MIT/LCS/TR-240

SAFETY AND OPTIMIZATION TRANSFORMATIONS FOR DATA FLOW PROGRAMS

Lynn Barbara Montz

Safety and Optimization Transformations for Data Flow Programs

by

Lynn Barbara Montz

January, 1980

Copyright 1980 Massachusetts Institute of Technology

This research was supported in part by the National Science Foundation under research grant MCS75-04060 A01 and in part by the Lawrence Livermore Laboratory of the University of California under contract 8545403.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

Safety and Optimization Transformations for Data Flow Programs

by

Lynn Barbara Montz

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 1980 in partial fulfillment of the requirements for
the Degree of Master of Science

ABSTRACT

The *data flow* concept of computation seeks to achieve high performance by allowing concurrent execution of instructions based on the availability of data. This thesis explores the translation of a subset of the high level language VAL to data flow graphs. The major problem in performing this translation for the target machine, the Dennis-Misunas data flow computer, stems from the restriction that graph execution sequences place at most one value on any given arc at any time. The *data/acknowledge arc pair transformation* is introduced as a means of implementing this required operational behavior. Its effect on data flow graph operation is subsequently explored as it relates to correctness and performance.

Though the arc transformation enables graphs to be executed without the possibility of deadlock, the resulting overhead and the potential loss of some concurrency represent significant costs. Two techniques aimed at minimizing these problems are developed for optimizing transformed graphs. The optimization to *eliminate unneeded acknowledge arcs* analyzes VAL constructs to identify arc pairs which may permit removal of their acknowledge arc. The optimization to *balance token flow* specifies a method of inserting identity operators into a graph for the purpose of pipelining input sets, and thereby increasing graph throughput. Though developed within the context noted, the translation and optimization issues described should prove applicable to other data flow architectures.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering

Keywords: data flow programming, data flow translation, optimization, asynchronous systems, Petri nets.

This empty page was substituted for a blank page in the original document.

ACKNOWLEDGEMENTS

Theodor Herzl said, אם תרצו אין זו אגדה , that is, "if you will it, it is no dream"; and though this thesis represents the completion of a personal endeavor, its accomplishment is owed in large part to several individuals to whom I am greatly indebted.

I wish to thank my thesis advisor, Jack Dennis, for offering me the opportunity to join the Computation Structures Group of the Laboratory for Computer Science, and for his subsequent guidance in the formulation and development of this research. I have gained much from my association with the members of his group, and am grateful for the very positive and warm working atmosphere which they have collectively created.

I would like to thank Clem Leung for providing direction and encouragement during the early stages of my work, and Dean Brock for his helpful editorial comments and suggestions concerning the technical content.

I am especially grateful to Bill Ackerman for countless invaluable discussions of the problems and ideas which arose throughout this research. His patience was truly remarkable, and his encouragement and confidence in my abilities should prove a permanent benefit.

Of course, I must express my special thanks to Chris Terman whose excellent preparation of the graphs in this thesis seems quite minor in comparison to the sincerity and value of his friendship.

Finally, I am most grateful to my parents and family, who have always provided me with support, encouragement, and love.

This empty page was substituted for a blank page in the original document.

TABLE OF CONTENTS

1. CHAPTER ONE 6

1.1 Introduction 6
1.2 Data Flow Graph Operation 7
1.3 Translation of VAL to Data Flow Graphs 9
1.4 Safety Transformations for Data Flow Graphs 12
1.5 Optimizing Transformed Data Flow Graphs 14
1.6 Structure of Thesis 15

2. CHAPTER TWO 16

2.1 The Safety Transformation 16
2.2 The Petri Net - Data Flow Graph Analogy 17
2.2.1 History and Analogy 17
2.2.2 Modelling Data Flow Graphs with Petri Nets 19
2.3 The Data/Acknowledge Arc Pair Transformation 22
2.3.1 Achieving Safe Data Flow Graph Operation 22
2.3.2 Preservation of Liveness 24

3. CHAPTER THREE 35

3.1 Balancing Token Flow 35
3.2 Formulating the Optimization 36
3.2.1 Identifying the Source of Bottleneck 36
3.2.2 Preview of a Solution 37
3.2.3 Analyzing Token Flow to Characterize the Solution 38
3.2.4 Observations 42
3.3 Full vs. Limited Buffering 43
3.3.1 Achieving Limited Buffering 43
3.3.2 Examples of Full vs. Limited Buffering 45
3.3.3 Additional Considerations 56

4. CHAPTER FOUR 57

4.1 Eliminating Unneeded Acknowledge Arcs 57
4.2 Considerations for Acknowledge Arc Removal 58
4.3 Analysis of the Conditional Construct 60
4.4 Analysis of the Iteration Construct 66
4.4.1 Acknowledge Arc Removal 66

4.4.2 Acknowledge Arc Removal in Iterative Programs 72

5. CHAPTER FIVE 79

5.1 Summary 79
5.2 Directions for Future Research 81

6. BIBLIOGRAPHY 83

CHAPTER ONE

1.1 Introduction

The short history of computing as a science is unique in its unparalleled rate of technological growth. In response to this, the demand for greater levels of computing power has risen as rapidly. Anticipating the continuation of this trend, research in the area of parallel computation seeks to achieve high performance by manipulating programs to exploit the parallelism inherent in many problems. Though this has led to the introduction of "do in parallel" constructs within certain languages, the sequential nature of conventional machine programming has proved to be a barrier to the formulation of an adequate and practical approach. The *data flow* concept of computation overcomes this difficulty by allowing the availability of data to determine the execution sequence, rather than a sequential instruction counter. In the data flow model, an operation is executed as soon as its required operands have been computed. The development of this concept has resulted in the proposal of several data flow machine architectures and associated data flow languages. This thesis addresses certain language translation problems which arise in translating the high level data flow language, VAL[2] for the Dennis-Misunas data flow machine[11].

The concept of data flow is best illustrated by *data flow graphs* which explicitly show the data dependencies of operations in a data flow program. The *operators* and *arcs* of data flow graphs are viewed as an abstraction of the instruction cells and operand registers of the data flow machine and as such, provide a model for describing translation problems. The chapter proceeds with a more detailed look at the components and operation of data flow graphs, followed by a brief look at the high level data flow language, VAL and its translation into graph form. The major problem, termed *safety*, which arises in making the translation will be identified and discussed in section 1.4. While resolving the

safety issue is straightforward, the solution introduces a secondary, more subtle set of problems to the graphs. Section 1.5 identifies these along with several optimizations of the initial solution aimed at minimizing such problems, an expanded discussion of which forms a major portion of this thesis. The chapter concludes with a synopsis of the remainder of the thesis.

1.2 Data Flow Graph Operation

The basic components of directed *data flow graphs* are *operators* and *arcs* which join the operators. When an operator *fires*, it absorbs values or *tokens* from its input arcs and produces tokens on its output arcs. There are three operator types, and corresponding rules defining their operation or *firing behavior*. The graph in Figure 1.1 which represents the VAL conditional construct:

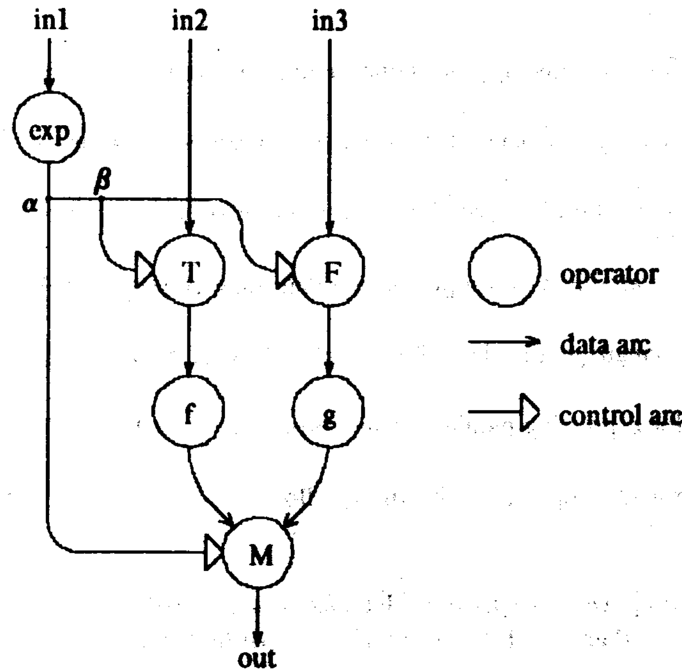
if *exp* **then** *f* **else** *g*

contains instances of each type. The *exp* node is an abbreviation for a VAL expression representing the predicate of the conditional. Thus, it should evaluate to a boolean value.

The most generalized operator type is the functional operator, represented in the figure by nodes *f* and *g*. These operators may perform simple arithmetic operations such as addition or multiplication, or more complex functions such as square root. The firing behavior rule for functional operators specifies that a token be present on each input arc for the operator to fire, at which time all inputs are absorbed, the appropriate function is computed and a result token is produced on each of the operator's output arcs.

The true and false control gates represented in Figure 1.1 by the T, and F nodes form a second operator type. Each of these operators requires a control and a data input to fire, and operates according to the following rule: If the control input matches the gate type, the data input is transmitted to the gate's output arc, otherwise the input data token is absorbed and no output is produced. Thus, a

Figure 1.1. Data flow graph of the VAL expression 'if *exp* then *f* else *g*'



T gate (F gate) will transmit its input data token to its output arc if and only if it receives a true (false) input control token.

The remaining operator type is the M gate, or merge control gate, which has three inputs; a control input, and two data inputs corresponding to true and false control input values. To fire, an M gate requires an input control token and corresponding input data token which is then transmitted to the gate's output arc. A value present on the input data arc not selected, is unaffected by the gate's firing. Appropriately, the M gate merges two paths in the graph. Thus, Figure 1.1 models the conditional construct behavior by allowing an input token to flow through either the T or F gate, (based on the evaluation of *exp*), to the M gate which merges the true and false paths to produce a result token on the graph output port.

1.3 Translation of VAL to Data Flow Graphs

While data flow graphs expose concurrency inherent in a computation by explicit representation of operator dependencies, it is impractical to express programs in this form. Instead, we introduce the high level data flow language VAL, acronym for value-oriented algorithmic language, and a translation algorithm mapping VAL programs into data flow graphs. Developed by Ackerman and Dennis[2] as a source language for data flow graphs, VAL is an applicative language containing constructs well suited for expressing parallelism in a program. A BNF specification of the syntax of a subset of VAL, used in the development of this thesis follows.

exp ::= *id* | *const* | *exp*, *exp* | *oper*(*exp*) | **let** *idlist* = *exp* **in** *exp* |
if *exp* **then** *exp* **else** *exp* | **for** *idlist* = *exp* **do** *iterbody*

iterbody ::= *exp* | **iter** *exp* | **let** *idlist* = *exp* **in** *iterbody* |
if *exp* **then** *iterbody* **else** *iterbody*

id ::= "programming language identifiers"

idlist ::= *id* {, *id* }

const ::= "programming language constants"

oper ::= "programming language operators"

The recursive translation algorithm mapping VAL expressions into their data flow graph implementations, defined by J. D. Brock[3], consists of the functions $\mathbf{T}$ and $\mathbf{T_I}$ which respectively map VAL expressions and iteration bodies into their graph implementations. Both functions produce graphs which have an input port for each free variable in the expression or iteration body being translated. $\mathbf{T}$[*exp*] has an output port for each value returned by the expression; $\mathbf{T_I}$[*iterbody*] has two sets of output ports, **I** and **R**, used respectively to re-iterate or return a set of values, and an output port *iter?* to signal which possibility has occurred. Translations of the conditional and iteration expressions are used

extensively in this thesis, and are shown in Figures 1.2 and 1.3 respectively.

Functioning of the conditional expression in Figure 1.2 should be clear from the discussion of Figure 1.1. Evaluation of $\mathbf{T}[exp_1]$ should produce an input control value for all gates in the graph, allowing tokens to flow through either the T or F gates, enabling computation of the graph represented by $\mathbf{T}[exp_2]$ or $\mathbf{T}[exp_3]$ respectively. The iteration expression of Figure 1.3 is formed by using M gates to merge the values resulting from evaluation of *exp*, with the iteration, I, outputs of $\mathbf{T}_I[iterbody]$. The control input port of each M gate is connected to the *iter?* output of $\mathbf{T}_I[iterbody]$, initialized with a false token to ensure that selection of the first set of data values is from $\mathbf{T}[exp]$. A set of data values will be iterated as long as successive *iter?* outputs are true and will be returned at the first instance of a false *iter?* output, which reinitializes the M gates. A more detailed explanation of the application of the translation algorithm to the conditional and iteration expressions, as well as to the remaining expressions specified in the VAL subset defined above, can be found in [3].

Figure 1.2. $\mathbf{T}[$**if** exp_1 **then** exp_2 **else** exp_3 **end**$]$

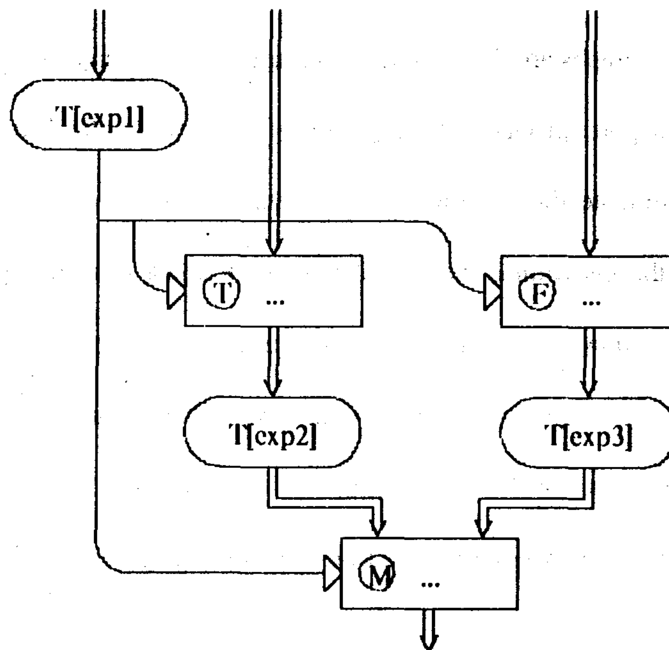
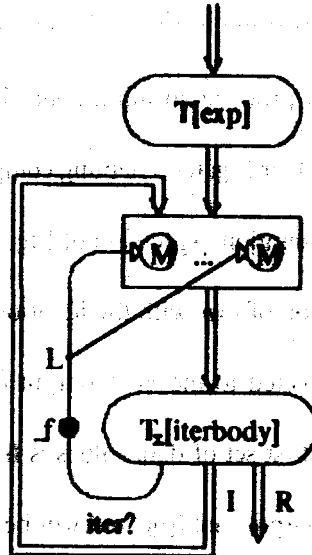


Figure 1.3. T[for *idlist* = *exp* **do** *iterbody* **end**]

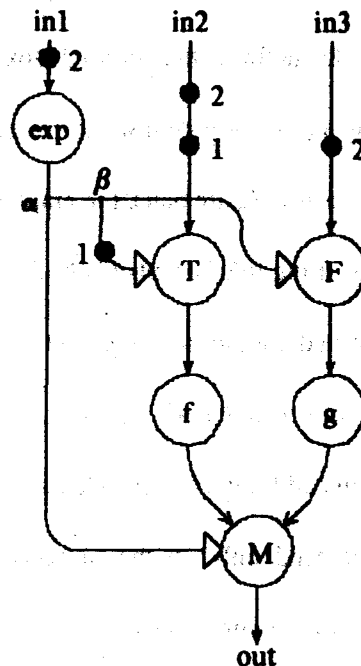


A major concern in generating data flow graph implementations of VAL expressions is ensuring correct modelling of the semantics of each high level construct. In fact, the translation algorithm is part of a two step process giving the operational semantics for the VAL subset: The *operational semantics* of a data flow program is a formal modelling of the execution of the program's data flow graph. The operators composing data flow graphs are *determinate*, meaning that every complete set of inputs to an operator (one for each input port) produces a unique set of outputs. Patil[25] proved that if the operators of a graph are determinate, the graph itself is determinate. Developing operational semantics for VAL is possible due to the determinate nature of its corresponding data flow graphs. Thus, a complete set of inputs to a data flow graph will produce a unique set of outputs, making it necessary to examine only one execution sequence of a graph to derive the result of its execution. The graphs in this thesis are generated from Brock's translation algorithm and are therefore assumed to be *correct* semantic representations based on the operational semantics developed in [3].

1.4 Safety Transformations for Data Flow Graphs

Though we accept the data flow graphs generated by the translation algorithm discussed in the previous section as theoretically correct, their arcs are assumed to be infinite queues -- this prevents their realization. While it might be possible to implement the graphs using sufficiently large finite buffers, this solution may not be acceptable. To examine the problem, consider the state of the graph shown in Figure 1.4. The token configuration shown can be reached by assuming that the graph occurs within an iteration construct which recycles the output of the construct. The second set of inputs shown could therefore have been generated in response to the output resulting from the first set of inputs. Assuming that the output of this first set was produced by propagating tokens through the false branch of the graph, it would be possible for the corresponding T gate inputs (tokens labelled 1) to still be present when the second set of tokens arrives, creating the computation state shown.

Figure 1.4. Unsafe token configuration resulting from infinite queue arcs



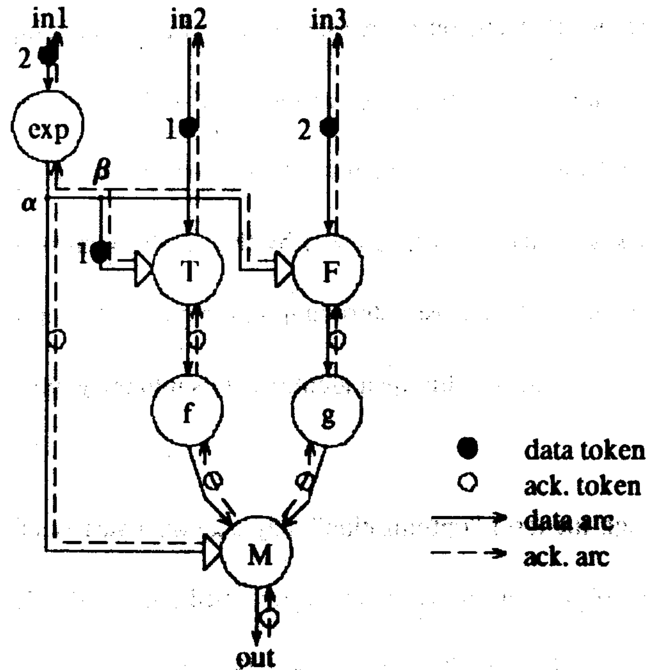
While an implementation of graph arcs as buffers of some constant size (greater than one) could accommodate this configuration, the design of a number of data flow architectures, including that of the Dennis-Misunas data flow machine, cannot support this: The correspondence of graph arcs to machine registers in such designs makes it necessary to consider only those execution sequences which place *at most one token on any given arc at any time.* In the Dennis-Misunas data flow machine, the consequences of placing more than one token on an arc or correspondingly, computing a successive register value before it can be stored, are possible nondeterminism, and deadlock as a result of values queueing up in its distribution network and blocking other values from reaching their destinations[24]. Meeting the one-token operational requirement involves preventing data flow operators from producing new tokens until their output arcs are empty. This behavior is achieved by defining the following firing rule for all graph operators:

> **Operator Firing Rule**: An operator is enabled to fire when all of its needed inputs are present and all of its output arcs are empty.

Application of this rule prevents the Figure 1.4 state from occurring.

While the operator firing rule defines the desired token behavior, the problem of implementation remains. By performing a transformation which replaces each arc of a data flow graph by an appropriate data/acknowledge arc pair (d/a arc pair), the graph's infinite queues are replaced by buffers of capacity one, and the operator firing rule is explicitly built into the graph. This is illustrated in Figure 1.5, which shows the transformed conditional construct of Figure 1.4. The transformation creates arc pairs which hold either a *data* or *acknowledge* token, where the later indicates that its corresponding data arc is empty. With the addition of acknowledge arcs and tokens, firing rules revert to their original specifications which depend only on the presence of tokens on input, including acknowledge, arcs: The operator firing rule requirement that output arcs be empty is ensured by the

Figure 1.5. Transformed Figure 1.4



enabling condition that acknowledge inputs be present.

The keyword used in describing this transformation is *safety*, where the underlying idea and the terminology is rooted in Petri net theory. Chapter 2 discusses the analogy between data flow graphs and Petri nets, and the influence of Petri net theory on the *safety transformation*. Included in the same chapter is a more detailed description of the transformation, and a consideration of its effect on the correctness of graphs.

1.5 Optimizing Transformed Data Flow Graphs

While the transformation of data arcs to d/a arc pairs enables the implementation of data flow graphs, it is imperative to question the cost of the acknowledging scheme and determine the inefficiencies, if any, that are introduced. In fact, there is much to say concerning these issues. Aside from the obvious overhead involved in incorporating acknowledge arcs and tokens, the constraints

which they impose on graph operation may cause bottlenecks. In response to this, we have developed optimization techniques which focus on decreasing overhead and increasing graph throughput. The optimization to *eliminate unneeded acknowledge arcs* is aimed at decreasing overhead, thereby reducing the cost of the transformation scheme. An analysis of data flow graphs of VAL constructs indicates that the effect of certain acknowledge arcs are realized by the graph's control structure, making the arcs unnecessary. On the other hand, increasing throughput, the goal of the optimization to *balance token flow*, is accomplished by introducing additional identity actors into the graph and consequently creating more d/a arc pairs.

Note that though the term "optimization" may take on a variety of meanings, our use of the word is confined to the d/a arc pair transformation described above: Both optimizations consider the number of acknowledges used in data flow graph translations. We do not consider program dependent optimizations which might typically involve modification of a graph's structure, i.e., removal of unnecessary data arcs or operators. This latter form of optimization is analogous to standard optimization techniques for conventional sequential programs and, though not yet fully explored, should prove readily adaptable to data flow.

1.6 Structure of Thesis

Having established a foundation, we proceed to consider the main tasks identified. Chapter 2 expands on the safety transformation introduced in section 1.4, and discusses related relevant theory. Chapters 3 and 4 respectively contain a development of the optimizations to balance token flow, and eliminate unneeded acknowledge arcs. Conclusions are presented in chapter 5 along with suggested areas for future research.

CHAPTER TWO

2.1 The Safety Transformation

The aim of the data/acknowledge arc pair transformation of data flow programs is to implement the operator firing behavior, defined in chapter 1, and restated here:

> **Operator Firing Rule:** An operator is enabled to fire when all of its needed inputs are present and all of its output arcs are empty.

This rule reflects the correspondence of data flow graph arcs to machine registers, which requires that the occurrence of more than one token on any arc be prevented. Restricting data flow graph behavior in this manner is necessary to ensure determinate and deadlock free execution for the architecture assumed. The analogy between the data flow graph characteristics of determinacy and deadlock and the Petri net theory properties of *safety* and *liveness* suggests the use of Petri net theoretical results to formulate and verify the d/a arc pair transformation. In fact, the strategy taken in developing the safety transformation is to extract relevant Petri net concepts and redefine them for data flow graphs.

This chapter proceeds with a closer look at the data flow graph - Petri net analogy, particularly focusing on the possibility of modelling the former with the later. Section 2.3 expands on the safety transformation and its effect in guaranteeing determinate (safe) and deadlock free (live) operation. While showing the existence of the former is straightforward, a significant question concerns whether or not the restrictions imposed to ensure safety affect liveness.

2.2 The Petri Net - Data Flow Graph Analogy

2.2.1 History and Analogy

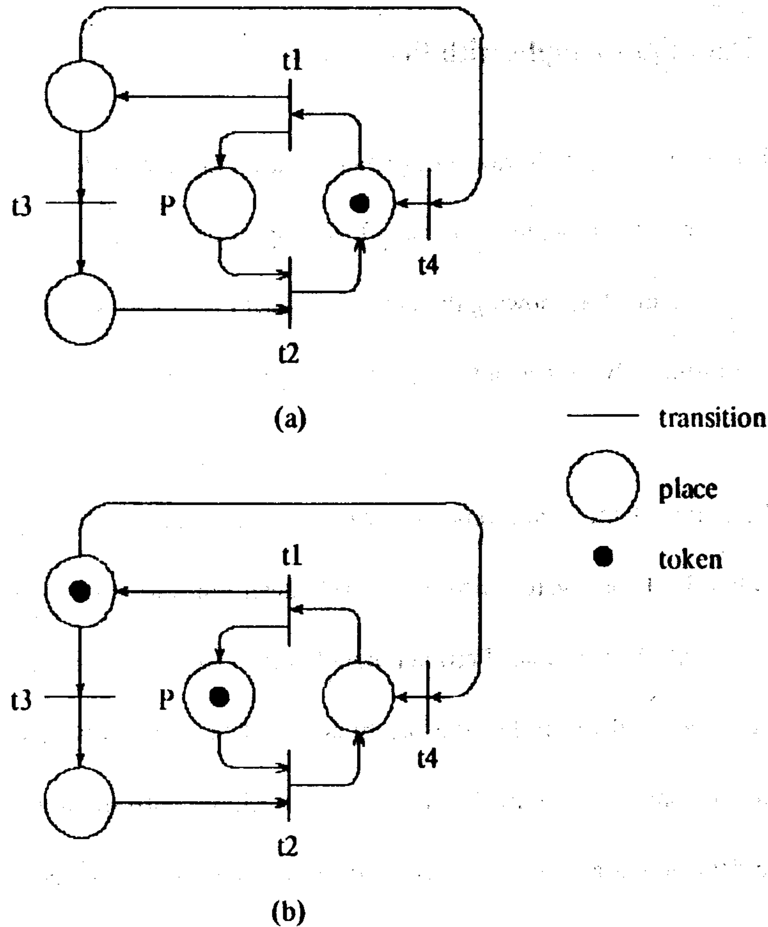
The major contribution of Petri nets is to aid in understanding systems. A closer look at the components of Petri nets seems an essential first step. As shown in the Figure 2.1 example, a Petri net is a graph composed of *transitions* and *places* with an initial *marking* determining the number of *tokens* (pieces of data) residing on each place. The transitions and places correspond respectively to data flow graph operators and arcs. A token must reside on each input place to a transition for it to be *enabled* for firing, where *firing* the transition causes a token on each input place to be removed, and one to appear on each output place. Figures 2.1(a) and (b) respectively show the Petri net token configuration before and after firing transition t1. The operation of a Petri net is *safe* if it behaves according to the following definition:

> **Definition.** For a marking M, a Petri net is *safe* if for every marking M' that can be reached by a sequence of firings from M, there is at most one token on any place.

This is precisely the behavior that we would like data flow graphs to satisfy. Note that the Figure 2.1 graph is, in fact, not safe since the sequence of transition firings: t1; t4; t1 will place two tokens on place P.

We briefly survey the evolution of Petri nets to introduce the theoretical results that could prove applicable to data flow. Petri nets were initially presented by Petri in 1962 [26] and modified by Holt in 1968 [15]. Extensive study of safety and liveness for Petri nets of the *marked graph* and *state machine* varieties has been done by Holt and Commoner [16]. Each of these classes form a particular subset of *free choice* Petri nets. This work has been extended by Michel Hack [14] to include free choice Petri nets. Hack introduces production schemas, similar to data flow graphs, and asserts that

Figure 2.1. Petri net token configuration before and after transition t1 firing



every production schema can be represented by a free choice Petri net. A major result known as the liveness-and-safeness theorem states circumstances under which a free choice net displays these properties. We explore the possibility of using such a result in producing determinate and deadlock free data flow graphs. Guaranteeing safety for free choice Petri nets involves ensuring that every place is part of some directed cycle containing one token. This fact should prove useful in determining if a data flow graph is safe, or in modifying it to be safe: We seek a modelling of data flow graphs by free choice Petri nets which allows us to conclude that a data flow graph is safe and live if its corresponding

Petri net is safe.

2.2.2 Modelling Data Flow Graphs with Petri Nets

The data flow graph firing behavior requirement that no arc ever hold more than one token, forces us to focus on the correspondence of data flow graph arcs to Petri net places. Were the correspondence of places to arcs 1-1, showing the Petri net model places safe would prove the data flow graph arcs "safe". Unfortunately, this is not always the case, as is seen in modelling data flow graph control structures.

Consider the graph of the conditional construct in Figure 2.2. Evaluation of the predicate results in enabling either the T, or F gate which respectively determines whether the input data value **x** will be processed by *f1* or *f2*. A free choice Petri net model of this data flow graph must enable a token to procede down one of two paths to reflect the two branches of the conditional and must merge the paths. A possible model is shown in Figure 2.3. Places and transitions corresponding to particular arcs and operators in the data flow graph are so designated. In comparing the decision structures of the Petri net model and data flow graph, note that place **aa'** in Figure 2.3 represents two arcs in the data flow graph. Although the mapping between places and arcs is clearly not 1-1, the Petri net decision structure presented is essential for allowing a token to take one of two paths. Unfortunately, this makes it more difficult to determine how properties of *place* **aa'** correspond to those of *arcs* **a** and **a'**.

A significant difference in the actual control structure is the absence of specific places and transitions in the model to represent the data flow graph predicate and its output control arcs. Whereas the decision concerning which branch of the conditional construct will be executed is uniquely determined by the output of the predicate, the Petri net is *nondeterministic*, providing a model for all possible decisions: Though each token arriving at place **aa'** will cause only one path of the Petri net to

Figure 2.2. Conditional construct data flow graph

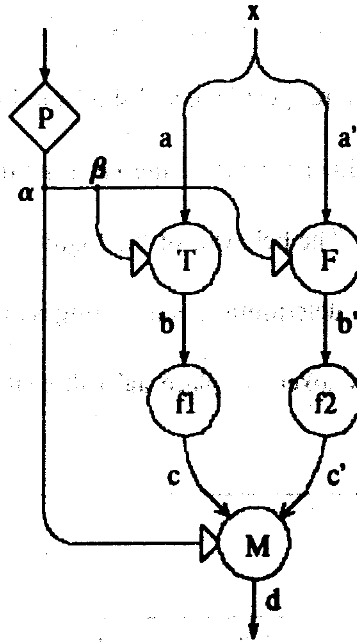
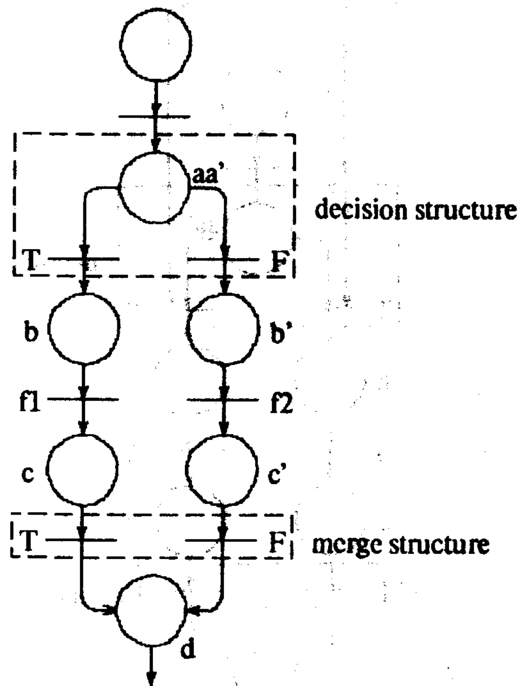
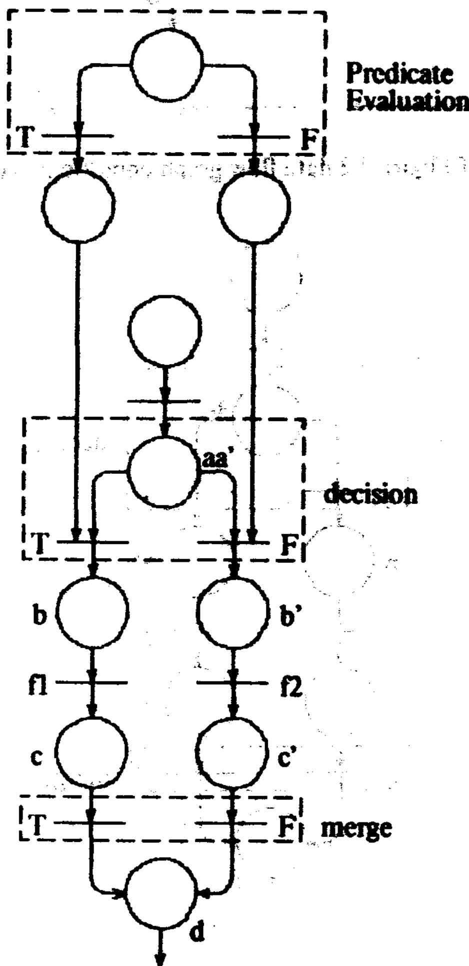


Figure 2.3. Petri net model of Figure 2.2 data flow graph conditional construct



become active, both paths are potential candidates. This situation emphasizes the use of Petri nets as general models for specific systems - in this case, data flow graphs [22]. To remedy the modelling problems of the Figure 2.3 Petri net, a more specific model shown in Figure 2.4 is built which attempts to localize the nondeterminism in an added portion of the Petri net meant to represent the predicate and control arcs of the data flow graph. The behavior of the Figure 2.4 transitions modelling the data flow graph T and F gates is consequently deterministic, since firing is now dictated by the portion of the net labelled "predicate evaluation". A token on place aa' will enable either the T or F transition,

Figure 2.4. Petri net model of Figure 2.2



thereby determining its path.

Though this Petri net modelling of the conditional construct more accurately captures the data flow graph behavior, the portion of the net representing the T and F gates violates the structure defining the *free choice* subset of Petri nets: If a transition following a particular place is firable at a marking M, then all transitions following that place are firable at M. Informally, the definition of a free choice Petri net states that every arc from a place must be either the unique output of the place or unique input to a transition. Thus, the configuration involving place aa' and the T and F transitions in Figure 2.4 violates the free choice property. Since free choice nets form the largest subset of Petri nets for which a developed theory of liveness and safety exists, there is no advantage to pursuing this modelling route. For this reason we change directions, attempting to accomplish our goals more directly by extracting the relevant concepts of Petri net theory and redefining them for data flow.

2.3 The Data/Acknowledge Arc Pair Transformation

2.3.1 Achieving Safe Data Flow Graph Operation

Since the Petri net properties of safety and liveness reflect the behavior we want data flow graphs to display, we attempt to redefine these terms for data flow via the correspondence of arcs and operators to places and transitions.

> **Definition.** For an initial configuration of tokens, a data flow graph is *safe* if every configuration of tokens that can be reached from the initial configuration contains at most one token on any individual arc.
>
> **Definition.** An initialized data flow graph is *live* if a complete set of inputs will eventually cause a complete set of values to appear on the output arcs of the graph.

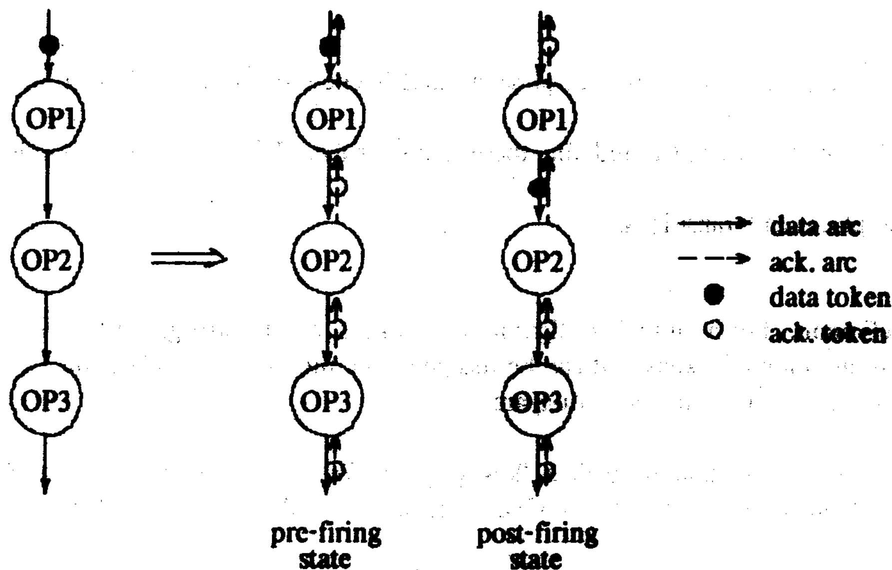
To ensure safe operation in Petri nets, every transition in the net must be part of a one-token directed cycle. Adapting this for data flow is accomplished by introducing initialized *data/acknowledge arc pairs*

(d/a arc pairs) and ensuring that every arc in a data flow graph is part of such a pair.

The mechanics of the transformation illustrated in Figure 2.5 involves replacing each full data arc with an arc pair composed of a full data arc and empty acknowledge arc; and each empty data arc with an arc pair composed of an empty data arc and full acknowledge arc. Alternatively, Brock's T algorithm can be modified to produce graphs with d/a arc pairs, rather than infinite queue arcs. We distinguish the two by terming such an algorithm $T_{d/a}$, as opposed to T_{∞}. The Figure 2.5 graph segment labelled, "pre-firing state" represents the transformation of the graph segment to its left. Having defined this transformation we must verify that, in fact, it accomplishes its intended function - to ensure the safety and liveness of data flow graphs.

An initially transformed graph is potentially safe since each of its arc pairs holds only one token. What must be shown is the preservation of this property under firing. In the pre-firing state of the Figure 2.5 graph segment, OP1 is the only enabled operator since it is the only operator which has

Figure 2.5. D/A arc pair transformation



tokens present on each of its input arcs. Firing OP1 produces the post-firing state shown. The firing action results in the absorption of a token from each of OP1's input arcs and the production of a token on each of its output arcs. Consequently, OP1 is disabled, and OP2 becomes the only enabled operator. More importantly, OP1 cannot be reenabled until it receives both a data, and an acknowledge input, where the appearance of the later is dependent on firing OP2. Firing OP2 will absorb its input data token and produce an acknowledge token, input to OP1. Thus, OP1's output data arc must be empty for it to fire a successive time, producing a new data output. This reasoning, shows the firing behavior dictated by the data/acknowledge arc pair transformation is safe.

2.3.2 Preservation of Liveness

Verifying liveness of data flow graphs under the d/a arc pair transformation is more difficult. Due to its determinate nature, a result obtained from a $\mathbf{T_{d/a}}$ graph will match that of its corresponding $\mathbf{T_{\infty}}$ graph: Any $\mathbf{T_{d/a}}$ graph firing sequence is a legal firing sequence in the $\mathbf{T_{\infty}}$ graph. The question to address is therefore, whether the firing rule constraint causes some $\mathbf{T_{d/a}}$ graph to deadlock that would not have done so in its $\mathbf{T_{\infty}}$ version.

The intuitive feeling that $\mathbf{T_{\infty}}$ graphs and their corresponding $\mathbf{T_{d/a}}$ graphs produce the same results is established via the theorem stated below. Its proof consists of a structural induction on the size of data flow graph expressions. By asserting an induction hypothesis for expression subgraphs, we show that the liveness property holds for $\mathbf{T_{d/a}}$ graphs composed of acyclic interconnections of *exp* subgraphs, or graphs whose top level is a conditional or iteration expression.

In analyzing the $\mathbf{T_{d/a}}$ iteration expression, we have to make some assumption about the behavior of its *iterbody* operator which represents an iteration subgraph. Recall that the $\mathbf{T_{I}}$ translation function produces iterative graphs which have one set of input ports and two sets of output ports

through which values can be iterated or returned, as well as a control output port to signal which of the two occurs. The behavior of the ports of an iterative subgraph within a well-formed live T_{∞} graph can be characterized as follows: When presented with n sets of inputs, the subgraph will produce n *iter?* control values-- k true ($0 \leq k \leq n$) and n-k false; and correspondingly, k sets of I data values and n-k sets of R data values for a total of n data output sets. To prove liveness for a $T_{d/a}$ graph containing an *iterbody* operator, we must first show that the port behavior of $T_{d/a}$ iterative subgraphs is the same as that displayed by T_{∞} iterative subgraphs. This will allow us to assume the desired *iterbody* port behavior, an essential step in proving the expression live.

Proving the correct port behavior for $T_{d/a}$ iterative subgraphs consists of a subproof occurring within the larger inductive proof. Since the iteration expression contains the only instance of an *iterbody* operator, the subproof should naturally appear just prior to proving the $T_{d/a}$ iterative expression live. However, to stem confusion only a statement of the assumed *iterbody* operator port behavior will be made: An outline of the subproof follows the inductive proof. Finally, inherent in this discussion is the assumption that the equivalence of T_{∞} and corresponding $T_{d/a}$ graphs is being shown for graphs which are *well-formed*, where this term is defined as follows:

> **Definition.** A *well-formed* data flow graph is derived from a syntactically correct VAL program using the T_{∞} translation algorithm.

We proceed with the liveness theorem.

> **Theorem:** A well-formed live data flow graph will remain live under the d/a arc pair transformation.

Stated in operational terms:

Any $T_{d/a}$ graph corresponding to a well-formed live T_{∞} graph, when presented with n complete input sets will either:

(1) have produced n complete output sets and absorbed n acknowledge sets on its output d/a arc pairs, and emitted n acknowledge sets on its input d/a arc pairs, or

(2) contain some enabled operator.

Proof:

Basis: A data flow graph consisting of a single functional operator will remain live under the d/a arc pair transformation.

An initialized functional operator is shown in Figure 2.6. On receipt of a complete input set, the operator will be enabled and when fired, will produce an output token absorbing the acknowledge token on its output arc pair, and emit acknowledge tokens on its input arc pairs. Since the operator's output arc pair is the graph output arc pair, within finite time the output token will be absorbed and a corresponding acknowledge token supplied reinitializing the graph. If an nth set of inputs has been presented to the operator and an nth output has not appeared, then the acknowledge arcs of the input arc pairs must have seen their nth acknowledges, n-1 of which were produced by firing operator *f*. This implies that the state of the output d/a arc pair is one of the following: The data arc has its n-1st data

Figure 2.6. Initialized data flow graph of a functional operator

value and the acknowledge arc is empty but has seen n-1 acknowledge tokens; the data arc is empty and the acknowledge arc is holding its nth acknowledge token. In the first case, within finite time the n-1st data value will be absorbed and an nth acknowledge token produced reenabling the operator. In the second case the operator is enabled.

Induction Hypothesis: In response to an nth complete input set, an *exp* operator (expression subgraph) will either:

(1) have produced an nth complete output set and absorbed an nth acknowledge set on its output d/a arc pairs, and emitted an nth acknowledge set on its input d/a arc pairs, or

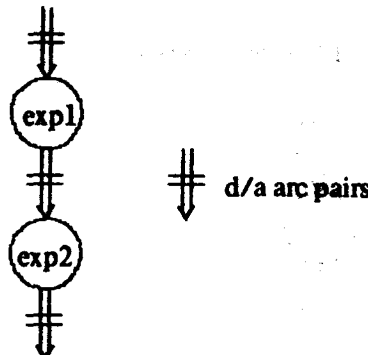
(2) contain some enabled operator.

Acyclic Interconnection of exp operators

Assume that the Figure 2.7 graph has been presented with an nth set of inputs and that it has not produced an nth output set. We will show that the graph must contain an enabled operator.

Suppose the graph has produced j output sets where j<n, and the output arc pairs have had their jth data values absorbed, and are holding their j+1st acknowledge tokens. This implies that exp_2 must have seen at least j input sets. Three possibilities arise.

Figure 2.7. Acyclic interconnection of expression subgraphs



Suppose exp_2 has not yet seen its j+1st input set. Then by the induction hypothesis, since exp_1 has seen its nth input set and only emitted j output sets where j<n, exp_1 contains an enabled operator.

Suppose exp_2 has seen part of its j+1st input set. Then by the induction hypothesis since exp_1 has seen its nth input set and not yet emitted a complete j+1st output set where $j+1 \leq n$, exp_1 contains an enabled operator.

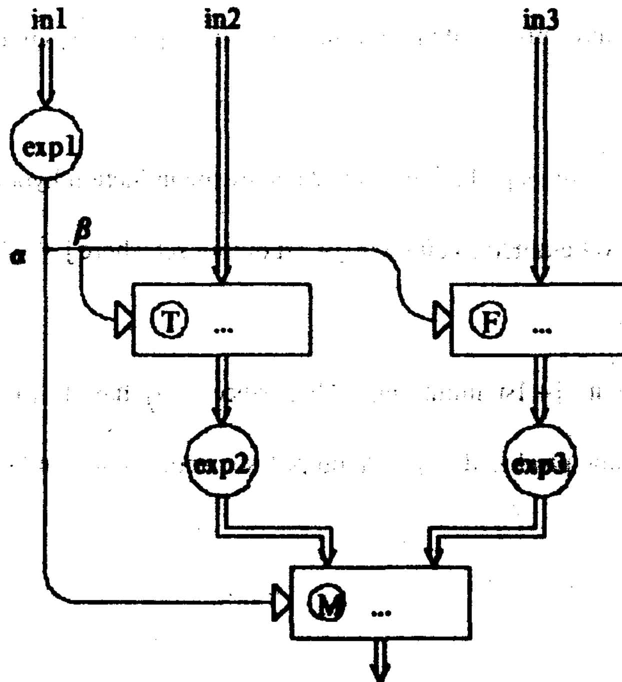
Suppose exp_2 has seen its j+1st input set. Then since exp_2 has its j+1st set of input acknowledges available, it has not produced a j+1st output set and, by the induction hypothesis contains some enabled operator.

Conditional Expression

The conditional expression is shown in Figure 2.8. In its $\mathbf{T}_{\infty}$ form, when presented with n inputs, exp_1 will produce n boolean outputs; k true where $0 \leq k \leq n$ and n-k false. In response to this, the M gates will see a total of n data input sets -- k on their true data input arcs and n-k on their false arcs. These are merged to produce the graph outputs according to the n M gate control inputs (k true; n-k false) which correspond to the M gate data inputs.

An important consequence of the d/a firing restriction is that once a control input value is presented to the M gate, a successive control input cannot appear on that control arc (between α and the M gate) until the M gate fires to absorb the previous value and emit an acknowledge token. The implication of this is that α is prevented from firing a successive time to reenable any gates in the graph before the output set corresponding to the previous control value has been produced. This in turn implies that only one input set will be within the branches of the conditional expression at any time.

Figure 2.8. Conditional construct data flow graph



Assume the graph has received an nth set of inputs. Assume further, that no operator is enabled within exp_1. By the induction hypothesis exp_1 must have produced an nth output set. The d/a arc pair between α and the M gate can be in one of two states. Either the arc pair is holding its n-1st control value, or it is holding an nth acknowledge token. Assume the arc pair is holding its n-1st control value. By the functioning of the graph described above, this implies that the n-1st input set is being processed. Since the graph has received its nth input set, this implies that the T and F gates must have emitted an n-1st set of acknowledges by firing in response to their n-1st set of inputs. We can assume as a result, that either exp_2 or exp_3 becomes enabled. By the induction hypothesis, within finite time we will see the n-1st output set on the appropriate *exp* output data arcs and an nth set of acknowledges on the *exp* input arc pairs. This action enables the M gates which when fired will produce an n-1st set of graph outputs and emit acknowledge tokens along its data and control input arcs. At this point, the arc pair between α and the M gate is in its second possible state-- holding its nth

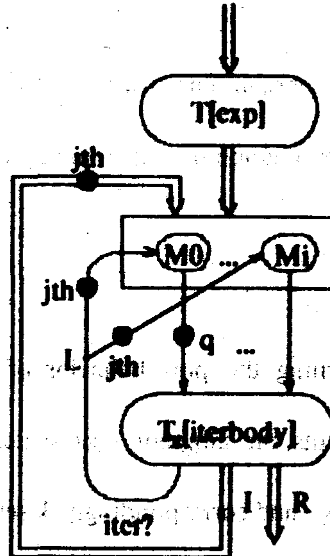
acknowledge. Note that if α is fired, which is now possible, the graph will be in the state it was in when the arc pair between α and the M gate held its n-1st control value. Since within finite time the n-1st set of graph outputs will be absorbed and each graph output will hold an nth acknowledge, we can repeat the above reasoning to show that an nth set of graph outputs is produced.

Iterative Expression

We assert the following concerning the port behavior of the *iterbody* operator: When presented with an nth complete set of inputs, the subgraph represented by *iterbody* will either produce n *iter?* control values -- k true and n-k false, and correspondingly, k sets of I data values and n-k sets of R data values or; will contain some enabled operator.

The iterative data flow graph is shown in Figure 2.9. We can make the following observations concerning the functioning of the graph in its $T_{d/a}$ form. Note that firing copy operator L causes each of the M gates to be presented with the next control input. The implication of this is twofold: Operator L cannot fire until every M gate has fired, absorbing its previous control input and emitting acknowledge tokens; the number of input sets processed by each M gate is either equal to, or one less than the number of control inputs that have been presented to each M gate. The operation of an iterative graph is such that a set of input values will be iterated in response to true *iter?* outputs until *iterbody* produces a false *iter?* output which signals return of the values. We consider these two stages of $T_{d/a}$ graph behavior -- iterating values and returning values, separately. Since the synchronizing affect of copy operator L prevents any interesting overlapping of graph input sets, it suffices to show that when presented with one complete input set the graph will produce an output set without deadlocking. We begin with the return case.

Figure 2.9. Iterative data flow graph



Assume *iter?* produces a false value. By the first implication above, L cannot present the M gates with this value until each has fired to acknowledge L and produce a data input to *iterbody*. Thus *iterbody* must see a complete set of inputs for the M gates to be reinitialized. The stated behavior of *iterbody* dictates that within finite time a complete set of return values will be produced in correspondence with the false *iter?*. Thus if the M gates are reinitialized, a set of outputs is guaranteed without the possibility of deadlocking. The possible ways of a deadlock occurring are considered in the iterative path argument which follows.

We proceed to show that a deadlock does not occur within the iterative path of the graph by assuming the opposite and reaching a contradiction, supporting the conclusion that an enabled operator exists within the graph. Assume that there exists some well-formed live iterative data flow graph which deadlocks under the d/a arc pair transformation. To see how the deadlock occurs we apply the same

sequence of computation steps to a T_{∞} graph and its corresponding $T_{d/a}$ graph, until we reach a state where there exists some operator which is enabled in the T_{∞} graph and not enabled in the $T_{d/a}$ graph. The cause of deadlock must be that an operator in the $T_{d/a}$ graph has its inputs available, but cannot fire due to the presence of a token on its output arc. We attempt to locate this operator, which must be an **M** gate or a gate within *iterbody*. We proceed to consider each case.

Assume merge operator **Mo** is in such a state, and that it has its jth set of iteration inputs available. The token on its output arc, labelled **q**, must be used in producing the **I** iterative input value of some other M gate, say **Mi**. Since the $T_{d/a}$ graph is deadlocked, one of two situations must exist:

(1) The path taken by token **q** through *iterbody* to the **I** input of gate **Mi** is blocked (every arc is full).

(2) Token **q** is input to some operator which lacks some input and therefore is not enabled.

Assume (1). Recall from our preliminary discussion of iterative graph operation, that if token **q** was produced as a result of the j-1st input set, it will be used to produce the jth **I** input of some M gate which, according to the assumption, is blocked. Thus, the token currently residing on the **I** input to that M gate must be part of the j-1st input set or some set previous to the j-1st set. This implies that the M gate has not yet fired j-1 times. But from our knowledge of iterative graph operation, this is not possible since firing copy operator **L** to present each M gate with a jth control input required the prior firing of each M gate a j-1st time sending j-1st acknowledges to **L** -- a contradiction.

Assume (2). Since firing **L** a jth time is only possible if each M gate has fired j-1 times, it must be that a complete set of inputs to *iterbody* is available contradicting the assumption that some input is not present.

Assume the disabled operator occurs as a result of *iterbody* and that its output arc is an I output arc. If the disabled operator has a jth set of inputs available, then they will be used to produce the j+1st I input of some M gate. The token on its output arc, must therefore be a jth I input of that M gate. By the twofold implication stated above, the fact that the disabled operator has its jth inputs available implies that every M gate was presented with a jth control input and has fired either j or j-1 times. Thus the M gate which has its jth I input available, must have fired j-1 times. If we can show that this M gate is enabled, then within finite time it will fire, sending an acknowledge to the blocked operator. Consequently, in finite time there will be an enabled operator within *iterbody*.

We know that the M gate has its inputs available, so it can only be disabled if its output arc is full. Assuming this situation, the token on its output arc must be from the j-1st input set and will be used to produce the jth input of some other M gate. But then we know that within finite time the operator to which this token is input will fire since by the twofold implication, every M gate has fired j-1 times. This simultaneously ensures that the operator has its inputs available and has an empty output arc. The acknowledge necessary to enable the M gate will be sent as a result of firing the operator. Thus, within finite time, the M gate and subsequently the blocked operator in *iterbody* will be enabled.

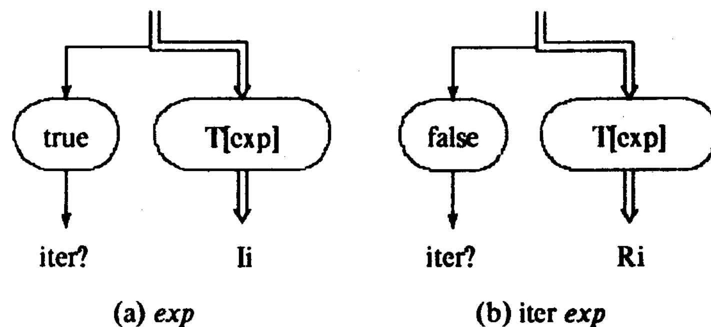
It follows that if the T_{oo} graph is well-formed and live, the corresponding $T_{d/a}$ graph is well-formed and live. Q.E.D.

The subproof concerning port behavior for iterative subgraphs is also inductive in that it must assume a behavior for iterative operators within subgraphs and then prove the behavior for the top level structures defining iterative subgraphs. The behavior to be shown has been stated above at the start of the section of the proof dealing with the iterative expression.

The simplest iterative structures, *exp* and **iter** *exp*, are shown in Figure 2.10. Since the iterative subgraph proof is within the inductive proof above, the induction hypothesis concerning *exp* subgraphs is valid. As a consequence, proving that the Figure 2.10 graphs satisfy the stated behavior is trivial. Establishing this fact for the conditional iteration body, **if** *exp* **then** *iteration*$_1$ **else** *iteration*$_2$, is tedious and will not be presented.

Having developed the data/acknowledge arc pair transformation and shown $\mathbf{T}_{\infty}$ and $\mathbf{T}_{\mathbf{d/a}}$ graphs equivalent, the task of determining the quality of this solution remains. Major concerns to investigate focus on cost and efficiency. Chapters 3 and 4 address these issues and present optimizations of the solution subsequently developed. Example graphs in the remainder of this thesis are assumed to have been produced by algorithm $\mathbf{T}_{\mathbf{d/a}}$. Therefore, though not explicitly shown, all arcs represent d/a arc pairs unless otherwise stated.

Figure 2.10.



(a) *exp* (b) iter *exp*

This empty page was substituted for a blank page in the original document.

CHAPTER THREE

3.1 Balancing Token Flow

The optimization to balance token flow discussed in this chapter addresses certain inefficiencies introduced by the acknowledging scheme presented in chapter 2. Though the d/a arc pair transformation prevents the occurrence of more than one token on an arc at any time, the firing restrictions it imposes are severe, and may significantly curtail concurrency. Specifically, the requirement that an operator receive acknowledge signals on each of its output ports before refiring, unnecessarily delays computation of successive input sets. While ensuring the safe operation of the graph is essential, it is possible to identify which output arcs are potential bottlenecks, and modify each so that it can be safely implemented as a fixed size buffer. The purpose of this change is to effectively enable arcs to hold more than one token, thereby eliminating bottlenecks by allowing computation of successive sets of inputs to "pipeline" through the graph. Safe implementation of these buffers involves the use of identity operators which, when inserted along an arc, act as place holders. Identifying arcs within a graph that may cause bottlenecks, and determining the extent to which they should be buffered are prerequisites to their modification. While the former of these tasks is straightforward, deciding on a buffering strategy is subject to a number of considerations including graph configuration and cost of buffering.

A simple example is presented in section 3.2 which clearly illustrates the problem addressed in this chapter, and serves to motivate the subsequent optimization. This discussion is formalized in an algorithm which produces optimized graphs. The section concludes by pointing out certain subtleties of graph operation and factors not accounted for in formulating the proposed solution. In response to this, section 3.3 introduces a modified version of the section 3.2 algorithm, along with several

comparative studies of graphs in their limited and fully buffered configurations.

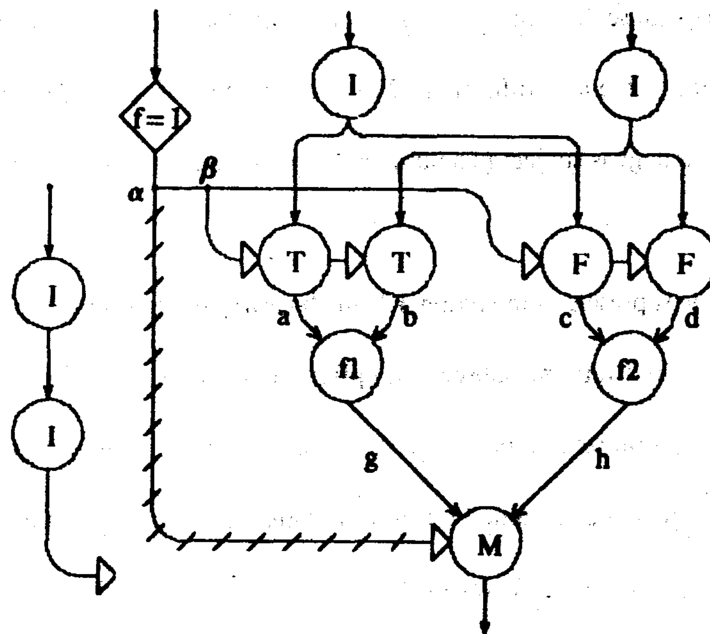
3.2 Formulating the Optimization

3.2.1 Identifying the Source of Bottleneck

The goal of the optimization to balance token flow through a graph is to increase throughput by modifying a graph to allow for maximum pipelining. The bottleneck problem, and therefore application of the optimization, arises in acyclic segments of a data flow graph. A clear illustration of the problem is shown in Figure 3.1, the graph translation of the VAL expression:

if f=1 then *f1* else *f2*

Figure 3.1. Buffering for a conditional expression



The interesting and problematic issues arise when considering the consequence of presenting the graph with multiple input sets. Hopefully, processing of a second set of inputs can begin before outputs of the previous set appear, with the optimum situation being one in which sets of inputs pipeline through the graph. Unfortunately, the control structure of the graph dictates that the overlap in processing of successive sets of inputs be minimal: Only one set of values may be within the branches of the outer conditional at any time. Referring to Figure 3.1, we see that in order for a second set of values to enter the branches of the conditional, both α and β must fire a second time presenting the sets of T and F gates with new control inputs. However, α cannot fire a second time until the M gate to which it also sends a control input has fired to emit an acknowledge. Thus, the d/a arc connecting α and the M gate (marked in Figure 3.1 by slashes), prevents sets of values from pipelining through the graph, creating a bottleneck whose severity depends on the depth of the computation performed within the branches of the conditional.

Eliminating this undesirable behavior so that successive sets of values may pipeline through the graph involves finding a method of enabling node α sooner, consequently allowing the slashed arc to hold more than one token. The ideal situation would be one in which the arc could hold as many tokens as the number of sets of values that could be pipelined through the graph.

3.2.2 Preview of a Solution

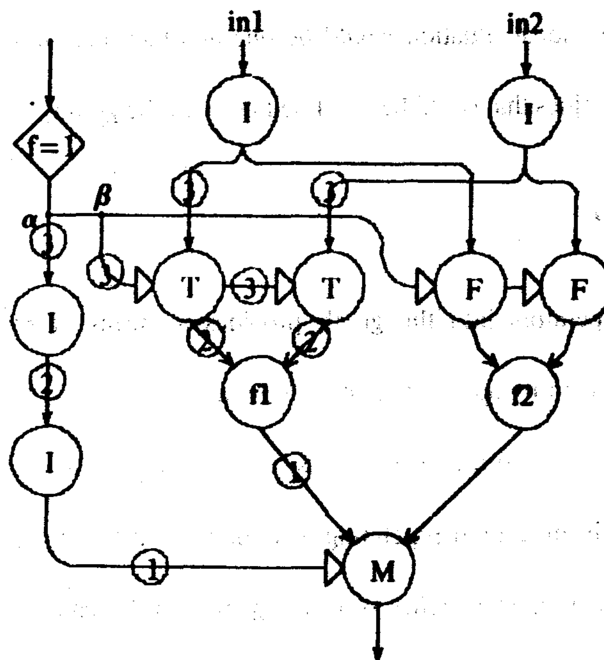
Introducing identity operators into the graph provides a means of realizing the desired behavior. Specifically, inserting identity operators along the slashed arc (Figure 3.1) would break it into d/a arc pair segments, allowing node α to fire several times before forcing the M gate to fire. Using this technique on Figure 3.1 to attain maximum pipelining is accomplished by replacing the slashed arc with the arc segment shown to its immediate left. As a consequence of this change, the state shown in

Figure 3.2 in which three sets of tokens are pipelining through the graph, can be reached. (The token sets have been numbered accordingly for clarity.) Thus the introduction of identity nodes has eliminated the bottleneck. Generalizing this optimization technique requires a determination of the ideal number and location of buffers to be inserted. To respond to such considerations, we attempt to analyze how tokens flow through the graph.

3.2.3 Analyzing Token Flow to Characterize the Solution

Though the data flow computer operates asynchronously and data flow programs nonsequentially, we can model optimum token flow through the graph by assuming a somewhat synchronous behavior. To do this, we analyze the firings within the graph in terms of time units where during any given unit of time all enabled actors must fire and produce a result. This assumption attempts to approximate optimum behavior by preventing an enabled actor from remaining enabled

Figure 3.2. Token configuration allowed by buffering scheme



and thereby slowing up processing for any length of time. Recalling that our aim is to pipeline computation through the graph, we wish to develop a method of modifying the graph so that under this "synchronous behavior" assumption it displays maximum pipelining and consequently, best throughput.

Referring back to Figure 3.1, we note that every input set to the graph results in the production of a token on the control (slashed) arc, and tokens that will either be processed by *f1* or *f2*. While under the synchronous behavior assumption the tokens being processed by these functional operators can move one step through the graph during every time unit, the control token on the slashed arc cannot, and must remain stationary until its corresponding tokens propagate through the graph to enable the M gate. As previously seen, the inability of the control arc to accept a second token prevents any tokens in a successive input set from being pipelined. The dependency between the control arc and the branches of the conditional, and the consequent need to equalize their buffering capacities to attain maximum pipelining has been recognized by the addition of identity nodes shown in Figure 3.2. An algorithm to equalize buffering along graph paths must be able to identify dependencies within a graph and pipeline their paths. This can be accomplished by an arc numbering scheme which compares and equalizes buffering capacities of dependent paths, recognized by identifying functional operators or gates which join two or more paths. An illustration of the algorithm which performs this optimization follows its presentation.

Algorithm to Maximize Pipelining -- I

> Starting from each graph input, descend through the graph assigning consecutive numbers to arcs joining successive sets of operators until a multi-input operator is encountered. Compare the arc numbers on the input arcs of the operator and:
>
> (a) if equal, continue the arc numbering process
>
> (b) if not equal, balance the arcs by inserting identity operators into the lower numbered arcs. Renumber the modified arcs and continue the arc numbering process.

Note that if the operator is an M gate, the comparison and balancing described above must involve all three input arcs, using the highest numbered arc as the goal.

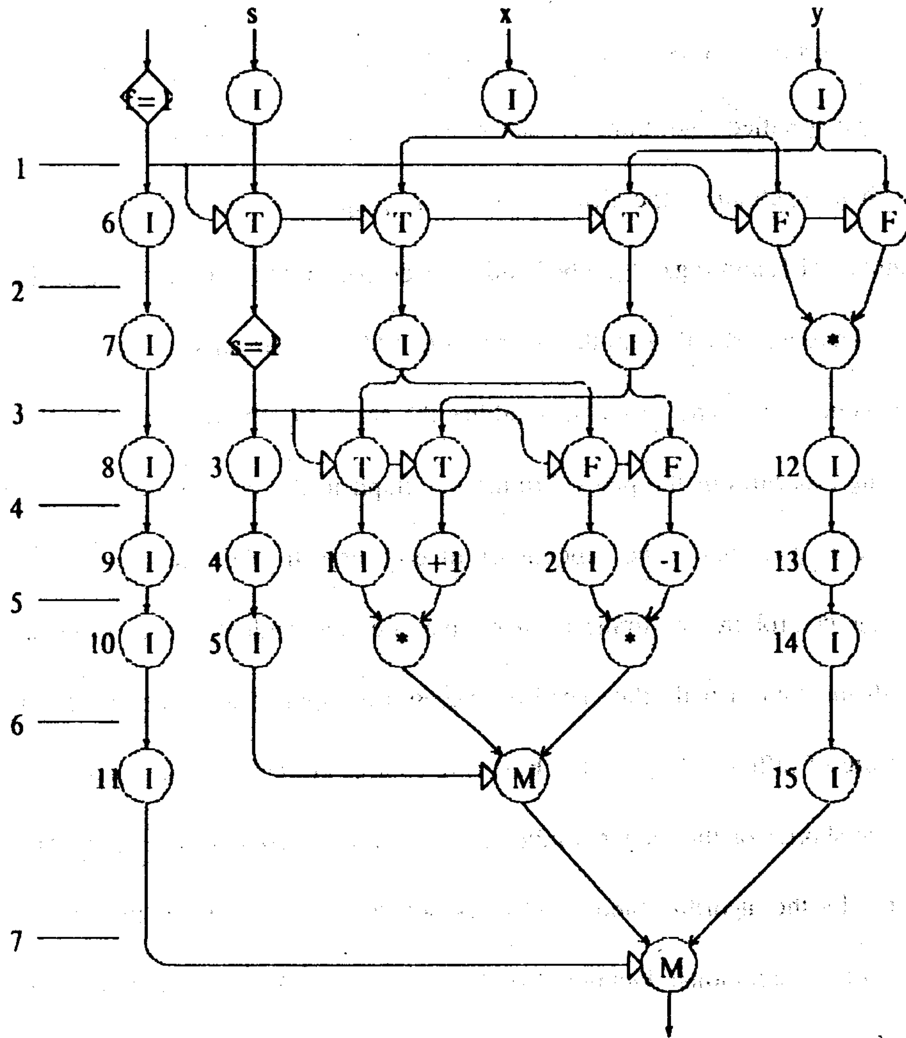
The result of applying this algorithm to the graph translation of the following program segment is shown in Figure 3.3:

if f=1 then if s=1 then x*(y+1) else x*(y-1) end else x*y end

For reference purposes, the added identity nodes have been numbered. The seven numbers shown at the extreme left of the graph result from the arc numbering process, and apply respectively to appropriate arcs moving horizontally across the graph. Nodes I1 and I2 have been added in response to the imbalances which occur when comparing arc numbers on the input arcs to the multiplication operators. I3 through I5 are added in response to the comparison of the input arcs to the inner M gate. Note that, as specified in the algorithm, arc number comparisons involve all three M gate input arcs. Finally, operators I6 through I15 are introduced as a result of comparing input arcs to the outer M gate.

One essential question to ask is whether or not the addition of identity operators changes the functionality of a data flow graph. This can be answered by recognizing that the essence of the change resulting from the application of Algorithm I is to replace some of the one-token arcs of a graph with queues of a given finite length. Since successive identity operators along the arc are separated by d/a

Figure 3.3. Example of maximal pipelining



arc pairs, the graph remains deterministic; and since an identity actor merely passes its input to its output arc, the functionality of the graph is unaffected. These observations ensure the functional equivalence of an optimized graph.

3.2.4 Observations

In developing this example, there are several interesting observations to make concerning the optimization and the specified algorithm. As stated above, the optimization is accomplished by first identifying and then pipelining dependent paths in the graph. While dependencies detected at functional operators and T, and F gates can be handled as described, those resulting from M gates hold some hidden considerations. Recall from the algorithm that M gate comparisons must involve the two data arcs and the control arc. The algorithm modifies the graph to achieve maximum pipelining by equalizing buffering capacities of the paths through the graph to the control arc and two data arcs. However, while the M gate signals the dependency of each branch of the conditional operating in conjunction with the control arc, the branches themselves are independent. Thus, while each branch must pipeline with the control path, they need not necessarily pipeline with each other. If the two conditional paths are of different lengths, the buffering choices available are to equalize the control path with either the shorter or the longer conditional branch, or to equalize all three. The latter of these, implemented by the algorithm above, achieves best throughput but has the disadvantage of causing the insertion of additional identity operators in the shorter conditional branch. Thus, maximum pipelining may be achieved at the expense of including a number of unnecessary identity operations. The other two choices recognize the independence of the two conditional paths and avoid excess buffering, but possibly at the cost of reduced throughput.

A factor not yet considered which interacts with this pipelining choice is the token distribution effect on the graph of a particular succession of input sets. In Figure 3.3 each input set can take any of three paths corresponding to the three possible states of *f* and *s*. This makes it unlikely that any one of the three paths will be filled with tokens, more likely that the control arc to the inner M gate will be filled and certain that a continuing succession of input sets will fill the control arc to the outer M gate.

If we consider a pattern of input sets such that no one of the three paths is taken twice in a row, identity nodes I1 and I2 would be unnecessary and could be removed without decreasing the throughput. In fact, many of the identity nodes could be removed with no effect. Certainly, the frequency with which graph paths are taken is an important factor in choosing a buffering strategy. An illustration of this point will be seen in the examples in section 3.3.2.

In identifying some tradeoffs and options to consider in maximally pipelining data flow graphs, it has become unclear whether or not this approach is always optimal. Perhaps the advantages of a less pipelined graph are worth a decrease in throughput. Some key issues influencing such a decision might include cost of identity operations, processor utilization, token flow patterns and width and depth of program. Though complete consideration of these would require knowledge of the machine and particular application, we attempt to illustrate the type of analysis that might be useful and necessary in making the choice.

3.3 Full vs. Limited Buffering

3.3.1 Achieving Limited Buffering

Having questioned whether fully balancing a graph is always necessary or optimal, we proceed by comparing several graphs in both their limited and fully buffered versions to uncover the tradeoff issues. A discussion of limited buffering including how it can be achieved and to what extent $T_{d/a}$ graphs display it is a necessary preliminary.

The difference between full and limited buffering in a data flow graph is seen in the time delay between successive firings of its operators. In a fully buffered graph, assuming synchronous behavior, the time delay between repeated firings of any particular operator should be one unit: An operator which fires at time one should receive acknowledges from its successive operators during time unit two,

reenabling it to fire during time unit three. In a graph displaying limited buffering, the delay between an operator's firing and receiving appropriate acknowledge signals may be several time units, thereby slowing repeated firings of the particular operator as well as all successive operators.

Presently, the $T_{d/a}$ translation algorithm produces data flow graphs in which every data arc is paired with an acknowledge arc. We could however, have considered an algorithm which caused acknowledge arcs to span two data arcs by having each acknowledge arc link alternate rather than successive operators. The consequence of such a scheme would be a delay in the sending of acknowledge signals and hence, a graph displaying limited buffering. While section 3.3.2 discusses an example data flow graph so configured, this approach is undesirable since it requires a significant modification to the present translation algorithm. The necessity for such an action is also unjustified since in most cases, $T_{d/a}$ graphs already display limited buffering, as did the Figure 3.3 graph which was modified to achieve full pipelining via Algorithm I. A slight revision of this algorithm will allow us to produce data flow graphs which display limited buffering to some predefined degree. For example, it is possible to specify that the delay in sending acknowledge signals be no greater than two time units. The algorithm shown below produces graphs meeting this requirement. While the purpose of Algorithm I was to equalize buffering of dependent paths within a graph, the modification to the algorithm ensures that dependent path lengths are within a specified bound. By allowing a graph to be easily reconfigured to display different degrees of pipelining, the algorithm provides a feasible and practical control method of studying varying levels of buffering in a graph. The modified algorithm is presented below as Algorithm II:

Algorithm to Limit Pipelining -- II

Starting from each graph input, descend through the graph assigning consecutive numbers to arcs joining successive sets of operators until a multi-input operator is encountered. Compare the arc numbers on the input arcs of the operator and:

(a) if the difference is less than or equal to 2, continue the arc numbering process

(b) if the difference is greater than 2, insert identity operators into the lower numbered arcs to reduce the difference to 2. Renumber the modified arcs and continue the arc numbering process.

An application of Algorithm II appears in section 3.3.2 where it is applied to the Figure 3.3 graph. We are now prepared to proceed with several graph comparisons of full and limited buffering.

3.3.2 Examples of Full vs. Limited Buffering

This section presents two data flow graphs, in both their fully and partially buffered versions. The first example achieves limited pipelining by relinking acknowledge arcs between alternate actors as described in section 3.3.1 above, while the second example is modified for limited pipelining via Algorithm II. Our aim in each case is to compare the functioning of each example's graph configurations with respect to throughput, acknowledgement overhead, and overall concurrency. The following assumptions are made concerning the graphs' operation:

(1) Graph firings occur according to the "synchronous behavior" pattern described in section 3.2.3

(2) All graphs are produced by $T_{d/a}$ with data/acknowledge arc pairs used throughout.

We begin with a simple example in an effort to establish some analysis guidelines. The program segment shown in Figure 3.4 is a composition of binary operators which, if produced by $T_{d/a}$, should display full pipelining. Thus, there is no need to apply either algorithm to this program segment. Rather, studying this graph in limited pipelined form will require its restructuring so that acknowledge arcs link alternate operators. The flow of tokens through the graph for multiple input sets can be followed using Table 3.1. (For convenience, the operators in the graph have been numbered.) The initial state of the graph, given in Table 3.1 at time 0, shows inputs (IN) available to OP1 and OP2, and acknowledges (A) present on all other arc pairs. Progressing through the table along the time axis, we see that at time 1, OP1 and OP2 fire and acknowledge (F/A), making inputs available to OP3, and producing acknowledges on their input arc pairs. During time unit 2, OP3 fires sending a result token to OP4, which consequently becomes enabled, and acknowledge tokens to OP1 and OP2. At the same time, a new set of inputs can appear on the input arcs to OP1 and OP2 so that they become reenabled. In time unit 3, OP1, OP2 and OP4 fire, sending appropriate data and acknowledge tokens which enable OP3 and OP5. These then fire in time unit 4, enabling OP4 as well as OP1 and OP2 which, as in time unit 2, concurrently receive a new set of inputs. This time unit is significant since during it, the output resulting from the first input set is produced. Following through the next few time units shows that due to the acknowledging scheme, the best throughput possible for a fully pipelined graph is an output every <u>second</u> time unit: Outputs resulting from the second and third input sets appear in time units 6 and 8 respectively.

An examination of the table shows that once the "pipe is full", (time unit 3), the operator firings of the graph can be grouped into two alternating sets, and consequently, the graph's operation is characterized by two alternating states. SET1 consists of OP1, OP2 and OP4 firings, or those of the first and third levels of the graph shown in Figure 3.4. SET2 consists of OP3 and OP5 firings which

Figure 3.4. Maximum pipelining in a simple data flow graph

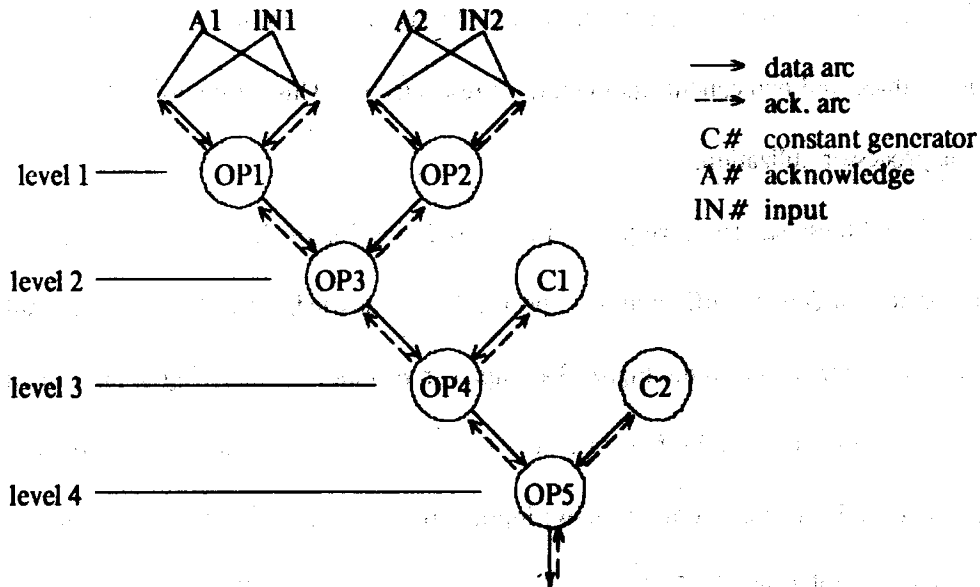


Table 3.1. Flow of tokens for Figure 3.4

operators	0	1	2	3	4	5	6	7	8
					output 1		output 2		output 3
OP5	A	A	A	IN	F/A	IN	F/A	IN	F/A
OP4	A	A	IN	F/A	IN	F/A	IN	F/A	A
OP3	A	IN	F/A	IN	F/A	IN	F/A	A	A
OP2	IN	F/A	IN	F/A	IN	F/A	A	A	A
OP1	IN	F/A	IN	F/A	IN	F/A	A	A	A
time	0	1	2	3	4	5	6	7	8
						set 1 state	set 2 state		

set 1: OP1, OP2, OP4; set 2: OP3, OP5

IN inputs present
F/A fire and acknowledge
A acknowledges present on input and output arc pairs

---

compose the second and fourth levels of the graph. Using the fact that alternating levels of the graph fire concurrently, we see that the minimum number of concurrent operations (assuming a full pipe) is

the number of levels divided by 2. The maximum number is found by computing the sum of the width of each firable level for each of the two sets to determine the larger. For the Figure 3.4 graph, SET1 and SET2 consist of three and two concurrent operations respectively. This information should prove useful in analyzing processor utilization.

Having gathered these statistics, we proceed by considering Figure 3.5 which shows the same graph, but in its limited pipelined configuration. Specifically, acknowledge arcs link alternate rather than successive actors. Comparisons to the Figure 3.4 graph can be made by analyzing the information contained in Table 3.2, which follows the flow of tokens through this graph. The initial configuration of the graph, specified in Table 3.2 at time 0, shows inputs present on OP1 and OP2 input arcs, and acknowledges available to OP3 and OP5. During time unit one, OP1 and OP2 fire to enable OP3. Note however, that the OP1 and OP2 input arcs are not acknowledged at this time as they were in the Figure 3.4 configuration. Acknowledgement of OP1 and OP2 is now dependent on OP3's firing which occurs during time unit 2, delaying the arrival of a new set of inputs until time unit 3. Firing of OP4 which also occurs during time 3 enables OP5 which can fire to produce an output at time 4. Again, reenabling of OP3 has been delayed to this time unit, 4, when it receives an acknowledge from OP5 and inputs as a result of OP1 and OP2 firing. Time unit 4 is significant in that an output is produced. However, following the operation of the graph for three input sets shows that the delay in acknowledging operators has reduced the throughput to an output every third time unit: The second and third input sets produce outputs in time units 7 and 10 respectively.

Analyzing the operation of the graph using Table 3.2, we see that the acknowledging scheme allows every third level in the graph to fire concurrently, thereby partitioning the graph into three interleaving sets of operators. Referring to Figure 3.5, levels 1 and 4 fire together, as would levels 2 and 5, and levels 3 and 6, were the graph to be extended. Corresponding respectively to these three groups

Figure 3.5. Limited pipelining in a simple data flow graph

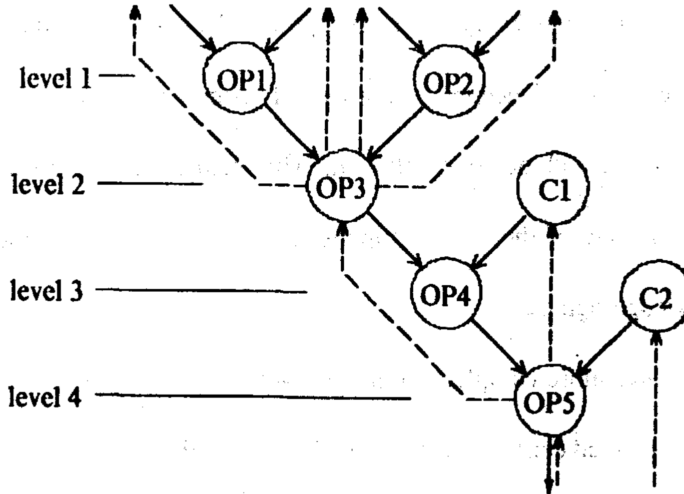


Table 3.2. Flow of tokens for Figure 3.5

operators	0	1	2	3	4	5	6	7	8	9	10
					output 1			output 2			output 3
OP5	A		A	IN	F/A		IN	F/A		IN	F/A
OP4	—		IN	F		IN	F		IN	F	
OP3	A	IN	F/A	—	IN	F/A		IN	F/A		A
OP2	IN	F		IN	F		IN	F			
OP1	IN	F		IN	F		IN	F			
time	0	1	2	3	4	5	6	7	8	9	10
					state 1	state 2	state 3				

IN	all inputs available
F/A	fire and acknowledge
A	acknowledges available
F	fire

are three states, shown in Table 3.2. Were the graph to be presented with continuous sets of inputs, its operation would rotate among these three states. For this graph, the number of concurrent operations per state beginning with state 1 are: three, one, and one, (determined by computing the sum of the width of each firable level for each of the states.) Using the "concurrent operations per state" statistic

shows that the Figure 3.4 graph alternates between processing three and two operations while the Figure 3.5 graph processes three operations every third time unit and only one during each of the intermediate two time units. The lower variance in the number of concurrent operations per state in the Figure 3.4 graph suggests that it will be more efficient with respect to processor utilization. Consequently, the only main advantage of the limited pipelined configuration is a reduction in the overhead associated with acknowledge signals.

A second more involved and more complete example, applies this analysis to the Figure 3.3 graph, which appears in its fully pipelined configuration. Note that unlike the previous example, which translates directly into its fully buffered state under $\mathbf{T}_{d/a}$, the production of the Figure 3.3 graph required the application of Algorithm I. The most significant point to note is the need to insert 15 identity operators to attain full pipelining. This represents approximately a 50% increase in the number of operators in the graph, making the cost of identity operators vs. the benefit of increased throughput and concurrency an extremely important issue to consider for an actual data flow machine and application.

Table 3.3 presents a summary of the token flow through the fully pipelined graph (Figure 3.3), assuming the control token produced by the predicate test involving f is true. For each time unit, the level of operators firing rather than the particular operators will be specified, where the assignment of levels to operators is indicated in Figure 3.6. The total number of operators for each level as well as their breakdown in terms of inserted identity operators as opposed to graph operators (all others) is also given. Thus, referring to Table 3.3, the second line states that during time unit 1, the first level of operators fired, all four of which were graph operations. During time unit 2, the second level of operators fired, one of which was an identity operator and five, graph operators. From the previous example, we know that successive sets of inputs will step through the graph with alternate levels firing

Figure 3.6. Fully pipelined data flow graph

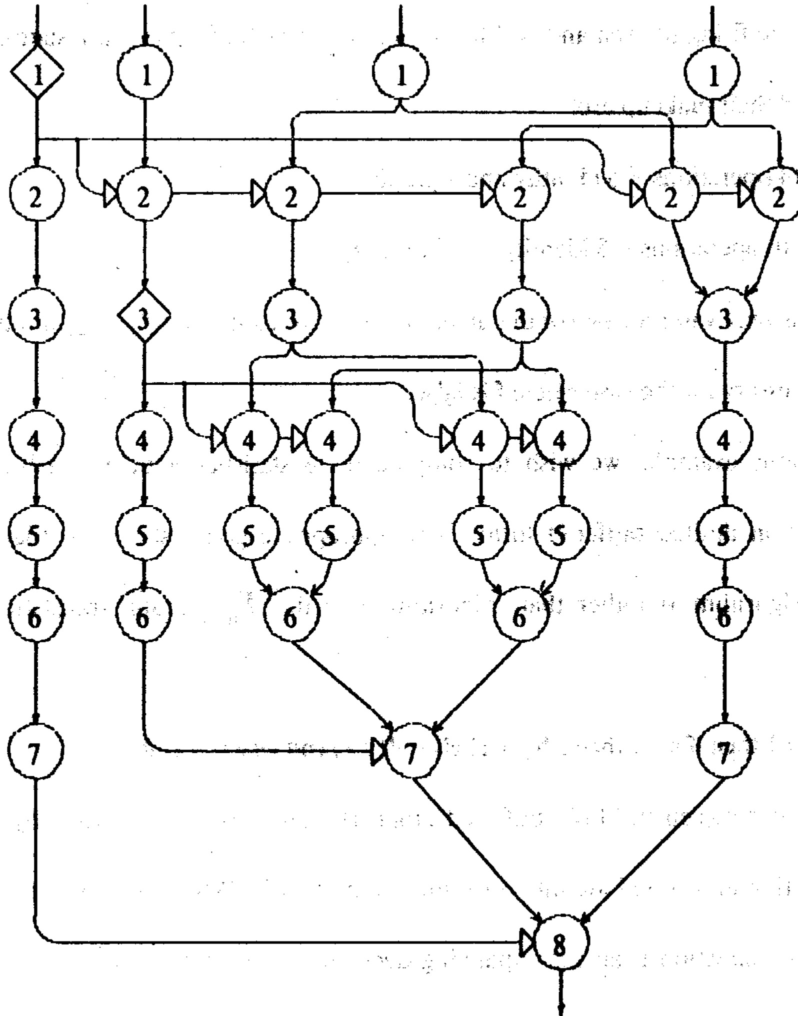
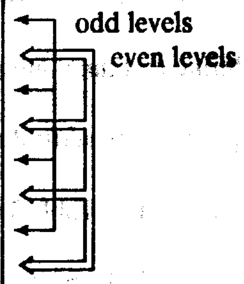


Table 3.3. Token flow through Figure 3.6

time	firing level	total operations	identity/graph
0	inputs available		
1	1	4	0/4
2	2	6	1/5
3	3	4	1/3
4	4	6	2/4
5	5	4	3/1
6	6	3	2/1
7	7	2	1/1
8	8	1	0/1



concurrently to produce an output every second time unit. In terms of the table this behavior corresponds to the alternate firing of even and odd levels, where for each of these firing states, the total number of operations and their makeup are:

ODD 14 operations -- 5 identity and 9 graph

EVEN 16 operations -- 5 identity and 11 graph

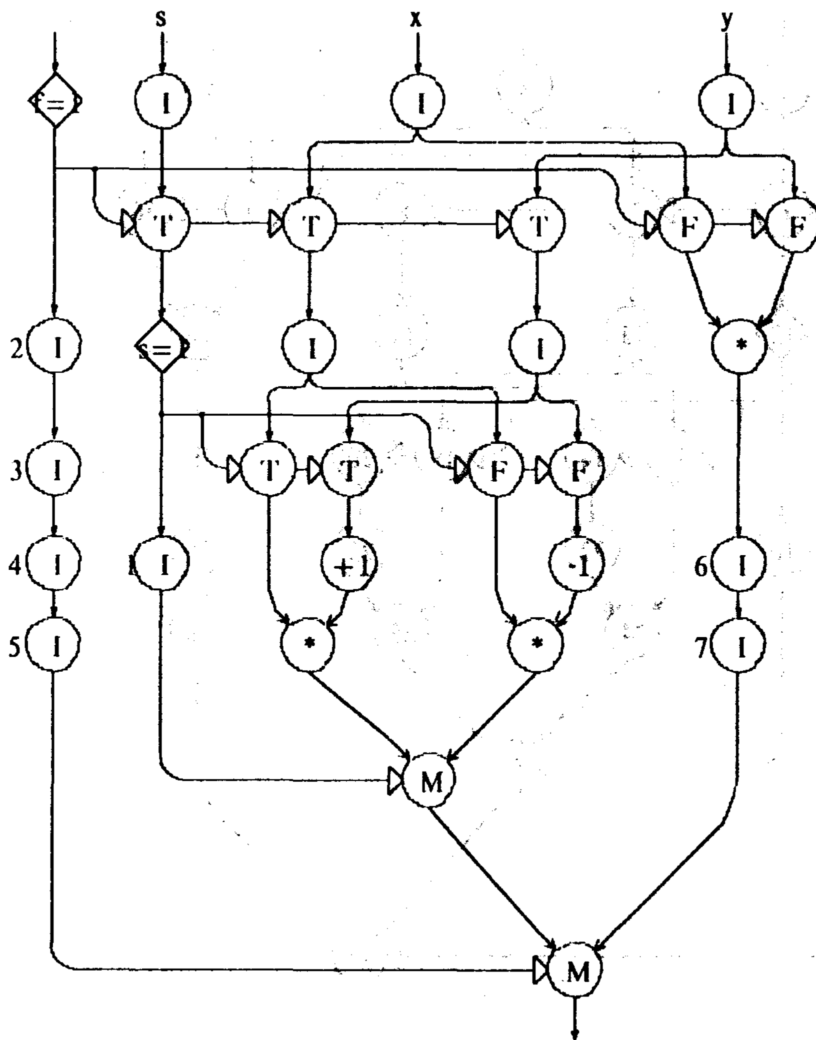
The Table 3.3 summary is only valid for two of the three possible *f* and *s* states; true-true and true-false. A separate analysis is necessary for the case where **f** is false.

As in the previous example, we wish to compare these statistics with an analysis of the functioning of the graph in limited buffered form. The appropriate graph shown in Figure 3.7 is obtained by applying Algorithm II rather than Algorithm I to the $T_{d/a}$ graph translation of the expression:

if f=1 then if s=1 then x*(y+1) else x*(y-1) end else x*y end

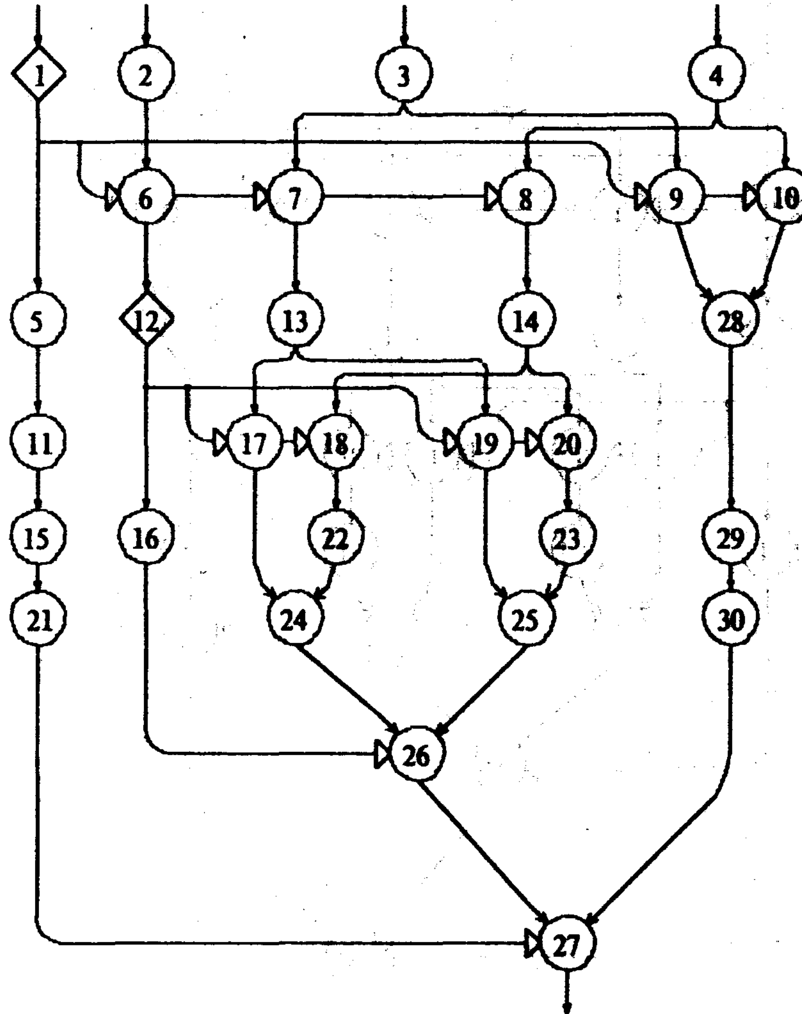
The most striking contrast between the fully buffered graph (Figure 3.3) and this partially buffered version is the large reduction in inserted identity operators from 15 to 7. What remains to be explored is whether the cost of this reduction is an accompanying decrease in performance (see also [27]). To determine this, we examine several token flow analyses for the Figure 3.7 graph, derived by considering different successions of input sets. The first example performs the analysis for four sets of inputs which all follow the same computation path; true-true. The progression of tokens through the graph can be followed via Table 3.4. The numbers in each box in the table represent the specific operators which fire during that time unit (given by the horizontal axis), as a result of tokens from the appropriate input set (given by the vertical axis), where the operators have been numbered as shown in Figure 3.8. Referring to this graph, Table 3.4 shows that, (assuming input set 1 is initially available), during the first time unit actors 1, 2, 3, and 4 will fire enabling actors 5 through 10 which will fire during the second time unit.

Figure 3.7. Example of limited pipelining



The second input set becomes present (P) during the second time unit so that operators 1 through 4 may fire in response to this second set during the third time unit along with operators 11 through 14 which fire in response to the first set. In this manner, the progress of the four sets of inputs through the graph can be followed. The time units during which the corresponding outputs appear have been noted in Table 3.4 along the top horizontal axis. This information reveals the expected decrease in throughput which may or may not be acceptable depending on the application.

Figure 3.8. Numbered Figure 3.7 graph to be used in conjunction with Tables 3.4 and 3.5



As mentioned earlier, the probability of a succession of input sets taking the same computation path is small. Therefore, a second analysis for this partially pipelined graph appears in Table 3.5 assuming input sets 1 through 4 take the computation paths true-true, true-false, false and true-true respectively. The table reveals that for this pattern of input sets the limited buffering scheme has no effect on the throughput, which remains optimal at an output produced every second time unit. This example confirms the point previously made concerning the significance of a sequence of input sets. A

								output 1
IN4 tt						P	1, 2, 3, 4	5, 8, 9, 10
IN3 tt				P	1, 2, 3, 4	5, 6, 7, 8, 9,10	11,14	13
IN2 tt		P	1, 2, 3, 4	5, 6, 7, 8, 9,10	11,12,13 14	15,18,19 20	17,22	16,24
IN1 tt	1, 2, 3, 4	5, 6, 7, 8, 9,10	11,12,13 14	15,16,17 18,19,20	21,22	24	26	27
time	1	2	3	4	5	6	7	8

		output 2				output 3		output 4
IN4 tt	7	6	11,12,13 14	15,18,19 20	17,22	16,24	21,26	27
IN3 tt	12	15,16,17 18,19,20	21,22	24	26	27		
IN2 tt	21,26	27						
IN1 tt								
time	9	10	11	12	13	14	15	16

Table 3.4. Token flow of four input sets through Figure 3.8 for computation path true-true.

								output 1
IN4 tt						P	1, 2, 3, 4	5, 6, 7, 8, 9,10
IN3 f				P	1, 2, 3, 4	5, 6, 7, 8, 9,10	11,28	29
IN2 tf		P	1, 2, 3, 4	5, 6, 7, 8, 9,10	11,12,13 14	15,18,19 20	17,23	16,25
IN1 tt	1, 2, 3, 4	5, 6, 7, 8, 9,10	11,12,13 14	15,16,17 18,19,20	21,22	24	26	27
time	1	2	3	4	5	6	7	8

		output 2		output 3		output 4		
IN4 tt	7,12,13, 14	16,17,18 19,20	11,22	15,24	21,26	27		
IN3 f	30	15	21	27				
IN2 tf	26	27						
IN1 tt								
	9	10	11	12	13	14	15	16

Table 3.5. Token flow of four input sets through Figure 3.8 for computation paths true-true, true-false, false, true-true.

further analysis of input sets for this data flow graph may reveal that, in fact, it is rarely necessary or best to transform the graph into fully buffered form.

3.3.3 Additional Considerations

Once an actual data flow machine is available, a study of the tradeoff of throughput for number of inserted identity operators should provide insight into the direction to take concerning optimization. Perhaps this information in combination with a particular application will indicate other optimization possibilities; for instance, concentrating efforts on only the main source of bottleneck within a graph. For the conditional construct this point appears to be the control arc to the M gate. Modifications of Algorithm I similar to the one which produced Algorithm II could also be weighed more realistically as alternative approaches.

A final point to note in the consideration of this buffering optimization strategy is the type of construct for which it is appropriate. The examples above which involve conditional constructs and general compositions of operators, turn out to be fairly representative of the type of graphs for which this optimization is applicable. In fact, this optimization approach is basically inappropriate for an iterative process whose function is to modify and recycle a single set of inputs at a time -- a process which does not involve pipelining (however, subgraphs within an iteration may be pipelined). For such constructs, a different optimization technique must be developed. This alternative strategy, which aims to minimize the number of acknowledges in a graph by eliminating those which are unnecessary, is the topic of the next chapter.

CHAPTER FOUR

4.1 Eliminating Unneeded Acknowledge Arcs

This chapter explores an optimization technique for removing unnecessary acknowledge arcs in a data flow graph. Though the uniform substitution of data/acknowledge arc pairs for data arcs yields a correct implementation of a data flow graph, the acknowledging scheme is costly. The overhead of processing acknowledge packets is felt in the routing networks and instruction cells of the data flow computer which must respectively handle the resulting increase in traffic and bookkeeping. Thus, there is value in questioning whether or not all acknowledge arcs are needed. While it is easy to find example data flow graphs containing arcs for which an acknowledge is unnecessary, methodical identification of such instances is extremely difficult due to an often context dependent decision: The graph configuration and particular construct under consideration are key factors in determining acknowledge arc removal. In response to this fact, the strategy to eliminate unneeded acknowledge arcs focuses on individual VAL constructs, attempting to identify candidate d/a arc pairs and provide a corresponding set of rules specifying conditions. Recursive application of the resulting set of rules to a data flow graph derived from a VAL program can then be used to test each candidate arc pair for removal of its acknowledge arc.

The following section considers the possibility of using Petri net theory to govern acknowledge arc removal, and subsequently discloses certain data flow graph operational characteristics important to the optimization process. Sections 4.3 and 4.4 develop acknowledge arc removal rules for the VAL conditional and iteration constructs respectively. The later section includes several example graphs illustrating applications of the rules formulated for the iteration construct.

4.2 Considerations for Acknowledge Arc Removal

The concern in removing acknowledge arcs from a data flow graph is whether the safe operation which the arcs ensure is maintained. Though we attempt once again to use Petri net theory as a guide, this strategy is discouraged not only as a consequence of the chapter 2 discussion, but as a result of examining T and F gate operators which display a fundamentally different behavior than that of transitions. A look at the operation of these gates and their effect on token flow shows the difficulty in using Petri Net theory, and motivates the formulation of new requirements for safe removal of acknowledge arcs in data flow graphs.

The role of the transition in Petri net theory is analogous to that of the <u>functional</u> data flow operator: Firing a transition moves tokens on input places to output places of the transition. The T and F gate function which allows a computation to proceed in one of two ways, is accomplished by the Petri net configuration shown in Figure 2.3 and repeated below in Figure 4.1. The essential difference in the operation of this Petri net is that once one of its T, or F transitions fires to place the input token on a particular path, the transition controlling entrance to the alternate path is no longer enabled. In a conditional data flow graph, when the gates corresponding to the control input fire, the opposite gates remain enabled and must fire to absorb their inputs as is shown in Figure 4.2.

Here the assumption is that the control input to the Figure 4.2 gates was <u>true</u>, allowing a token to flow through the T gate to enable operator *f1*. The data flow graph behavior will allow an output to be produced at the M gate independent of whether or not the input presented to the F gate has been absorbed. This phenomenon does not occur in the Figure 4.1 Petri net since an input token is switched down one of the two paths leaving no extra tokens behind. The significance of this difference becomes clear when considering the possibility of iterative graph configurations. If we focus on the input arcs to the F gate, and view the Figure 4.2 graph as the body of an iteration construct which recycles its output

Figure 4.1. Petri net model of the conditional construct

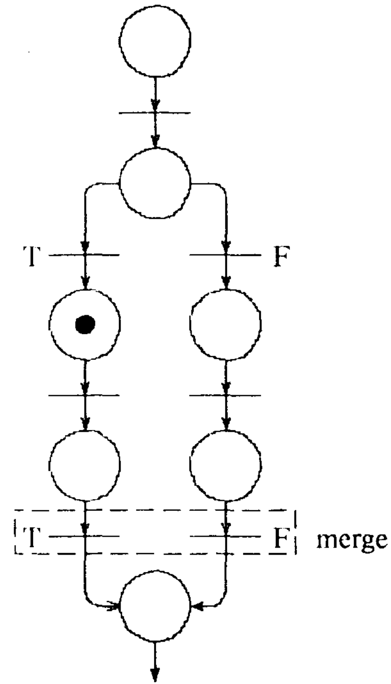
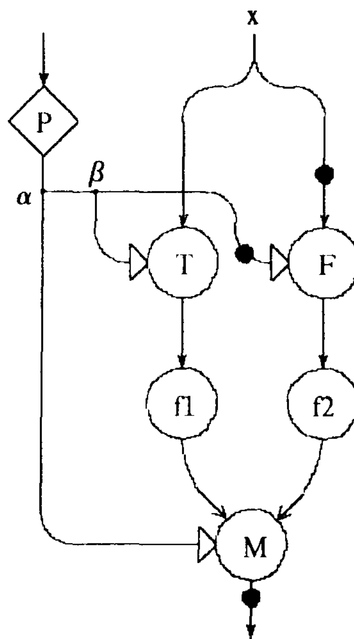


Figure 4.2. Conditional construct data flow graph



token, ensuring conflict-free operation requires that the input arcs to the F gate be d/a arc pairs.

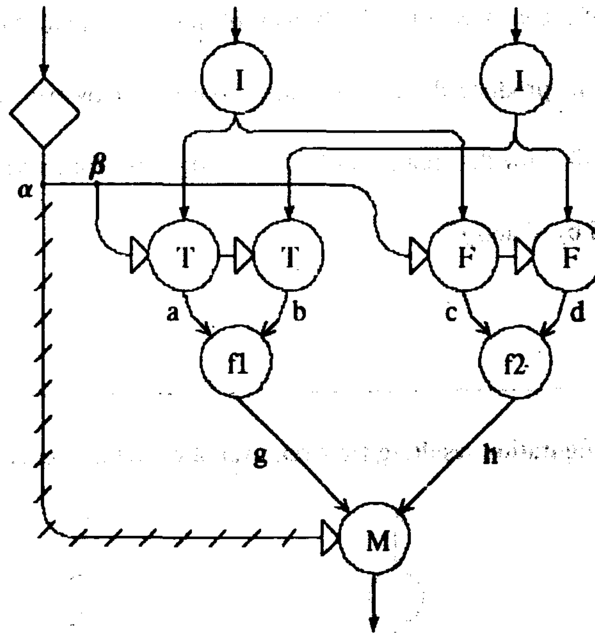
Since the possibility of a similar conflict is absent from the Petri net modelling of the data flow graph, the difference in operation of the two renders Petri nets insufficient as a guide for acknowledge arc removal in data flow graphs. As a result, the applicability of Petri net theory to the process of identifying candidate arc pairs is limited. Instead, the strategy followed examines the various VAL constructs to develop rules specifying conditions for acknowledge arc removal for each candidate arc pair identified in a construct.

An implication of this conditional construct behavior is that the acknowledge arcs of the input arc pairs to a T or F gate cannot be removed since the presence of a token on an acknowledge arc is the only way to guarantee the absence of a token on a corresponding data arc: A T or F gate output arc gives no indication of the state of the gate's input arcs since firing may or may not produce an output token. An illustration of additional problems resulting from T and F gate behavior in combination with the possibility of nesting conditionals appears in the next section.

4.3 Analysis of the Conditional Construct

To illustrate the analysis needed for finding removable acknowledge arcs we consider the data flow graph translation of a general conditional construct, shown in Figure 4.3. We begin by focusing on the slashed arc pair connecting α and the M gate. Recall that the behavior of this arc pair is such that it cannot accept a second token until the M gate fires to process the previous control token, and send an acknowledge token to α. This guarantees that a second set of tokens cannot be within the branches of the conditional until processing of the preceding set has completed. While overcoming the restricting behavior of this arc pair was the aim of the chapter 3 optimization designed to balance token flow in the graph, it is an advantage to the process of removing acknowledge arcs as is seen by following an input

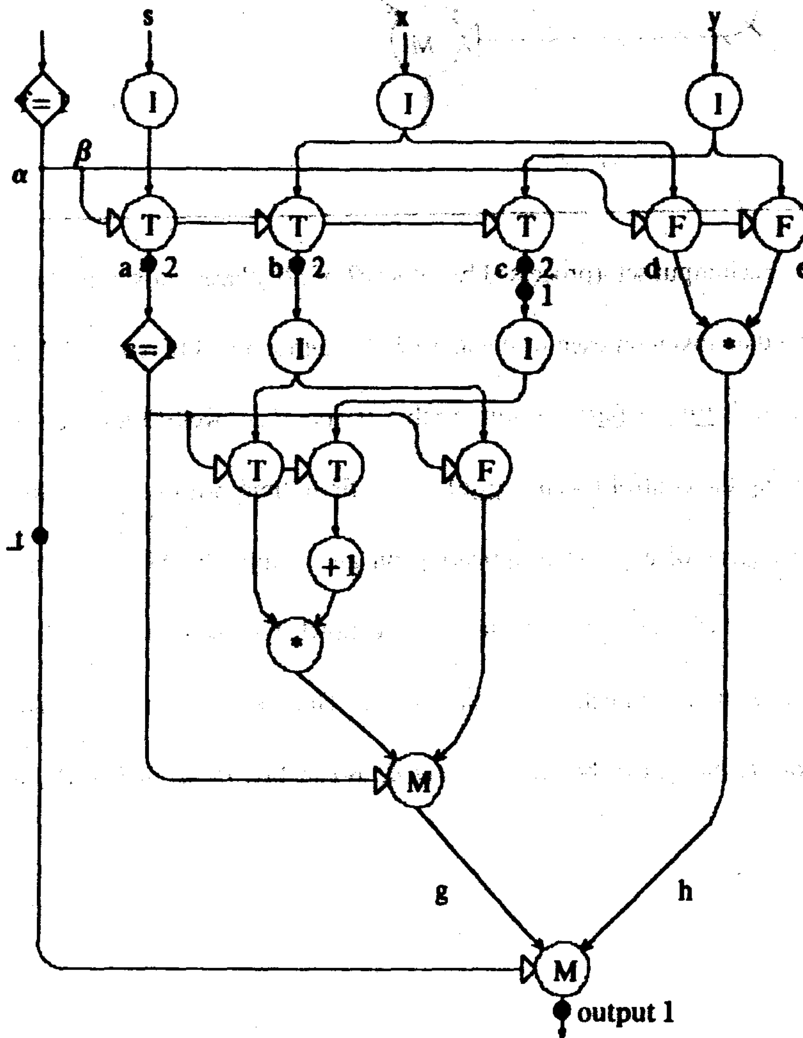
Figure 4.3. T[if *exp* then *f1* else *f2*]



set through the graph. Each input set (processed by either *f1* of *f2*), places a token on the control input arc of the M gate and a data token on each of the arcs labeled either a and b, or c and d, depending on whether the control token is true or false. Assuming that *f1* and *f2* are well-formed, an output should appear on arc g (assuming the control token is true) within finite time, with no possibility of a second token appearing on arc g, or of any token appearing on arc h until the M gate fires. This event simultaneously processes the token on arc g and sends an acknowledge token to α, consequent to which a successive input set may enter a branch of the conditional. The token flow behavior guarantees that the acknowledge arc of arc pair g can be safely removed, as can that of arc pair h (by an analogous argument).

One might be tempted to remove the acknowledge arcs from arc pairs a, b, c, and d under the assumption that once a set of tokens has entered a branch of the conditional, the tokens must be used by the appropriate function to produce the corresponding output. However, a consideration of the Figure 4.4 data flow graph will show that removal of acknowledge arcs for these arc pairs is dependent on the subgraphs represented by *f1* and *f2*.

Figure 4.4. Unsafe token configuration resulting from removal of c's acknowledge arc



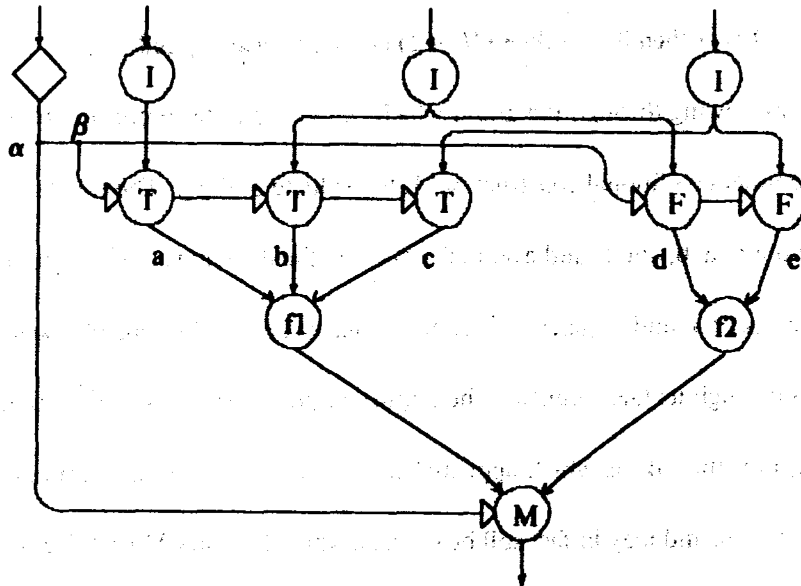
The Figure 4.4 graph is a translation of the following VAL program segment:

if f=1 then if s=1 then x*(y+1) else x end else x*y end

Consider a set of tokens flowing through the graph which causes the outer predicate, f=1, to evaluate to true and that of the inner conditional construct, s=1, to evaluate to false. The tokens on inputs s, x, and y should appear on arcs a, b, and c, and eventually become the data and control input tokens to the inner conditional construct's T and F gates. Since the inner conditional's control token is false, the computation proceeds through its false branch. The important point to note is that continuation of the computation, only requires the tokens which appeared on arcs a and b. The token on arc c need not propagate through the graph, and may in fact still be on arc c when the outer M gate fires to produce an output and an acknowledge token, allowing the processing of a successive set of values to begin. Were a set of inputs to flow through the graph in this manner, removal of c's acknowledge arc would make it possible to reach the unsafe token configuration shown in Figure 4.4. (The tokens are numbered to indicate the input set to which they belong). This behavior is a consequence of T and F gate functioning, the foundation of the conditional construct structure.

Understanding the analysis is aided by Figure 4.5 which generalizes the Figure 4.4 graph to expose the subgraph structure. The Figure 4.4 example shows that the necessity of acknowledge arcs for d/a arc pairs a through e is dependent on whether or not their values are guaranteed to be used in producing the outputs of the appropriate subgraph (*f1* or *f2* of Figure 4.5). Examining subgraphs *f1* and *f2*, which respectively represent the inner conditional construct and multiplication operator of Figure 4.4, reveals that tokens arriving on arcs a, b, d, and e must be used to produce their corresponding output, while the need of a token arriving on arc c is dependent on the outcome of the inner decision operator. Therefore, c's acknowledge arc must remain but those of arc pairs a, b, d, and e can be removed.

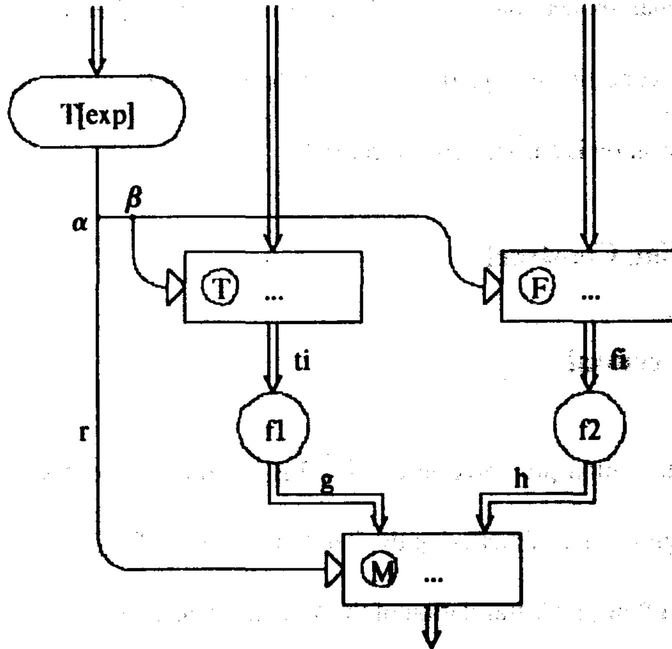
Figure 4.5. Generalized version of Figure 4.4 data flow graph



This analysis, specific to the conditional construct, results in designating all input arc pairs to the *f1* or *f2* subgraphs subject to rule C1, shown in Figure 4.6, for determining acknowledge arc removal. While the rule serves to identify, and state conditions under which certain arcs within the conditional construct may not need acknowledges, it gives no method for testing the conditions. This requires a recursive look at the constructs composing subgraphs *f1* and *f2*, the strategy just used in analyzing arc pairs a through e in the Figure 4.4 example. It is interesting to note that the analysis can be applied at the source level by first recognizing that subgraph *f1* was a conditional construct, and then taking the intersection of variables appearing in its **then** and **else** clauses. Variables found in the intersection are guaranteed to be used in producing the output of the construct. Therefore, arcs in the data flow graph corresponding to these variables should not require acknowledges.

Finally, we look at the only arc in the conditional construct of Figure 3.3 not yet analyzed -- the control (slashed) arc connecting α and the M gate. While the elimination of acknowledge arcs within our example conditional construct has been largely dependent on the existence of this controlling arc

Figure 4.6. Acknowledge arc removal rules for the conditional construct



Arcs	Removal rule
g, h	unconditional
t_i, f_i	C1
r	C2

C1: The acknowledge arc of an input arc pair to subgraph *f1* or *f2* may be removed if any token arriving on the arc must be used in producing the output of the subgraph.

C2: The acknowledge arc of the control arc connecting α and the M gate can be removed if the acknowledge arcs of the output arc pairs of the M gate has been removed.

---

pair's acknowledge, its presence enables the acknowledge of an inner conditional construct's control arc to be removed. The argument to justify this is the same as that used to explain the removal of arc g's acknowledge. Consequently, in the general conditional construct the control arc between α and the M

gate is marked as candidate for acknowledge arc removal, and is subject to rule C2 shown in Figure 4.6.

This completes the analysis necessary for performing the optimization to remove unneeded acknowledge arcs within the conditional construct. As a second example, we discuss the iteration construct for which this optimization is particularly appropriate.

4.4 Analysis of the Iteration Construct

4.4.1 Acknowledge Arc Removal

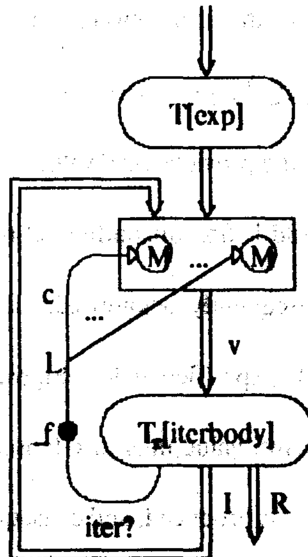
The fact that the optimization presented in chapter 3 is specific to acyclic segments of a data flow graph, emphasizes the significance of analyzing the iteration construct for unneeded acknowledge arcs. Figure 4.7 shows the data flow graph translation of the VAL iteration expression:

for *idlist* = *exp* **do** *iterbody* **end**

The function of this construct is to evaluate *exp* and then perform *iterbody*, which outputs an *iter?* control value and a set of data values on either its I (iteration) or R (return) output arcs, depending respectively on whether the *iter?* output value is true or false. Successive evaluations of *iterbody* are made until a false *iter?* value is produced, at which time evaluation of the construct with a new set of inputs can begin.

The function of the *iter?* arc is to provide the control value to the group of M gates which present successive sets of inputs to the iteration body. The arc is initialized with a false control value to ensure proper selection of the first set of data values. Assuming that the *iter?* value is dependent on at least some of the M gate inputs, a number of them must fire before a second *iter?* value is produced. This necessarily implies the firing of copy operator "L" in Figure 4.7, to present the M gates with *iter?* control inputs needed to enable them -- consequently ensuring that the *iter?* output arc of *iterbody* must be empty for a successive *iter?* value to be produced. As a result, the acknowledge arc of this arc pair

Figure 4.7. Acknowledge arc removal rules for the interation construct



Arc	Removal rule
iter?	unconditional
c_i	T1
I_i	T2
v_i	T3

T1: The acknowledge arc for an arc pair between operator L and the sequence of M gates can be removed if its data value must be used in producing the *iter?* value.

T2: The acknowledge arc of an I (iteration) arc pair can be removed if either

(1) The iteration body cannot emit a value on that output arc until it has absorbed the corresponding input value on the corresponding input arc.

(2) The *iter?* value depends on the corresponding input arc.

T3: The acknowledge arc of a v_i arc pair can be removed if the arc pair is not input to a T, or F gate, and the *iter?* output value of *iterbody* depends on the v_i arc value.

(between *iterbody* and L) can be removed.

No such guarantee can be made for the arcs between copy operator L and the M gates, since the *iter?* value need not be a function of every M gate input. This implies the possibility of producing a second *iter?* value before every instance of the previous *iter?* value appearing on the arc pairs between L and the M gates has been absorbed. Should L fire, unconditional removal of the acknowledge arcs of these arc pairs could cause a conflict. Consequently, acknowledge arcs of these arc pairs are marked as conditionally removable subject to rule T1, specified below Figure 4.7: M gates whose data value inputs are used in producing the *iter?* control value must fire (absorbing the current *iter?* value, their control input) before a successive *iter?* value is produced, and consequently, need no acknowledge arcs.

Examining the form of the iteration construct's *iterbody* is a necessary preliminary to determining acknowledge arc removal for the remaining arc pairs in the iterative graph. Since the function of the construct is to iterate or return a set of values based on some boolean function, *iterbody* must contain a conditional. The BNF specification of VAL confirms this via the production:

$$iterbody ::= \textbf{if } exp \textbf{ then } iterbody_1 \textbf{ else } iterbody_2 \textbf{ end}$$

Figure 4.8 shows the data flow graph translation of this conditional iteration body. Graph inputs are respectively presented to the subgraph representing either $iterbody_1$ or $iterbody_2$ via T, or F gates, as a result of evaluating *exp*. The selected subgraph will produce a set of outputs at either its I (iteration) or R (return) output ports according to its *iter?* output value: true for I outputs; false for R outputs. The *iter?* output values of the iteration body subgraphs, along with the output of the predicate subgraph, *exp*, are the inputs to the IC gate which controls the graph output ports. The IC gate has three outputs: A graph *iter?*, and an I control value and R control value which provide control inputs to two sets of M gates respectively merging the I and R data outputs of the iteration body subgraphs to produce graph outputs. A more detailed specification of the IC gate is given in Table 4.1. Functioning of the

Figure 4.8. T_I[if *exp* then *iterbody*$_1$ else *iterbody*$_2$ end]

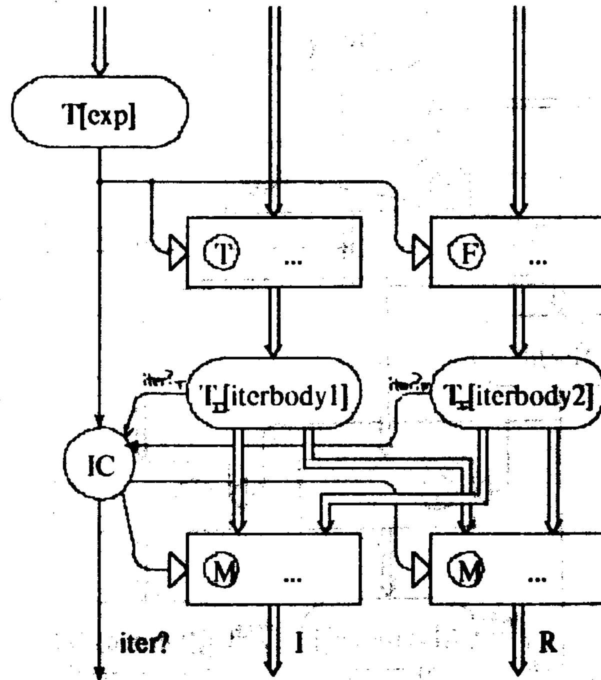


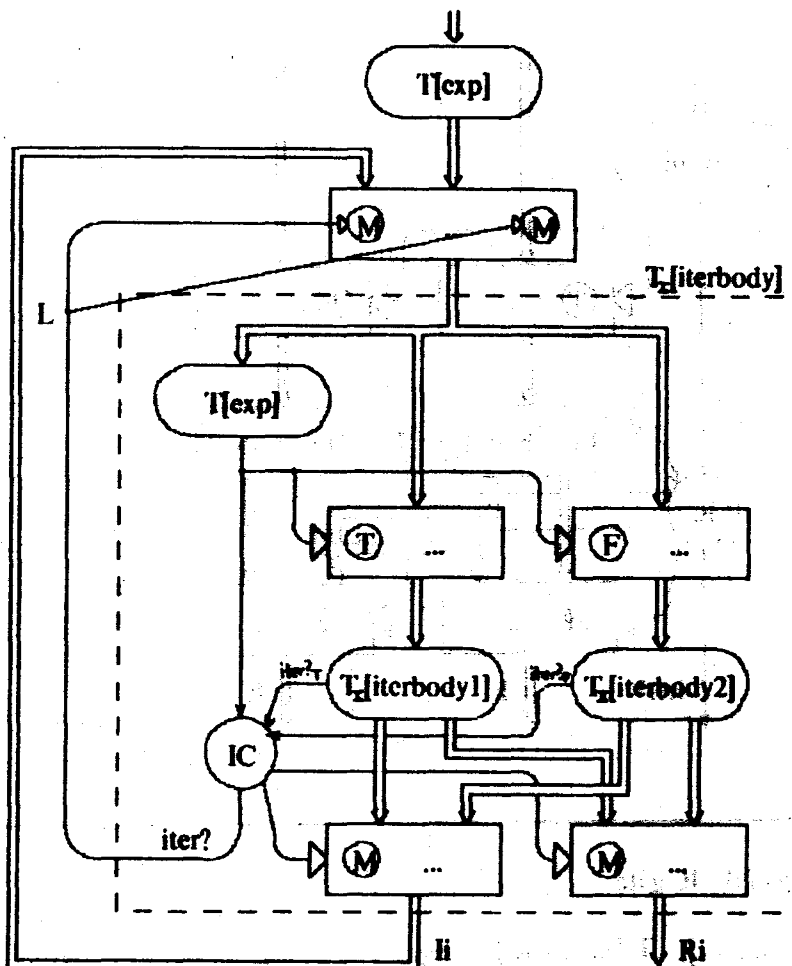
Table 4.1. Functioning of the IC gate.

predicate control	T_I[*iteration*$_1$] *iter?*	T_I[*iteration*$_2$] *iter?*	graph *iter?*	I control	R control
true	true	--	true	true	--
true	false	--	false	--	true
false	--	true	true	false	--
false	--	false	false	--	false
error	--	--	false	--	error

conditional iteration body is seen through several examples presented in section 4.4.2.

By replacing *iterbody* in the Figure 4.7 graph of the iteration construct with the Figure 4.8 conditional iteration body to produce Figure 4.9, the I output arcs of the iteration construct can be analyzed for acknowledge arc removal.

Figure 4.9. Iterative data flow graph containing *iterbody* subgraph of Figure 4.8



Recall that a set of output values should appear on the I arcs for each <u>true</u> *iter?* value produced. The acknowledge arc of a particular I output arc may be removed if either of two conditions is satisfied. The first is the case in which production of the output value is dependent on the corresponding input value; appearance of a new value implies absorption of the previous value. At first glance this would seem to occur always. In fact, it is possible to produce a second output on some I arc without using the previous value, as is seen in the example in section 4.4.2. The second condition under which an I acknowledge arc can be removed is dependence of the *iter?* value on the corresponding I input.

To understand this we look at the IC gate in Figure 4.9, one of whose output arcs is *iter?*. Firing the IC gate will produce values on two of its three output arcs; the *iter?* arc and either the iteration or return control arcs which respectively provide control input values for M gates connected to the graph I and R output ports. Until the IC gate fires, these M gates will not be enabled. A set of values appearing on the graph I output ports therefore requires the prior IC gate firing to produce the M gate control values, as well as an *iter?* value. It is clear that if this *iter?* value is dependent on a particular I arc input value, that I arc must be empty for it to receive a successive iteration value. Consequently, acknowledge arcs of I arc pairs satisfying this *iter?* dependence are not needed. The two conditions under which the acknowledge arc of an I arc pair can be removed are summarized in rule T2, of Figure 4.7.

To complete analysis of the iteration construct we discuss the input arc pairs to the iteration body labelled v_i, in Figure 4.7. Testing for acknowledge arc removal must be done individually for each v_i according to the following guidelines. If the arc pair is input to a T, or F gate, the acknowledge arc must remain: This follows from the discussion of T and F gate behavior. If the arc pair is input to a functional operator or M gate, the acknowledge arc can be removed if the *iter?* output of the *iterbody* is dependent on the v_i arc value. The v_i arc pairs are outputs of a set of M gates controlled by the graph *iter?* value. In order to remove the acknowledge arc of a particular v_i arc pair, it is not sufficient that the v_i value be needed in computing a successive iterative value in response to a <u>true</u> *iter?* output. The v_i value must also have been used before a new input value resulting from a <u>false</u> *iter?* value appears. This is ensured if *iter?* depends on the v_i value. Rule T3 shown in Figure 4.7 states the acknowledge arc removal rule for the v_i arc pairs.

4.4.2 Acknowledge Arc Removal in Iterative Programs

To apply the acknowledge arc removal rules developed in the previous section, we begin with the simple but familiar factorial algorithm expressed as the following VAL program:

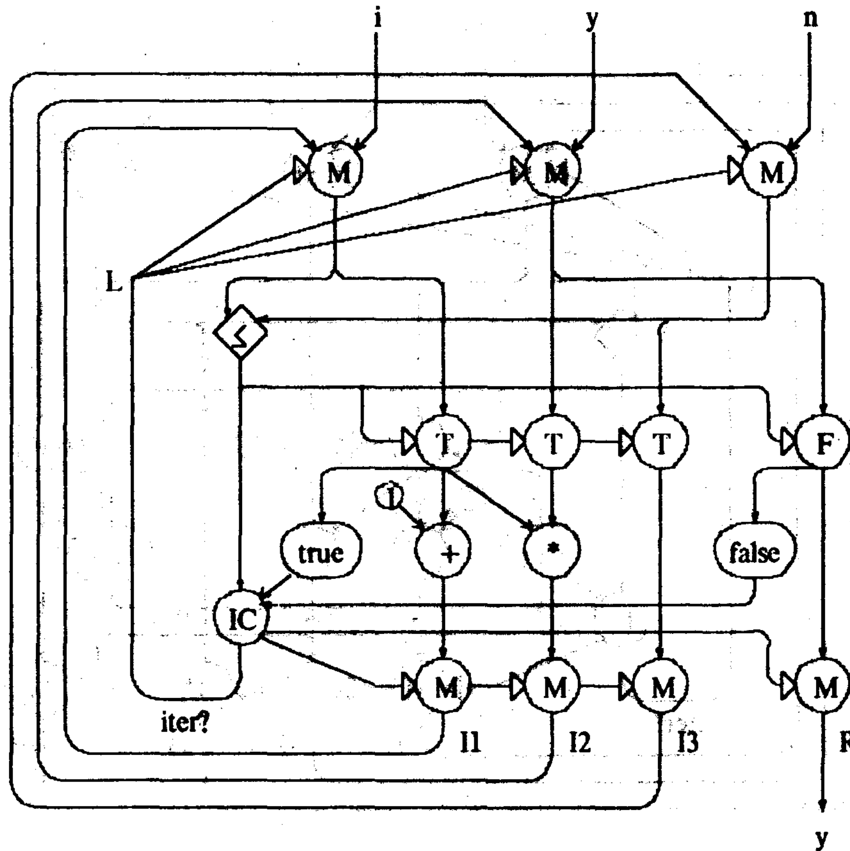
```
for i, y = 1, 1 do
        if i ≤ n then iter i+1, y*i else y end
end
```

The data flow graph representation of this program is shown in Figure 4.10. The graph is composed of an iteration construct whose *iterbody* is a simplified form of the conditional iteration expression shown in Figure 4.8. The simplification occurs since only the **then** clause of the conditional iteration body will actually iterate values. Though both branches have the ability to iterate and return values, the tail recursive structure of the algorithm causes values to be iterated through one branch and returned through the other.

If a set of rules existed for each VAL construct, determining which acknowledge arcs to remove for the factorial data flow graph would begin with analysis of the inner conditional iteration body. However, since we have only developed rules for the conditional and iteration constructs, we must leave the conditional iteration body as is, and proceed to the surrounding iteration construct.

Clearly, the acknowledge arc between the IC gate and operator L can be removed. Rule T1 governs the arc pairs between L and the M gates. The **i** and **n** data values must be used in producing the *iter?* control value; therefore, only the acknowledge arcs of the arc pairs between L and the M gates controlling the **i** and **n** data values may be removed. I1, I2, and I3 (iteration) arc pairs satisfy the first condition of rule T2; a successive value cannot be produced on the I output arc until the corresponding input value on the corresponding input arc has been absorbed. Thus, none of these needs an acknowledge arc. Finally, we examine the v_i arc pairs, which in the Figure 4.10 graph represent all six arc pairs emanating from the three M gates controlling the **i**, **y** and **n** data values. According to rule T3,

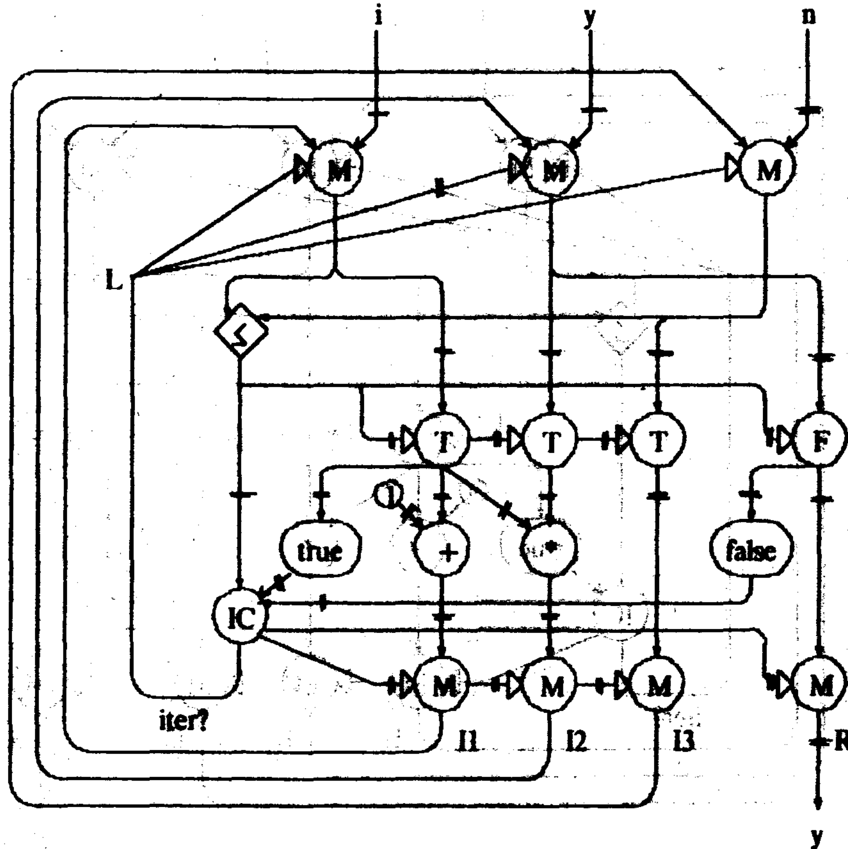
Figure 4.10. Data flow graph of the factorial algorithm



only the two arc pairs input to the predicate of the conditional iteration body can have their acknowledge arcs removed. The other four are input to T and F gates, making their acknowledge arcs essential. The results of this analysis are shown in Figure 4.11 where each arc requiring an acknowledge arc has been marked with a double bar, ||; those not marked are assumed to be single data arcs.

While the factorial data flow graph shown in Figure 4.10 is produced by the T algorithm, the simplified form of the conditional iteration body is significant in that the M gates which merge iteration and return values of the construct, though present, serve no function. The temptation is to optimize the graph by removing these M gates as well as the IC gate I and R control outputs. Though possible, rule T2 must be reevaluated as a direct consequence of this action since the analysis used to formulate rule T2 relies on the standard form of the conditional iteration body shown in Figure 4.8. Specifically, the

Figure 4.11. Optimized factorial data flow graph



reasoning behind case (2) of rule T2 is dependent on the presence of the I and R M gates. We state rule T2 and proceed to reexamine each of its cases.

T2: The acknowledge arc of an I arc pair can be removed if either:

> (1) The iteration body cannot emit a value on that output arc until it has absorbed the corresponding input value on the corresponding input arc.
>
> (2) The *iter?* value depends on the corresponding input arc.

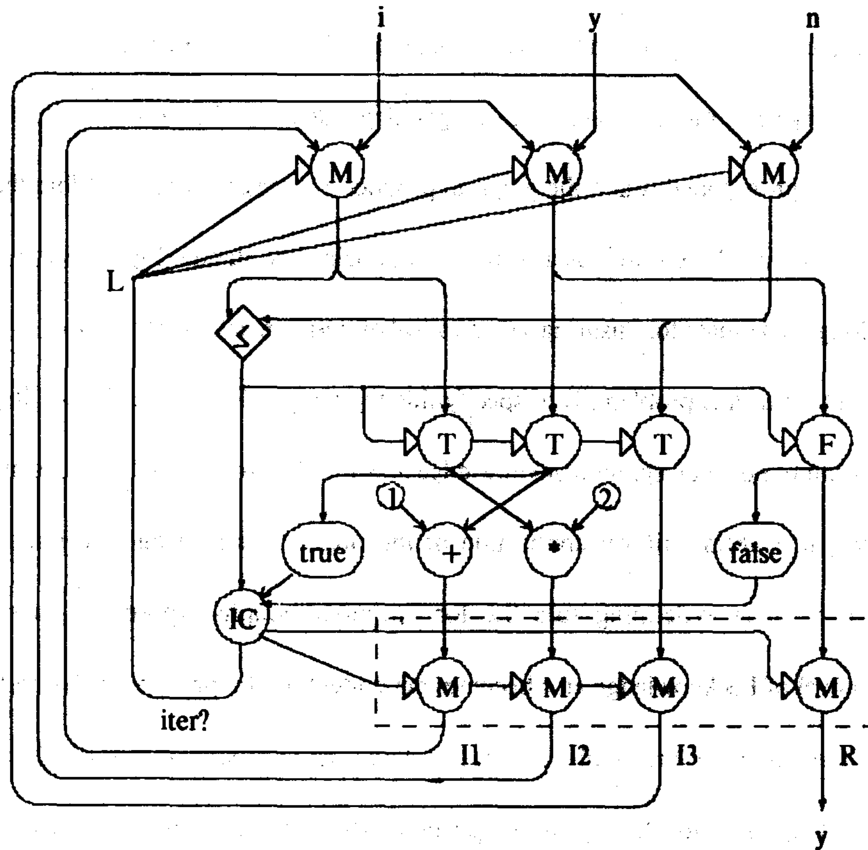
Condition (1) of this rule still applies, since it describes the situation in which each successive iteration value is a function of its previous value. Clearly, only one value can appear on an arc which satisfies this condition at any time. Removing the M gates does not affect this case. To reevaluate case

(2) of rule T2, we focus on the data flow graph shown in Figure 4.12, the representation of the VAL program:

```
for i, y = 1, 1 do
        if i ≤ n then iter y+1, i+2 else y end
end
```

This graph, similar in structure to the factorial graph, displays the same M gate phenomenon, but is significant in its reassignment of iteration variables. Each of these two variables is a function of the other: Iteration variable i is a function of y, and iteration variable y is a function of i.

Figure 4.12. Example data flow program



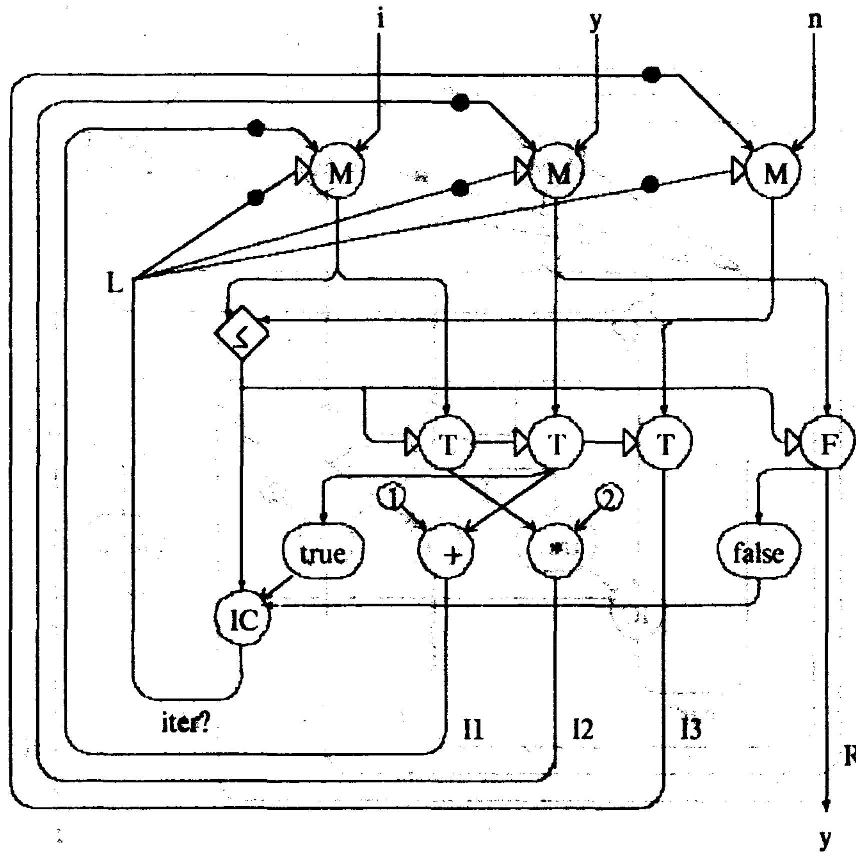
Iteration arcs of the factorial data flow graph satisfied case (1) of rule T2 -- dependence of a successive value on its previous value, allowing their acknowledge arcs to be removed. Case (1) does not apply to the I1 and I2 arc pairs in the graph in Figure 4.12 due to the "crossover" reassignment of iteration variables. However, their acknowledge arcs can be removed since case (2) of rule T2 is satisfied: Production of the *iter?* value depends on both i and y. Variable i is needed to compute the IC gate control input, and variable y generates the gate's true data input.

The structure of the Figure 4.12 data flow graph enables us to examine whether case (2) of rule T2 correctly determines acknowledge arc removal if the graph is optimized by removing its I and R M gates and IC output control arcs (portion of the graph shown in the dashed box). Consider the state of the graph shown in Figure 4.13, the optimized version of the Figure 4.12 graph.

It is now possible for a sequence of operator firings to place a successive value on I2, resulting in the unsafe state shown in Figure 4.14. Even though the IC gate is dependent on the y value, the production of successive iteration values is no longer dependent on the *prior* firing of the IC gate. Thus, the i value can propagate through the graph to produce a successive y value before the previous y value has been absorbed. We see that as a result of optimizing the standard graph form, the case (2) condition is no longer adequate for ensuring safe removal of iteration acknowledge arcs.

One approach to this problem, is to specify this type of graph optimization as illegal. Such a restriction favors the removal of iteration acknowledge arcs over the removal of unnecessary operators. At the same time, it enables uniform application of the present acknowledge arc removal rule. A second approach involves redefining rule T2 for optimized graphs whose M gates have been eliminated. Removal of I acknowledge arcs becomes dependent on the predicate value rather than the *iter?* value. The functioning of the graph dictates that data used in producing I or R values must come through the T or F gates controlled by the graph predicate. This ensures that M gates controlling

Figure 4.13. Modified data flow program from Figure 4.12



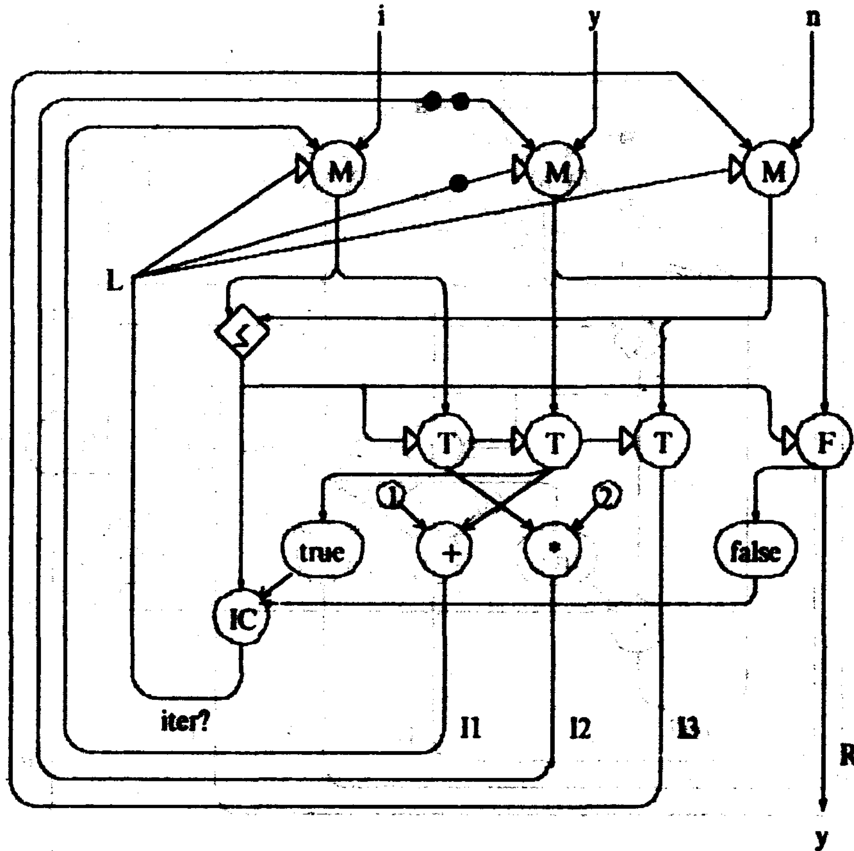
variables used in computing the predicate must fire before new iteration values can be produced. The modified version of rule T2, case (2) reflects this analysis by specifying that an iteration acknowledge arc may be removed if its corresponding input arc must be used in producing the predicate value.

T2: The acknowledge arc of an I arc pair can be removed if

(1) The iteration body cannot emit a value on that output arc until it has absorbed the corresponding input value on the corresponding input arc.

(2) the predicate output value depends on the corresponding input arc.

Figure 4.14. Unsafe token configuration for Figure 4.13



Using this rule, the acknowledge arc of iteration arc pair I2 can *not* be removed since computation of the predicate value does not involve y, the variable controlled by its corresponding input arc.

This analysis of the factorial algorithm emphasizes the options and problems which quickly surface in considering rather basic examples. The acknowledge arc removal rules, while adequate for graph configurations derived by straightforwardly applying the T algorithm, could require significant expansion to be compatibly used with other optimizations. A study of more complex graphs or of those requiring this optimization in conjunction with other optimizations would be useful in determining the general applicability of these rules, and is designated as an area of interest for future research.

CHAPTER FIVE

5.1 Summary

The aim of this thesis has been to address problems which arise in translating a high level language for a machine architecture designed for parallel processing. While the high level language is nearly indistinguishable from source languages for standard sequential processors, the *data driven* execution of its instructions requires a radically different form of translation. This study of *data flow translation* uses the high level language VAL and the Dennis-Misunas architecture. While standard methods of data flow processing do not yet exist, the model used reflects the type of translation issues to be tackled in the realm of data flow. The problems unveiled and solutions proposed are illustrated using data flow graphs, which result from applying the T translation algorithm to VAL programs. Though these data flow graphs closely correspond to the machine language representation of VAL programs, their level of abstraction and explicit representation of data dependencies make them a generally accepted model of data flow.

Chapter 2 focuses on the firing behavior of data flow graph operators which must ensure a maximum capacity of one value per arc as dictated by the Dennis-Misunas architecture. While restrictions of other data flow architectures may be less severe, the need to place some finite limit on arc capacity is common to most. The transformation of arcs within data flow graphs to data/acknowledge arc pairs is introduced as a means of implementing the desired operator behavior. A formal argument establishes that the *safe* operation resulting from the transformation is guaranteed, and that the *liveness* and functionality of the graph is not altered. The use of data/acknowledge arc pairs does however have a profound effect on operator firing sequences within a given graph, and therefore on its throughput. The remainder of the thesis explores the consequences of incorporating d/a arc pairs and suggests

methods of modifying the transformation algorithm to improve graph performance.

Though safe operation is achieved by preventing any given operator from firing until appropriate acknowledges are received, the delayed firing of an operator may cause a subsequent and unnecessary delay to operators dependent on its output. This phenomenon is the subject of chapter 3. The algorithm developed in this chapter eliminates potential bottlenecks within a graph by buffering arcs with identity operators so that all paths through the graph are an equal length. Analyses of performance show that this approach maximizes throughput, but at a potentially high cost in terms of identity operations. While performance statistics indicate that this latter strategy is promising, the choice of an optimum buffering scheme is complicated by the number of interacting factors.

A second approach for optimizing a transformed data flow graph, which aims to decrease overhead by eliminating unneeded acknowledge arcs, is discussed in chapter 4. By identifying situations in which particular arcs do not depend on an acknowledgement to prevent multiple token occurrances, the number of acknowledge arcs can be minimized. This is accomplished by analyzing the data flow graph implementation of each VAL construct to find arc pairs that may be subject to acknowledge arc removal, and specifying rules which enable these situations to be recognized. The chapter concludes with several examples illustrating this optimization. While the techniques of balancing token flow and removing unnecessary acknowledge arcs have been developed independently, the optimum configuration for any given data flow graph is reached by application of both optimizations. The absence of specific information about hardware (e.g., operator execution times, etc.), prevents the development of an algorithm combining the two at this time; however, an attempt is made to identify the major factors contributing to the choice of optimizations. These issues developed in chapters 3 and 4 should prove applicable to translation and optimization problems arising in other data flow models.

5.2 Directions for Future Research

Three areas of research are natural extensions of the work presented. The first focuses on further development of the chapter 4 optimization. The work presented analyzed the VAL *conditional* and *iteration* constructs to determine the circumstances under which certain arc pairs could safely function without an acknowledge arc. A more extensive study of data flow graphs containing these constructs would be useful in determining the completeness of the rules presented. Certain graph configurations may reveal additional cases to test for in removing acknowledge arcs, thus leading to an extension of the proposed rules. A more straightforward task involves application of the chapter 4 analysis to the remaining VAL constructs. This work is required for the development of a recursive algorithm which could perform acknowledge arc removal for the data flow graph representation of a program.

A second avenue of research centers on performance evaluation of data flow graphs. As data flow computer prototypes become available, the type of performance analysis shown in chapter 3 should produce more accurate data. Statistical studies can be made of token flow patterns for various graph configurations, and corresponding optimization schemes. Information gathered should determine when or whether the benefits of an optimized graph outweigh the cost incurred. A study of different configurations of a single data flow graph should provide valuable data on optimization tradeoffs. This would contribute invaluable information toward formulating an algorithm integrating the optimizations of chapters 3 and 4.

Finally, the research can be extended to include more traditional optimization techniques. This would initially require a determination of which of these optimization strategies are applicable and adaptable to data flow. While redefining optimizations such as *strength reduction* seems possible and fairly straightforward, the adaptation of other traditional optimizations to a parallel processing

context may require a different set of considerations. A data flow version of these optimizations could depend on the development of certain tools, such as a categorization of equivalent graph configurations. A comprehensive examination of the application and meaning of such traditional optimizations in data flow remains. The potential in following this route, and of further developing optimizations particular to data flow computation is just beginning to be tapped. The extensive history of sequential programming optimization techniques will no doubt have its counterpart in the world of data flow.

BIBLIOGRAPHY

[1] Ackerman, W. B., "Interconnections of Determinate Systems", Computation Structures Group (Note 31), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1977.

[2] Ackerman, W. B., and J. B. Dennis, *VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual*, Laboratory for Computer Science (TR-218), MIT, Cambridge, Massachusetts, June 1979.

[3] Brock, J. D., *Operational Semantics of a Data Flow Language*, Laboratory for Computer Science (TM-120), MIT, Cambridge, Massachusetts, December 1978.

[4] Brock, J. D., and L. B. Montz, "Translation and Optimization of Data Flow Programs", *Proceedings of the 1979 International Conference on Parallel Processing* (O. N. Garcia, Ed.), August 1979, 46-54. Also, Computation Structures Group (Memo 181), Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

[5] Chamberlin, D. D., "The 'Single-Assignment' Approach to Parallel Processing", *AFIPS Conference Proceedings 39, 1971 Fall Joint Computer Conference*, November 1971, 263-269.

[6] Commoner, F., "Deadlocks in Petri Nets", *Research Report of Applied Data Research*, Inc., Lakeside Office Park, Wakefield, Mass., June 1972.

[7] Commoner, F., S. Even, A. W. Holt, and A. Priveli, "Marked Direct Graphs", *Journal of Computer and System Sciences* 5, October 1975, 511-523.

[8] Denning, P. J, "On the Determinacy of Schemata", *Record of the Project MAC Conference on Concurrent System and Parallel Computation*, ACM, New York, 1970, 143-147.

[9] Dennis, J. B., "First Version of a Data Flow Procedure Language", *Programming Symposium: Proceedings, Colloque sur la Programmation* (B. Robinet, Ed.), *Lecture Notes in Computer Science 19*, 1974, 362-376. Also, Laboratory for Computer Science (TM-61), MIT, Cambridge, Massachusetts.

[10] Dennis, J. B., "A Language Design for Structured Concurrency", *Design and Implementation of Programming Languages: Proceedings of a DoD Sponsored Workshop* (J. H. Williams and D. A. Fisher, Eds.), *Lecture Notes in Computer Science 54*, October 1976. Also, Computation Structures Group (Note 28-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

[11] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", *The Second Annual Symposium on Computer Architecture: Conference Proceedings*, January 1975, 126-132. Also, Computation Structures Group (Memo 102), Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

[12] Dennis, J. B., D. P. Misunas, and C. K. C. Leung, "A Highly Parallel Processor Using a Data Flow Machine Language", Computation Structures Group (Memo 134-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, June 1979. To appear in *IEEE Transactions on Computers*.

[13] Dennis, J. B., and K.-S. Weng, "An Abstract Implementation for Concurrent Computation with Streams", *Proceedings of the 1979 International Conference on Parallel Processing* (O. N. Garcia, Ed.), August 1979, 35-45. Also, Computation Structures Group (Memo 180), Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

[14] Hack, M., *Analysis of Production Schemata by Petri Nets*, Laboratory for Computer Science (TR-94), MIT, Cambridge, Massachusetts, February 1972.

[15] Holt, A. W., *Final Report of the Informations System Theory Project*, Technical Report RADS-TR-68-305, Rome Air Development Center, Griffiss Air Force Base, New York, 1968.

[16] Holt, A. W., and F. Commoner, "Events and Conditions", *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, ACM, New York, 1970, 3-52.

[17] Jaffe, J. M., and L. Montz, "Two Data Flow Solutions of Laplace's Equation", Computation Structures Group (Note 37), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1978.

[18] Karp, R. M., and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", *SIAM Journal of Applied Mathematics 14*, November 1966, 1390-1411.

[19] Kessels, J. L. W., "Parallel Programming Concepts in a Definitional Language", *SIGPLAN Notices*, 11(10), October 1976.

[20] Kosinski, P. R., *A Data Flow Programming Language*, IBM T. J. Watson Research Center (RC 4264), Yorktown Heights, New York, March 1973.

[21] Kosinski, P. R., "A Data Flow Language for Operating Systems Programming", *Proceedings of ACM SIGPLAN-SIGOPS Interface Meetings, SIGPLAN Notices 8*, 9(September 1973), 89-94.

[22] Leung, C. K. C., *Formal Properties of Well-Formed Data Flow Schemas*, Laboratory for Computer Science (TM-66), MIT, Cambridge, Massachusetts, June 1975.

[23] Leung, C. K. C., D. P. Misunas, A. Neczwid, and J. B. Dennis, "A Computer Simulation Facility for Packet Communication Architecture", *The Third Annual Symposium on Computer Architecture, Computer Architecture News 4*, 4(January 1979), 58-63. Also, Computation Structures Group (Memo 127-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

[24] Misunas, D. P., "Deadlock Avoidance in Data-Flow Architecture", *Proceedings of the Third Milwaukee Symposium on Automatic Computation and Control*, April 1975. Also, Computation Structures Group (Memo 116), Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

[25] Patil, S. S., "Closure Properties of Interconnections of Determinate Systems", *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 1970, 107-116.

[26] Petri, C. A., *Communication with Automata*, Supplement 1 to Technical Report RADC-TR-65-377, Vol. 1, Griffiss Air Force Base, New York 1966. [Originally published in German: Kommunikation mit Automaten, University of Bonn, 1962.]

[27] Ramchandani, C., *Analysis of Asynchronous Concurrent Systems by Petri Nets*, Laboratory for Computer Science (TR-120), MIT, Cambridge, Massachusetts, February 1974.

[28] Tesler, L. G., and H. J. Enea, "A Language Design for Concurrent Processes", *Proceedings of the AFIPS Conference 32*, 1968, 403-408.

[29] Weng, K.-S., *Stream-Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science (TM-68), MIT, Cambridge, Massachusetts, October 1975.