent construct in the MERG instruction.
5. Our system provides no special structure store, and has no structure operations.

6. An infinite number of processors are provided. There is no processor contention.

D.1 Instructions Provided

The simulator has an easily extensible instruction set. We implemented only those instructions necessary to simulate our example data flow programs. The instructions provided include:

MERG This instruction allows any number of input edges, each in its own enabling set. There is only one output edge. An input edge will be randomly selected from those which contain tokens, the token absorbed and reproduced as an output token. This operation executes in zero time.

Its function corresponds exactly to the case in the Manchester machine where two input edges enter the same vertex at the same position. We provide an explicit instruction to do this, simply to keep the input format of our analysis program and our simulator the same.

DUP The DUP instruction has one input edge. It duplicates an input token, and outputs it on two output edges.

SUBR The SUBR instruction takes no input edges and allows any number of output edges. It functions simply as a label for a subroutine. The output edges correspond to the parameters of the subroutine. CALL instructions which refer to this subroutine must have the same number of input edges

(not including the subroutine name) as the corresponding SUBR instruction has output edges.

CALL The first input edge to a CALL instruction should be a string constant naming the subroutine to be called. The remaining input edges correspond to parameters. There should be the same number of input parameter edges as there are output edges to the corresponding SUBR instruction.

The CALL instruction allows a varying number of output edges. The number of output edges for a particular CALL instruction should equal the number of input edges to the appropriate RET instruction in the called subroutine.

The CALL instruction first obtains a new unique “invocation ID” for the subroutine being called. It saves the invocation ID of its input tokens, the label of the CALL instruction, and the new invocation ID in an “invocation memory.” It changes the invocation IDs of its input tokens to the new invocation ID. The CALL instruction then send the tokens out on the output edges of the named SUBR instruction.

RET This instruction takes any number of input edges and produces no output edges. When it receives input tokens, it looks up their invocation ID in the invocation memory. It sets the invocation IDs of its input tokens to the *old* invocation ID. It then produces output tokens on the appropriate CALL instruction.

PLUS, MINUS, TIMES These instructions all take two input operands and produce one output token. Their functions are obvious. If inputs are integers, they produce an integer output. If inputs are real, they produce a real output.

ABS This produces the absolute value of its single input token. If the input is an integer, it produces an integer output. If the input is real, it produces a real output.

OR, AND, NOT These instructions perform corresponding logical operations on their boolean inputs.

BRR This branch instruction takes a number of inputs greater than 1. The first input edge must carry a boolean value. If there are n input edges, the BRR instruction must have $2(n - 1)$ output edges. If the first input edge carries a false value, the tokens on input edges 2 through n will be copied to output edges 1 through $(n - 1)$. If the first input edge carries a true value, the tokens on input edges 2 through n will be copied to output edges n through $2(n - 1)$.

CGR This “compare greater” instruction compares its two input tokens. If the left is greater than the right, it outputs a true value. Otherwise it outputs a false value.

ADL This “add to iteration level” instruction increments the iteration level number of the left input token by the integer value of the right input token.

SIL This “set iteration level” instruction sets the iteration level number of the left input token to the integer value of the right input token.

STUB This instruction merely absorbs an input value, and produces nothing.

ISERROR This instruction produces a true output value if the input token is an error value. It produces a false output value if the input token has any other value.

Bibliography

- [Arvi87] Arvind and R.S. Nikhil, *Executing a Program on the MIT Tagged-Token Dataflow Architecture*, Technical Report CSD Memo 271, Massachusetts Institute of Technology, Cambridge, Massachusetts (1987).
- [Back78] J. Backus, Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM*, 21(8):613-641 (August 1978).
- [Burt81] F.W. Burton and M.R. Sleep, Executing Functional Programs on a Virtual Tree of Processors, pp. 187-194, in *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architectures* (1981).
- [Camp87] M.L. Campbell, *Data Flow Graphs as a Model of Parallel Complexity*, PhD dissertation, UCLA Computer Science Department, Los Angeles, California (1987).
- [Chan81] T.L. Chang and P.D. Fisher, A Block-Driven Data-Flow Processor, pp. 151-155, in *Proceedings of the 1981 International Conference on Parallel Processing*, Columbus, Ohio (1981).
- [Chan84] P.K. Chan, *A Dataflow Multiprocessor: Programming, Simulation and Performance Prediction*, Technical Report CSD-840044, UCLA Computer Science Department, Los Angeles, California (November 1984).
- [Denn79] J.B. Dennis, The Varieties of Data Flow Computers, pp. 430-439, in *Proceedings of the IEEE Conference on Distributed Systems* (1979).
- [Denn80] J.B. Dennis, Data Flow Supercomputers, *Computer*, 13(11):48-56 (November 1980).
- [Erce84] M.D. Ercegovic, P.K. Chan, and T.M. Ravi, A Data Flow Multi-microprocessor Architecture for High-Speed Simulation of Continuous Systems, in *Proceedings of the International Workshop on High-Level Architecture*, Los Angeles, California (1984).
- [Even79] S. Even, *Graph Algorithms*, Computer Science Press, Inc., Rockville, Maryland (1979).

- [Gaud82] J.L. Gaudiot, *On Program Decomposition and Partitioning in Dataflow Systems*, Technical Report CSD 821212, UCLA Computer Science Department (December 1982).
- [Gaud84] J.L. Gaudiot and M.D. Ercegovac, Performance Analysis of a Data-Flow Computer with Variable Resolution Actors, in *IEEE Proceedings of the International Conference on Distributed Computing Systems* (1984).
- [Gaud85] J.L. Gaudiot, R.W. Vedder, G.K. Tucker, D. Finn, and M.L. Campbell, A Distributed VLSI Architecture for Efficient Signal and Data Processing, *IEEE Transactions on Computers*, C-34(12):1072-1087 (December 1985).
- [Gost80] K.P. Gostelow and R.E. Thomas, Performance of a Simulated Data-Flow Computer, *IEEE Transactions on Computers*, C-10(10):905-919 (October 1980).
- [Grna80] A. Grnarov, L. Kleinrock, and M. Gerla, A Highly Reliable, Distributed Loop Network Architecture, pp. 319-324, in *Proceedings of the Tenth International Symposium on Fault-Tolerant Computing*, IEEE (1980).
- [Gurd78] J.R. Gurd, I. Watson, and J.R.W. Glauert, *A Multilayered Data Flow Computer Architecture*, Technical Report, Department of Computer Science, University of Manchester, Manchester, England (July 1978).
- [Gurd85] J.R. Gurd, C.C. Kirkham, and I. Watson, The Manchester Prototype Dataflow Computer, *Communications of the ACM*, 28(1):34-52 (January 1985).
- [Harr84] W.J. Harrod and R.J. Plemmons, Comparisons of Some Direct Methods for Computing Stationary Distributions of Markov Chains, *SIAM Journal on Scientific and Statistical Computing*, 5:453-469 (1984).
- [Heym87] D. Heyman, Further Comparisons of Direct Methods for Computing Stationary Distributions of Markov Chains, *SIAM Journal on Algebraic and Discrete Systems*, 8(2):226-232 (April 1987).
- [Huda85] P. Hudak and B. Goldberg, Distributed Execution of Functional Programs Using Serial Combinators, *IEEE Transactions on Computers*, C-34(10):881-891 (October 1985).
- [Kape86] A. Kapelnikov, *Analytic Modeling Methodology for Evaluating the Performance of Distributed, Multiple-Computer Systems*, Technical Report unpublished, UCLA Computer Science Department, Los Angeles, California (December 1986).

- [Kape87] A. Kapelnikov, R.R. Muntz, and M.D. Ercegovac, A Methodology for the Performance Evaluation of Distributed Computations, in *Proceedings of the IFIP Conference on Distributed Processing* (October 1987).
- [Kape88] A. Kapelnikov, R.R. Muntz, and M.D. Ercegovac, A Modelling Methodology for the Analysis of Concurrent Systems and Computations, to appear in *Journal of Parallel and Distributed Computing* (1988).
- [Karp66] R.M. Karp and R.E. Miller, Properties of a Model for Parallel Computation: Determinacy, Termination, Queuing, *SIAM Journal on Applied Mathematics*, 14(6):1390–1411 (November 1966).
- [Karp69] R.M. Karp, Parallel Program Schemata, *Journal of Computer and System Sciences*, 3(4):147–195 (May 1969).
- [Klei75] L. Kleinrock, *Queueing Systems, Volume 1: Theory*, John Wiley & Sons, New York (1975).
- [Lave83] S.S. Lavenberg, editor, *Computer Performance Modeling Handbook*, Academic Press, New York (1983).
- [Leco81] M. Lecouffe, Architecture of a Multiprocessor Using Data Flow at a Program Block Level, pp. 160–161, in *Proceedings of the International Conference on Parallel Processing*, Columbus, Ohio (1981).
- [Mago80] G.A. Mago, A Cellular Computer Architecture for Functional Programming, pp. 179–187, in *Proceedings of IEEE COMPCON* (1980).
- [McGr85] J. McGraw and S. Skedzielewski, *SISAL: Language Reference Manual, version 1.2*, Technical Report M-146, Lawrence Livermore National Laboratory, Livermore, California (1985).
- [Pete77] J.L. Peterson, Petri Nets, *ACM Computing Surveys*, 9(3):223–252 (September 1977).
- [Slad87] S. Slade, *The T Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1987).
- [Thom78] R.E. Thomas, *Performance Analysis of Two Classes of Data-Flow Computer Systems*, Master's thesis, University of California, Irvine (1978), Department of Information and Computer Science Technical Report 120.
- [Thom85] A. Thomasian and P. Bay, Performance Analysis of Task Systems Using a Queueing Network Model, in *International Workshop on Timed Petri Nets*, Torino, Italy (1985).

- [Thom86] A. Thomasian and P. Bay, Analytic Queueing Network Models for Parallel Processing of Task Systems, *IEEE Transactions on Computers*, C-35(12):1045–1054 (December 1986).
- [Turi36] A.M. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, pp. 241–265, in *Proceedings of the London Mathematical Society, Series II* (December 1936).
- [Vegd84] S.R. Vegdahl, A Survey of Proposed Architectures for the Execution of Functional Languages, *IEEE Transactions on Computers*, C-33(12):1050–1070 (December 1984).
- [Wats82] I. Watson and J.R. Gurd, A Practical Dataflow Computer, *Computer*, 15(2):51–57 (February 1982).