1978

Compile time prediction of intrinsic parallelism in high level programs

C. Retnadhas

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
Retnadhas, C., "Compile time prediction of intrinsic parallelism in high level programs" (1978). Retrospective Theses and Dissertations. 6415.
https://lib.dr.iastate.edu/rtd/6415

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.
RETNADHAS, C.
COMPILE TIME PREDICTION OF INTRINSIC PARALLELISM IN HIGH LEVEL PROGRAMS.
IOWA STATE UNIVERSITY, PH.D., 1978
Compile time prediction of intrinsic parallelism in high level programs

by

C. Retnadhas

A Dissertation Submitted to the Graduate Faculty in Partial Fulfillment of The Requirements for the Degree of DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Department

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1978
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER I. INTRODUCTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>2</td>
</tr>
<tr>
<td>Underlying Data Flow Model</td>
<td>6</td>
</tr>
<tr>
<td>Outline of Thesis</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER II. MEASUREMENT OF INTRINSIC PARALLELISM AND RESOURCE REQUIREMENTS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>17</td>
</tr>
<tr>
<td>High Level Target Language</td>
<td>17</td>
</tr>
<tr>
<td>Overview of Data Flow Processing</td>
<td>18</td>
</tr>
<tr>
<td>The Simulation</td>
<td>25</td>
</tr>
<tr>
<td>Data for Simulation</td>
<td>33</td>
</tr>
<tr>
<td>Transformations</td>
<td>35</td>
</tr>
<tr>
<td>Simulation Results</td>
<td>38</td>
</tr>
<tr>
<td>Conclusions</td>
<td>59</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER III. ANALYSIS OF PROGRAM GRAPH</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>62</td>
</tr>
<tr>
<td>Definitions and Notations</td>
<td>63</td>
</tr>
<tr>
<td>Background Material</td>
<td>67</td>
</tr>
<tr>
<td>Acyclic Directed Graph for Model I Architecture</td>
<td>69</td>
</tr>
<tr>
<td>Graph Representation for Model II Architecture</td>
<td>99</td>
</tr>
<tr>
<td>Stream Read and Program Structure</td>
<td>110</td>
</tr>
<tr>
<td>Sources of Error in Approximate Timed Program Graph</td>
<td>113</td>
</tr>
<tr>
<td>Conclusion</td>
<td>122</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER IV. INTEGRATION OF PROGRAM ANALYSIS IN COMPILER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>123</td>
</tr>
<tr>
<td>Overview of Compiler</td>
<td>124</td>
</tr>
<tr>
<td>Compiler Generated Tables</td>
<td>125</td>
</tr>
<tr>
<td>High Level Construct, Execution Time and Program Graph</td>
<td>133</td>
</tr>
</tbody>
</table>
iii

Generation of Timed Program Graph 159
Example 164
CHAPTER V. CONCLUSIONS 181
REFERENCES 184
ACKNOWLEDGMENTS 186b
APPENDIX 187
Resource Requirements 187
CHAPTER I. INTRODUCTION

The unparalleled growth of data processing has opened up new avenues of applications requiring a higher processing rate. This demand for a higher processing rate can be accomplished by either increasing component speed or by using an efficient system organization. The component speed is limited by technological barriers and the availability of cheap and faster system components. The concept of multiprocessing has become an attractive and viable alternative to meet the demand for faster execution speed.

For a given level of technology, an optimal system organization is one which is well-balanced to match exactly the process. Even in a well-balanced system, the inherent limitations of the system components inhibit an exact match between the system components and processes. An alternative to this dilemma is to let the processes be small enough to match the system components to obtain a better fit.

A number of parallel processing processors [1, 18, 37] have been developed to increase the processing rate by exploiting the inherent parallelism present in computational processes. Lookahead processors obtain speedup in execution by detecting concurrency in a subset of the instruction stream, however, they fail to exploit all the intrinsic parallelism present in the program. Certain parallel processors demand special processing structures to realize the system potential
to its full extent. In general, parallel processing as it exists today refers to concurrent processing of several processes which are themselves executed strictly sequentially. This mode of processing enhances the potential for deadlock, race conditions and processor and memory switching problems.

An alternative approach that can avoid these inefficiencies appears to be a processor design based on data driven sequencing of computations [2, 8, 27, 32]. In this mode of processing, the computations subdivide in a natural way to span the processing capabilities at an operational level and are executed as soon as their operands are available, regardless of their position in the instruction set. The operations are independent of execution speed, thereby introducing the possibility of using a large number of slow processors to achieve a cost effective implementation of the data driven processing capability. Since the data flow processors are driven by the inherent parallelism in the high level program, they show promise of achieving a significant cost performance advantage over von Neuman-like processors in many areas of application.

Summary

The concept of data flow processing, where instructions are executed as soon as data is available, allows a higher processing rate by utilizing the intrinsic concurrency in computational processes. This principle has been used to
demonstrate a large potential gain in execution speed for
certain applications [8] when executed on a data flow inter-
preter. In this thesis, an attempt is made to study the impact
of data flow processing in general purpose computing appli-
cations compared to sequential von Neuman-like processing.

Intrinsic parallelism and the cost of concurrent proc-
essing are used as measures in evaluating the data flow archi-
tecture. A software simulation was used to obtain the measures
because no hardware implementation of the feedback data flow
processor exists at this point in time.

To capture such measurements, high level programs are
translated to a data flow language and executed on the simu-
lator. The parallel execution time (the execution time on a
data flow processor) and the sequential execution time (the
execution time on a sequential processor with equivalent proc-
essing capability) are obtained by assigning execution times
to the data flow instructions. The ratio of sequential to
parallel execution time for the entire program is defined as
the intrinsic parallelism in the high level program. The cost
of concurrent processing, which consists of the data path,
memory, and processor requirements along with the overhead re-
quired to maintain determinate and deadlock free computation,
is obtained for minimum parallel execution time. Resource
contention is avoided by providing more than enough functional
units of each type. Numerical measurements of parallelism and resource requirements are reported for a set of computer programs. The suitability of forward substitution transformation for programs to be executed on a feedback data flow processor is studied by measuring the decrease in parallel execution time and the amount of redundancy introduced. The inhibiting effect of sequential file operation on parallel execution time is illustrated by obtaining the parallel execution time without the I/O effect.

The simulation method is invaluable in evaluating system performance accurately at a step close to the real system. But the cost of simulation increases with the program size, thereby limiting the study to a set of programs which are small both in static size and in number of instructions executed. Even though the numerical results are fairly accurate estimations of real world performance, the result do not reveal explicitly the cause and effect relationship between the program characteristics and the resulting execution speedup and resource requirements.

The timed program model is introduced as an alternate method to obtain the system performance. Its attractiveness is its ability to model programs with a large number of statements. The graph model explicitly shows the relationship between the program structure and the performance characteristics. This graph model provides estimates of the intrinsic parallelism without
requiring an actual execution of the program. Also, the estimated intrinsic parallelism in the high level program for various augmenting architectural features at a high level can be obtained with modest additional effort and less cost using the graph model.

In the graph model, called the timed program graph, nodes represent expressions, assignment statements, procedure invocations and input/output processing implied by individual components in the I/O lists as they appear in the high level program. The arcs represent the data dependencies among the nodes. The timed program graph model is an approximation of the data flow graph representation of the program. For a given semantic model of an architecture, the timed program graph for each of the high level constructs and its execution time are formed. The characteristic timing equations, consisting of the sequential and parallel execution times, are derived from the timed program graph for the entire program. An estimate of intrinsic parallelism is obtained from the characteristic equations. The ease with which a timed program graph can be used to obtain the execution speedup possible in high level programs using an architecture including a vector processing capability is illustrated. The ability of the timed program graph model to estimate performance characteristics such as production rate is demonstrated by including a stream input feature in the high level program. This feature makes the program structure behave
like a pipeline for consecutive sets of input data. The graph model is validated by comparing the measure of intrinsic parallelism obtained by simulated execution to the model estimate. Finally, the sources of error introduced in the graph model are identified. The techniques to implement the timed program on a existing compiler which generates the data flow code for a high level language is discussed. A sample timed program graph formed from the compiler generated tables is also presented.

Underlying Data Flow Model

The intrinsic parallelism in programs is measured by translating the high level program to data flow language and executing it on a simulated data flow processor. Hence, the data flow processor and the data flow language form the basis for the work reported in this thesis. Even though intrinsic parallelism can be measured using a different computer organization, the data driven processor and language are selected as tools here. Therefore, the data flow language and the data flow processor is described in this section. The data flow language [5] and processor developed by Dennis and Misunas [7] were selected as the base machine over other data flow architectures [2, 27, 32] because the language is powerful enough to express algebraic computations and the processor is well-defined.

Data flow language

The attractiveness of the data flow language as a semantic support for concurrent processing is its ability to reveal
concurrency at the operation level, to provide the natural control flow based on the availability of data and the presence of determinate interaction between modules. The data flow language is a directed graph where the nodes represent operations (actors) and the arcs (links) can be considered as a channel to transfer tokens (data values). The arcs may carry control or data tokens.

A node representing an operation is enabled only when all its operands are available. An enabled operator can produce a result token by absorbing the input token(s) only if no token is present in any of the output arcs. This process of producing a result token after a certain amount of elapsed time is called the firing of the actor. All the operators with the exception of the merge operator require all operands to be enabled. The merge operator selects a token from one of its two input arcs depending on the type of control token present on the control arc. The token not affected by the operation remains undisturbed.

The operators and the conditions under which an operator can produce a result token are shown in Figure 1.1. These operations can be divided into three groups. The first group of operators manipulate data values and they are represented as square boxes in the data flow program. The second group consists of a set of operators which selectively advance data values to successor operators depending upon the value of the
control token. For example, a true control token on the control arc of the T-gate in Figure 1.1b passes the data token on its input arc and a false control token absorbs the data token on the input arc. The merge control operator advances the token on the false side if the control token is false, and on the true side if the control token is true. The third group of operators correspond to the predicate operators in high level languages. The apply operator provides the procedure application support and requires the program text and an input structure consisting of the actual parameters. The output token is a structure consisting of the results of procedure application.

Figure 1.1. Data Flow Operators and Their Firing Rules.
structure operations

a. Group 1 Operators

T - Gate

F - Gate

Figure 1.1 (Continued)
A segment of high level program which computes the sum of two vectors is used to illustrate the data flow language. Figure 1.2 represents the high level program and the data flow graph taken from [5]. Assuming that the values of \( a, b, \) and \( n \) are available on the input arcs of the merge gate, the false control token on the control arc of the merge operator, lets the values go through the false side of the merge for the first
time. The firing of merges enable the decider. Since the output arcs of the decider do not have tokens present on them, the decider can fire, thereby sending control tokens to all its successor operators. The arrival of a true control token enables and fires the plus, the merge for \( n \) and the two select operators. The merge for \( a \) and the merge for \( b \), even though enabled cannot fire because the output arcs contain tokens from the previous firing. Two select operators fire, thereby selecting \( a(1) \) and \( b(1) \). At this point in time, the decider is enabled for the second time, but the merge for the array \( c \) has a control token from the previous firing sitting on its input arc, which is one of the output arcs of the decider. Therefore, the decider cannot fire for a second time until this merge fires again. The append operator fires after the plus operator producing the value \( c(1) \). This sends a token to the merge for array \( c \) and forces the merge to consume the control token. The firing of merge allows the decider to fire for the second time. This sequence repeats until the decider produces a false control token. The arrival of false control tokens initializes the merge operators to receive the next set of input data.

```plaintext
input (a, b, n)
    i := i;
    while i <= n do
```
begin
    c(i) := b(i) + a(i);
    i := i + 1;
end

output c

a. Segment of High Level Program for Vector Addition

Figure 1.2. A Segment of High Level Program and the Data Flow Program to Add Two Vectors
The data flow simulator presented in Chapter II executes an extended version of the data flow language. Even though the data flow language is powerful enough to represent algebraic computations, it may be difficult to program directly at this level. There has been considerable interest in developing high level data flow representations which expose intrinsic parallelism in computations in a deterministic way. This calls for a language whose semantic capabilities imply data driven sequencing and concurrent execution without introducing non-determinancy. The concept of single assignment [5, 27, 36, 38] allows concurrency to be expressed at a higher level compared to the data flow language presented above.

**Data flow processor**

The data flow simulator described in Chapter II is an extension of the basic data flow processor proposed by Dennis [7]. A brief description of the basic data flow processor is presented. The organization of the basic data flow processor is shown in Figure 1.3, which is taken directly from [7]. The instruction cells in the memory contain the encoded data flow program. The instruction cell consists of registers to contain the opcode of the operation to be performed and the operands for the operation. The operators T-gate and F-gate in the data flow program are made part of the operand register, thereby eliminating the need for a separate operator in the data flow processor. An instruction cell is enabled when all the
operands, including the gate values (if applicable), are present. The enabled instructions are forwarded to the arbitration network which guides the instruction packet formed from the enabled instruction cell to the appropriate operation or decision unit. The operation unit operates on the operand(s) to form a result packet and forwards the result packet to the

Figure 1.3. Organization of a Basic Data Flow Processor.
distribution network. The distribution network stores the result value in the operand register of the instruction cell. The arrival of result value may enable other instruction cells in the memory. If the instruction packet was forwarded to the decision unit, then the control network distributes the control tokens to instruction cells. The presence of a control token may enable an instruction cell if the gate value matches the control token. If the control token and the gate value do not match, then the operand value is discarded and the operand register goes back to its initial state, thus performing the function of the F-gate or T-gate in the instruction cell itself.

Outline of Thesis

The intrinsic parallelism in computer programs, as exploitable by a feedback data flow processor, and the corresponding resource requirements is studied in this thesis. The source of parallelism, normally exploitable by the concept of streams in a data flow program, is conservatively estimated in a graph analysis of the program by assuming the availability of conventional vector functional units.

The material is organized as follows: In Chapter II the simulation study of intrinsic parallelism in high level programs and the resource requirements for executing the programs are presented. Measurements of the effect of forward substitution and the sequential file operation are also presented. In Chapter III the timed program graph model is described as an
alternate way to capture program measurements. Two underlying architectures are considered. An attempt is made to validate the model by comparing the numerical results of simulation with numeric results derived from the timed program graph. Chapter IV gives details for generating the timed program graph at compile time, thereby automating much of the required analysis. This is followed by Chapter V which contains the conclusions.

Many persons have been involved in the developmental efforts needed to make possible the work presented in this thesis. In particular, the data flow simulator, described in Chapter II, is the work of several people. I was responsible only for the development of the data memory component and the instrumentation of the simulator to capture measurements. The high level language, described in Chapter II, and its intermediate form along with implied translation and flow analysis routines, described in Chapter IV, is entirely the work of other individuals. Wherever necessary, brief descriptions are presented in this thesis for the sake of completeness. Details of the data flow simulation and compiler can be found in [26]. In addition, many ideas for the graph model of computer programs were the result of group discussions with several individuals. Details of compile time generation of program graphs and the associated timing equations are my work and a culmination of my participation in the other mentioned areas of development.
CHAPTER II. MEASUREMENT OF INTRINSIC PARALLELISM AND RESOURCE REQUIREMENTS

Introduction

In this chapter, an attempt is made to evaluate the impact of the data-driven approach on the execution of existing computer programs using simulation. A brief description of the high level language, to which all the programs are transformed, is presented. Then, a conceptual overview of the data flow language and a highly instrumented software simulator are described. Finally, the simulation results for a set of real computer programs are given. These results represent estimates of intrinsic parallelism and resource requirements for the executed programs, the optimizing effect of forward substitution in the high level program, and the inhibiting effect of sequential file operations in a parallel environment.

High Level Target Language

The selected real programs, from existing sources, are mapped to the high level target language before being translated to the data flow language. The target high level language has expressive power similar to Algol 60 and contains basic features which are considered desirable and are expected to remain in future programming languages.

Initially, however, the number of data types have been limited to real, integer, boolean and array. A limited input/output formatting capability is provided along with implied
iterative input/output. The language is procedure oriented and eliminates side-effects in procedures and functions by allowing changes through the parameter list only. Also, by providing the feature of directionality attribute for both actual and formal parameters, the data flow analysis is simplified to some extent. All the control constructs are forced to have a single entry and exit. Features such as go to statement, labels, global references outside a procedure, strings and string operators, procedure variables, case statements, generalized structures and pointers are not supported at present. Table 2.1 summarizes the features supported at present. Data flow analysis is somewhat more complicated, as it is not a single assignment language. This section is intended merely as an overview of the main features supported by the language; an in-depth study of data flow analysis and code generation is presented in [26].

Overview of Data Flow Processing

The target language compiler produces the data flow code to be executed on a feedback data flow interpreter. The data flow language used and the underlying architecture on which it is executed are a variation and an extension of the interpreter specified by Dennis [5], Dennis and Misunas [7]. This feedback data flow interpreter is capable of operations on scalar values only, and does not have the computational power of the unrolling interpreter [2] or vector processors [16]. The feedback data
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Blocks</td>
<td>The language is a static block structured language.</td>
</tr>
<tr>
<td>2. Procedures and functions</td>
<td>The language is procedure oriented. Procedures (functions) may be defined in the declarations of any block. Procedures (functions) have no side effects.</td>
</tr>
<tr>
<td>3. Parameters</td>
<td>All parameters, both formal and actual must carry a directionality attribute (input, output or input-output).</td>
</tr>
<tr>
<td>4. Control Structures</td>
<td>Control structures supported are: if-then, if-then-else, while-do, repeat-until.</td>
</tr>
<tr>
<td>5. Input/Output</td>
<td>Input/Output formats supported are I,F,E,X,SKIP and PAGE. A restricted implied do-loop is supported. Only sequential files are supported.</td>
</tr>
<tr>
<td>6. Simple variable types</td>
<td>Variables may be real, integer or boolean.</td>
</tr>
<tr>
<td>7. Arrays</td>
<td>Arrays can be of type integer, real or boolean. Array bounds can be specified by the programmer or dynamically declared at run time.</td>
</tr>
<tr>
<td>8. Operators</td>
<td>Operators supported are - arithmetic: +,-,*,/,** - boolean:¬,¬ - relational: =,&gt;,&lt;,&gt;=,&lt;=,=.</td>
</tr>
<tr>
<td>9. Functions</td>
<td>Abs, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, sqrt, log.</td>
</tr>
</tbody>
</table>
flow interpreter instruction set provides direct semantic support for both implicit and explicit operations that are used to write scientific programs in the target high level language.

The data flow program, consisting of operations, may be viewed as a two-way link. Each operator has static forward links to successor instructions that depend on the result of the operation and backward links to the processor instructions that supplied the operand values. When an instruction executes, the operands are consumed and an acknowledgment is sent to each predecessor determined by the backward link. In the feedback interpreter, an operator is enabled for execution if it has been supplied with needed operands from predecessors and it has received acknowledgments from each of its successors.

To prevent more than one source (possible triggered by the next input value or start of the next iteration) from attempting to supply the same operand, due to concurrent execution and prior to consumption of the first, a merge instruction with special firing rules is implemented. This instruction provides a staging area for an operand arriving from one of two possible sources. Therefore, through the use of backward links for acknowledgment, special merge instruction, and structured control in high level languages with enforcement of appropriate rules, computations are determinate and deadlock free.

**Instruction set**

The instruction format and the necessary and sufficient
condition for the instruction to be enabled are given in [7, 26]. Here, the instruction set in the data flow language is presented. The feedback data flow interpreter instruction set is capable enough to support operations that appear in high level algebraic languages and to provide synchronization and staging needed to eliminate deadlock and indeterminate computation. The operations and their semantics are given below.

a. **Arithmetic Operations**

   **Binary:** +, -, *, /, **
   
   **Unary:** -

   The arithmetic processing units will accept real or integer type operands. Mixed mode computation is performed in real mode and a result of type real is produced. If necessary, the result is transformed to match the destination type specified by the destination.

b. **Boolean Operations**

   **Binary:** &,, |,
   
   **Unary:** ¬

   The processing units will accept only boolean operands and produce a boolean or signal result.

c. **Relational Operations**

   **Binary Arithmetic:** <, >, ≤, ≥, =, ≠
   
   **Binary** : exists
   
   **Unary** : elem
The binary arithmetic processors will combine real and/or integer operands or combine boolean operands to produce a boolean or signal result. Exists requires the first operand to be of type structure and the second operand to be of type integer. Exists test for the existence of a particular component of a structure. Elem will accept any type operand and determines if the associated data is elementary.

d. **Constant Operation. Cons**

This is a special purpose binary operation to produce a constant value. The first operand may be of any designated type. The second operand is the constant value to be produced. During execution, the arrival of the first operand enables the instruction and triggers the production of the constant value.

e. **Staging Operations**

Unconditional replication: Identity (ID)

Conditional replication : Merge

If the operand to be replicated has been produced by a predecessor instruction (an address), then it is sent to the specified destinations. If the operand to be replicated is a constant, then the data value is first created in data storage and the address is sent to the specified destinations. A variable operand may be a pointer to any structure or elementary value in data storage.
f. **Structure operations**

Creation: Append

Component selection: Select

The append units form a new structure by adding a substructure to an existing structure using the selector value. The components of structure not altered by the addition of the substructure is shared with the old structure. The selector is type integer and the substructure can be type structure or any one of the defined types. The result is type structure. The select units select the substructure referred to by the given selector. The substructure selected may be of type structure or any one of the defined elementary types.

g. **File operations**

Data transmission: Read, Write

Editing: Readedit, Writedit

The read and readedit operations are defined to support standard formatted input from a card image data file. The read unit removes the required number of characters from the input file and converts to the internal form according to the specified format, then forms a structure whose components are data values just read and the rest of the input file as second and first components, respectively. The readedit unit advances the current position in the input file by the specified control format. The readedit operation requires a file name and a format item as operands and returns the file name.
The write and writedit operations support the formatted output. The write operation requires a file name, a format item, and a value to be printed. The value is converted to the character form using the format item, if necessary, and then added to the output file. The file name is returned as the result to be forwarded to the successor instruction. The writedit operation requires a file name and a format item as operands and returns the file name as a result.

h. **Procedure operation**

Procedure call: Apply

The apply instruction has two operands. The first one is a pointer to the location of the procedure in the program memory. The second is a pointer to the input argument structure. The procedure call cannot be activated until all the input parameters are collected into an argument structure. The apply unit sets up the environment for the procedure activation by setting the base register to point to the environment and establishing the return protocol to forward the result structure to the calling procedure.

A data flow procedure which is not active is deactivated. To monitor the activity of a procedure, an activity count is set up when the environment for the procedure is created. As soon as the activity count reaches zero, the procedure can be deactivated by reclaiming the instruction memory which destroys the environment. Care should be taken in maintaining
the activity count when there is a change of environment, such as returning the result structure.

The Simulation

**Software implementation**

The simulator is a software model of the feedback data flow interpreter to simulate the parallel processing inherent in the data flow processing, and the simulator is intended to be independent of any actual hardware implementation. The clock cycle in the simulator is based on the execution time of the basic data flow operations and the systems capacity is limited by the total number of functional units available.

Figure 2.1 represents the functional description of the simulator. Only the pertinent information necessary to get an overall view of the simulator modules and their interaction is presented here.

**Memory**

The instruction memory module consists of the program memory, instruction memory and a file memory. The program memory is used to hold static copy of the procedures which make up the program. When a procedure is activated by an apply instruction, the apply operation creates an execution environment for this procedure by copying it to instruction memory for execution. When a procedure deactivates, the associated instruction memory space is reclaimed for further use.

The file memory consists of character strings, one for
CL - Check-List
RL - Ready-List
EL - Execution-List
FT - Functional unit Table
ET - Execution Time Table

Figure 2.1. Functional Description of the Simulator
each sequential file known to the program. Formatted input and output statements in the high level language are supported by the read, readedit, write, and writedit operations at the base machine level. These file operations return a modified file as a result, the necessary sequencing of file operations is enforced by data dependency among the operations. The actual movement of data between file memory and external storage is assumed to be performed by a peripheral processing facility.

**Instruction issuer module** The instruction issuer can be considered as a set of processes manipulating a set of tables to select and guide enabled instructions through the simulator modules. The set of tables are Ready-List, Execution-List, Functional Unit Table and Execution Time Table. The Ready-List is a bit map on the instruction memory and indicates the instructions in the instruction memory which are enabled and are waiting to be fetched. An enabled instruction is one which has received all its tokens and acknowledgment signals from its predecessors and successors, respectively. The Execution-List is a linked list of all the enabled instructions in instruction packet form. The instruction packet consists of information relevant to the execution of the instruction. The Functional Unit Table and the Execution Time Table contain the number of functional units of each of the base operations and their execution times, respectively.
The two processes in the Instruction Issuer are List-Former and Decoder. The List-Former forms an instruction packet for each of the enabled instructions indicated by the Ready-List, if functional units are available for their execution. After forming the instruction packet, the List-Former links the instruction packet to the Execution-List. The Execution-List is ordered according to the completion times of the instructions. The List-Former calculates an instruction's completion time by adding the execution time of the instruction from the Execution Time Table, to the system clock time at that moment. As soon as the instruction is fetched from the Instruction Memory, acknowledge signals are sent to the predecessors which supplied the operands. The presence of the instruction packet on the Execution-List has changed the status of this instruction and this is reflected by removing the instruction from the Ready-List, and the number of functional units available for further execution of an instruction of this type is reduced by one in the Functional Unit Table. If no functional unit is available to execute any enabled instruction, it is left in the Ready-List.

The Decoder maintains the system clock by setting its value to the earliest completion time of the instruction on the Execution-List. The Decoder removes all the instructions whose completion time equals the clock time and sends them to the appropriate functional units in the appropriate
functional units in the functional unit module. The forwarding of an instruction packet to the functional unit module signifies the end of instruction execution; therefore, the Decoder removes all instructions sent to the functional unit module and releases the functional units used for execution by incrementing the available functional unit in the Functional Unit Table. The Decoder also decrements the activity count of the active procedures. An activity count of zero indicates that the procedure can be deactivated.

**Functional unit module** The functional unit module represents the processing capability of the feedback data flow interpreter. Each simulated base machine operation is assumed to have multiple functional units and the Functional Unit Table represents this multiplicity of functional units. The functional units fetch operand values from the data memory module and perform the specified operation to form the result. If the type of result produced does not match, a type conversion is performed for certain operations and an error message is produced for other operations. The result formed is stored in the data memory and the location in data memory is forwarded to the update module.

**Update module** The update module consists of the process Enabler and the table Check-List. The Check-List is a bit map representation of all potentially enabled instructions in the instruction memory.
The Enabler process stores the result token produced by the functional unit module in the successor instructions operand field as indicated by the destination fields in the instruction packet. Now, all the potentially enabled instructions on the Check-List are searched and if an instruction is enabled, it is put on the Ready-List by setting the Ready-List bit map corresponding to this instruction and the Enabler also increments the activity count. If the gate-c representing the expected gate value of a gated operand does not match the gate value produced by the conditional, the enabled operand is destroyed by false firing the operand and sending the acknowledge signal to the predecessors. The false firing of an operand requires updating a reference count on some data cells in the data memory.

Data memory module The data memory module consists of storage cells to hold elementary values or structure values and the memory manager process.

The storage cells in the data memory module are conceptually organized as an acyclic graph in which each node is the root of a tree-structured subgraph. Elementary non-pointer data values (real, integer, boolean) appear at the leaves of the graph, while arcs between the nodes represent selectors in the case of structured data. This type of organization facilitates the sharing of values among concurrently executing parts of the data flow program. Using a reference
count at each node of the graph, dynamic deallocation of storage can take place when a node is no longer needed for computation, thereby freeing memory cells for reuse by the computation [6]. The reference count of a node is the total number of operands which point to this node plus the number of predecessor nodes existing in the graph. Therefore, a node is released when its reference count is zero.

The memory manager process performs all data memory operations on behalf of the modules which request operations on data memory cells. Upon request to create a data value, the memory and manager allocates a free storage, stores the data value and returns the location of the data value to the requesting module after suitably updating the reference count of the created data cell. The memory manager also performs the reference count decrement and increment operations when requested by other modules. As a result of a reference count decrement action, if the reference count of any node equals zero, that node is deallocated and is returned to the free pool by the memory manager.

The simulator is initialized through specification of a set of parameters, including the number of functional units of each type and their execution times. The data flow program to be executed is preloaded by the simulator into the program memory. Execution is initiated through a boot-strap apply instruction which loads the main procedure into the instruction cells and sets the Check-List bit for all initially enabled
instructions.

Instrumentation

The simulator is instrumented to record the resource requirements to execute the data flow instructions as the enabled instructions move through the functional modules. The resource requirement for each enabled instruction is recorded when the instruction visits the functional module and requests for the resources. The total resources required by all the enabled instructions executed is the sum of the individual resource requirements. Before updating the clock to initiate the next cycle, summaries of resource requirements are updated. For a given execution of a program, the maximum, the average, and the standard deviation of memory, processor and path utilization are recorded. The sequential time steps and the parallel time to execute the program are also recorded. The statistics collected are:

a. Data path utilization The maximum, the average, and the standard deviation are computed for the
   1. number of instructions fetched from the instruction memory,
   2. number of destination addresses sent to the instruction memory (data tokens),
   3. number of gate signals sent to the instruction cells (control tokens),
4. number of acknowledgments sent to the predecessor instructions (feed back signals),
5. number of elementary data memory references, and
6. number of structure memory references.

b. Memory utilization The memory utilization consists of
   1. maximum number of integer, real and structure cells in use at any point in time,
   2. average and standard deviation of the above over the lifetime of the programs.

c. Processor utilization The processor utilization consists of
   1. maximum number of functional units of each type used during any period of time,
   2. average and standard deviation of each type of functional unit over those periods when a demand existed.

Table 2.2 presents the type, location in the simulator and execution stage during which the statistics are collected.

Data for Simulation

Since the constructs constituting a typical program are unknown, twenty-five programs from different sources [4, 14, 15, 21, 35] were selected as data for simulation. These programs represent algebraic computation and represent a set of real
Table 2.2. Description of Instrumentation

<table>
<thead>
<tr>
<th>Statistics collected</th>
<th>Simulator Module</th>
<th>Execution Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledge signal</td>
<td>Instruction Issuer</td>
<td>1. When the instruction is fetched for execution</td>
</tr>
<tr>
<td></td>
<td>and update</td>
<td>2. When an operand is false fired</td>
</tr>
<tr>
<td>Instruction fetch</td>
<td>Instruction Issuer</td>
<td>When the instruction is fetched from the Ready-List</td>
</tr>
<tr>
<td>Destinations</td>
<td>Update</td>
<td>When the enabler sends the result token to successor instructions</td>
</tr>
<tr>
<td>Control signal</td>
<td>Update</td>
<td>When the enabler sends the control tokens to successor instructions</td>
</tr>
<tr>
<td>Structure memory</td>
<td>Data memory</td>
<td>When the structure operations are executed by the functional units</td>
</tr>
<tr>
<td>reference</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elementary memory</td>
<td>Data memory</td>
<td>When the functional unit executes operations</td>
</tr>
<tr>
<td>reference</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer data cells</td>
<td>Data memory</td>
<td>When the memory manager creates and releases an integer data cell</td>
</tr>
<tr>
<td>Real data cells</td>
<td>Data memory</td>
<td>When the memory manager creates and releases a real data cell</td>
</tr>
<tr>
<td>Structure data cells</td>
<td>Data memory</td>
<td>When memory manager creates and releases a structure cell</td>
</tr>
<tr>
<td>Processor demand</td>
<td>Instruction Issuer</td>
<td>When forming the instruction packet and decoding the operation</td>
</tr>
</tbody>
</table>
programs without introducing undue programming bias, as some of the programs are in Fortran and some in Algol. There was no attempt made to select programs that exhibited a high degree of parallelism. Fortran programs containing the equivalence feature which introduces global variables were rejected. To get a general mix of programs in terms of high level control constructs, programs with the following characteristics were included in the sample.

1. Programs with data dependent blocks of assignment statements
2. Programs with single and nested conditionals
3. Programs with data dependent loops
4. Programs with data independent loops
5. Programs with nested loops
6. Programs with conditionals inside the loop
7. Programs with input/output
8. Programs involving array operations.

These programs were first translated to the high level target language according to a predetermined set of rules. Table 2.3 lists the application area, reference source, and static size of each program.

Transformations

Application of certain transformations to high level programs eliminates data dependency or alters the operator precedence without introducing indeterminacy in the results,
<table>
<thead>
<tr>
<th>Program</th>
<th>High Level Statement</th>
<th>Data Flow Instr.</th>
<th>Reference Source</th>
<th>Function Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEWB</td>
<td>35</td>
<td>69</td>
<td>X4</td>
<td>Solution of $X = N \times \exp(N)$</td>
</tr>
<tr>
<td>PRPF</td>
<td>90</td>
<td>295</td>
<td>S1</td>
<td>Normal Approximation</td>
</tr>
<tr>
<td>MULPLY</td>
<td>43</td>
<td>112</td>
<td>X4</td>
<td>Product of Chebyshev Series</td>
</tr>
<tr>
<td>GRJJP2</td>
<td>63</td>
<td>203</td>
<td>S1</td>
<td>Groups data into equally probable ranges</td>
</tr>
<tr>
<td>BP3</td>
<td>26</td>
<td>97</td>
<td>K4</td>
<td>Runge-Kutta for O.D.E.</td>
</tr>
<tr>
<td>BP6</td>
<td>43</td>
<td>144</td>
<td>K4</td>
<td>False position to find root</td>
</tr>
<tr>
<td>SSP2</td>
<td>40</td>
<td>131</td>
<td>X2</td>
<td>Differentiate a tabulated function</td>
</tr>
<tr>
<td>CUBN</td>
<td>45</td>
<td>117</td>
<td>X4</td>
<td>Integration using cubic spline</td>
</tr>
<tr>
<td>PLY</td>
<td>86</td>
<td>235</td>
<td>S1</td>
<td>Roots of a polynomial</td>
</tr>
<tr>
<td>GNFILT1</td>
<td>54</td>
<td>132</td>
<td>S1</td>
<td>Generate symmetrical filter with given amplitude response</td>
</tr>
<tr>
<td>RTD2</td>
<td>28</td>
<td>201</td>
<td>K4</td>
<td>Rigula falsi method to find root</td>
</tr>
<tr>
<td>SPDI</td>
<td>73</td>
<td>232</td>
<td>S1</td>
<td>Second probability density of integer series</td>
</tr>
<tr>
<td>EXSM</td>
<td>35</td>
<td>134</td>
<td>X3</td>
<td>Develops the triple exponential smoothed series</td>
</tr>
<tr>
<td>SE35</td>
<td>37</td>
<td>85</td>
<td>X3</td>
<td>Smooth an equidistantly tabled function using a third degree polynomial fit relevant to five points</td>
</tr>
<tr>
<td>MEBS</td>
<td>44</td>
<td>94</td>
<td>X3</td>
<td>Bounds for the Eigenvalues of a symmetric matrix</td>
</tr>
<tr>
<td>DGT3</td>
<td>51</td>
<td>108</td>
<td>X3</td>
<td>Differentiate a tabulated function using Lagragian interpolation of degree 2</td>
</tr>
<tr>
<td>JELF</td>
<td>74</td>
<td>152</td>
<td>X3</td>
<td>Jacobian elliptical function</td>
</tr>
<tr>
<td>MV5</td>
<td>66</td>
<td>168</td>
<td>S1</td>
<td>Moving mean square average</td>
</tr>
<tr>
<td>QSF</td>
<td>37</td>
<td>74</td>
<td>X3</td>
<td>Integration of tabulated function by Simpson's rule</td>
</tr>
<tr>
<td>SIMQ</td>
<td>98</td>
<td>235</td>
<td>X2</td>
<td>Solution of simultaneous linear equations</td>
</tr>
<tr>
<td>MXRA</td>
<td>92</td>
<td>288</td>
<td>S1</td>
<td>Maximizing a given function in a region</td>
</tr>
<tr>
<td>MEST</td>
<td>101</td>
<td>276</td>
<td>X3</td>
<td>Eigenvalues of a symmetric tridiagonal matrix</td>
</tr>
<tr>
<td>QATR</td>
<td>93</td>
<td>177</td>
<td>X2</td>
<td>Integration using trapezoidal rule with Romberg extrapolation</td>
</tr>
<tr>
<td>PECR</td>
<td>137</td>
<td>353</td>
<td>X3</td>
<td>Polynomial economization over the range (0-A)</td>
</tr>
<tr>
<td>ELI1</td>
<td>84</td>
<td>217</td>
<td>X3</td>
<td>Elliptical integral of first kind</td>
</tr>
</tbody>
</table>
thus increasing concurrent execution. Some of these transformations are complex. The tree transformations [20] which consist of associativity, commutativity and distributivity, are simple and easy to use. These transformations applied to arithmetic expressions reduce the parse tree height and hence the number of steps needed to evaluate the expression. Associative and commutative transformations do not increase the number of operations, but the distributivity transformation adds redundant calculation and may even increase the tree height.

Forward substitution is used to eliminate data dependency from a block of assignment statements. Forward substitution removes data dependency among statements and presents a larger tree for tree balancing transformation. Therefore, forward substitution along with tree balancing transformations reduce execution time by increasing concurrency. For other transformations and their effects on execution speedup refer to [20].

To study the effect of sequential file operation on execution, the input effect in programs is eliminated by assigning input data to the identifiers inside the program. The output effect is removed by eliminating the output statement in the high level language. Figure 2.2 shows the input/output transformation.

Example:

```
procedure IO
begin
    real A, B, C, D;
```

Figure 2.2. Input/Output Transformation
file inf, ouf;
input A, B file = inf format = F(5,2), F(5,4);
C := A*B + A*A;
D := C*A + A+B/2.0;
if C<D then C := C/4
else C := C/2 + D;
output C file = ouf format = F(8,5)
end

(a) Program With Input/Output

procedure IO
begin
  real A, B, C, D;
  A := } Data value
  B :=
  C := A*B + A*A;
  D := C*A + A+B/2.0;
  if C<D then C := C/4
else C := C/2 + D
end

(b) Program Without Input/Output

Simulation Results

Quantities measured

This section defines the quantities measured by executing the data flow program on the simulator.

Intrinsic parallelism  Intrinsic Parallelism is the ratio of time steps taken to execute the high level program sequentially ($\tau_s$) to the time steps taken to execute the
translated program on the simulator ($\tau_p$). The ratio, $\text{IP} = \frac{\tau_s}{\tau_p}$, depends on the computer organization, the structure of the computation, and the transformation used to expose concurrency. Intrinsic parallelism is measured assuming an underlying feedback data flow processor capable of scalar operations.

High level programs when translated to data flow form contain identity and merge operators which provide staging for operands. Also, to reduce complexity in code generation the compiler generates identity operations. These identity and merge operators are a part of the data flow language and do not have a sequential language counterpart. If the time to execute these operations are included in $\tau_s$ and $\tau_p$, $\tau_s$ gets inflated unproportionately compared to $\tau_p$. The reason for the uneven inflation of the execution time is due entirely to merge and identity executing concurrently. To eliminate the effect of merge and identity operators in the measurement of intrinsic parallelism, they are considered as overhead operators by assigning zero execution time. Now, $\tau_s$ is the sum of all the operation times and $\tau_p$ is the parallel execution time. The ratio of $\tau_s/\tau_p$ obtained by eliminating the effect of overhead operator is the Intrinsic Parallelism of the executed program.

Data path utilization Active procedures are held in the instruction memory and any instruction in the active procedure which is enabled is fetched for execution if a functional unit is available. The total number of such
instructions fetched in a clock cycle is the demand for data path imposed by the instruction fetch.

All the instructions have forward and backward links to successor and predecessor instructions. The predecessor instructions need an acknowledgment from all their successors before they can execute again. The number of acknowledgments sent in a clock cycle represents the data path demand due to acknowledgment signals.

On execution of enabled instructions, result tokens are sent to all successor instructions using the forward link in the executing instruction. The number of destinations (data tokens) sent in a clock cycle represent the data path demand for destinations.

A conditional instruction upon execution forwards control tokens to successor instructions. The number of control tokens forwarded in a clock cycle is the data path demand for control tokens.

When a functional unit executes an instruction, it fetches operand(s) from data memory and stores the result back in data memory. When the operation is a conditional the control token is directly distributed without storing it in the data memory. The number of references to the data memory is divided into structure and elementary memory references according to the type of operand being accessed. The number of accesses to the elementary data in a clock cycle represents the
the data path demand due to elementary data types. A similar result is obtained for the data path demand due to structure data access.

The above results represent the Data Path Utilization of a data flow program executing on a feedback data flow interpreter.

**Data memory utilization** Data memory utilization represents the number of data cells used by a program in execution. The number of data cells created minus the number of data cells released in a clock cycle represent the data memory demand for one clock cycle. The total number of data cells used up to that time represent total data memory demand for each type of data.

**Processor utilization** The function units are capable of executing only one specified type of operation and the processor utilization is the number of functional units of each type used by the executing program. The average processor demand is the calculated average over those periods when a demand existed.

**Operation execution time**

For the simulated execution of data flow programs on the feedback data flow interpreter, each operation is assigned an execution time. Assigned execution times are integral multiples of the memory cycle time which is assumed to be unity. Figure 2.3 represents the model used for scalar operands.
Register

Register

Binary Operator

Register

Data Memory

$t_{\text{operation}} = t_{\text{fetch}} + t_{\text{function}} + t_{\text{store}}$

Figure 2.3. Model for Scalar Operation

The operands are fetched from the data memory to the registers in a memory cycle time. Then the specified operation is performed producing a result after a certain amount of elapsed time depending on the complexity of the operation. The result is then moved to the data memory from the result register in one memory cycle time. Note that the memory bandwidth is assumed to be sufficient to support an arbitrary number of functional units accessing data memory.

Assuming the operation of the add functional unit to be the same as the memory cycle time, an add instruction consists of two memory fetches to obtain the operands, unit add operation time and a memory cycle to store the result. Thus, an add takes four ($t_{\text{add}} = 2+1+1 = 4$) units of memory cycle time. Append, select, and cons are considered as memory operations and need only a memory cycle time.
The execution time of other instructions and builtin functions (such as abs, sort, etc.) are obtained in terms of an add operation. Implementing Delugish [9] algorithms for builtin functions in hardware, using a maximum of three adders reduces operation times to a few multiplication cycles. These algorithms use normalization and redundant representation of output digits and trade hardware for faster execution time. Algorithms using normalization techniques represent operands or a simple function of that operand as a continued product or continued summation, where the terms of the summation or product can easily be chosen in a step by step process.

The builtin functions and divide operation can be executed in one to three multiplication cycles to a register length precision, assuming that the time needed for addition is significantly greater than the time necessary to perform low precision comparisons, shift operations or complementations and a fast ROM with a size of 100 words is available. Therefore, the operations such as divide, logarithm, square root, arc tangent and absolute value take one multiplication cycle; tangent, sine and cosine take two; and exponentiation, arc sine and arc cosine take three multiplication cycles, where a multiplication cycle time is defined as the time required to perform, on the average, m low precision comparisons, m shifting operations and m/3 additions, m being the length of the mantissa of a floating point operand.
An apply operation is assigned an execution time of fourteen, which includes setting up the procedure execution environment and passing the argument structure. Read and write operations take ten memory cycles, including data conversion. Readedit and writedit operations execute in three memory cycles. Finally, multiplication operation is assigned an execution time of three memory cycles. Table 2.4 shows the feedback data flow interpreter operations and their operation times.

Table 2.4. Execution Time of Base Machine Operations

<table>
<thead>
<tr>
<th>Operation Time</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>select, append, constant</td>
</tr>
<tr>
<td>3</td>
<td>negate, not, readedit, writedit</td>
</tr>
<tr>
<td>4</td>
<td>+, -, relationals, or, and</td>
</tr>
<tr>
<td>5</td>
<td>abs, arctan, log&lt;sub&gt;e&lt;/sub&gt;, sqrt</td>
</tr>
<tr>
<td>6</td>
<td>*, /</td>
</tr>
<tr>
<td>8</td>
<td>tanh, cosh, sinh, cos, sin, tan</td>
</tr>
<tr>
<td>10</td>
<td>read, write</td>
</tr>
<tr>
<td>12</td>
<td>arcsin, arccos, **</td>
</tr>
<tr>
<td>14</td>
<td>apply</td>
</tr>
</tbody>
</table>
To study the sensitivity of intrinsic parallelism to changes in operation execution time, the execution time of all the operators except the builtin functions and read are assigned a value of unity. Read operation is assigned a value of two, as it uses two append operations to produce the structure consisting of the input data value and file. The builtin functions are again given execution times as in the previous case. Table 2.5 represents the second set of execution time.

Table 2.5. Operator Execution Times

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>negate, abs, not, atan, +, -, *, /, relationals, select, append, apply log, sqrt, write, readedit, writedit, cons</td>
</tr>
<tr>
<td>2</td>
<td>tan, cos, sin, tanh, cosh, sinh, read</td>
</tr>
<tr>
<td>3</td>
<td>acos, asin, **</td>
</tr>
</tbody>
</table>

Results

The intrinsic parallelism in the high level programs and the resource requirements to execute the programs, whose characteristics are shown in Table 2.6, are obtained by translating the high level program to data flow language and executing on a feedback data flow interpreter. The programs
Table 2.6. Program Characteristics

<table>
<thead>
<tr>
<th>Program Name</th>
<th>High Level</th>
<th>Assembly Instrs.</th>
<th>Total Iterations</th>
<th>Conditionals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statement</td>
<td>Total Nesting Level</td>
<td># statements</td>
<td>Total Nesting Level</td>
</tr>
<tr>
<td>W EW B</td>
<td>35</td>
<td>69</td>
<td>11 2 49 2 1 15</td>
<td></td>
</tr>
<tr>
<td>PRPF</td>
<td>90</td>
<td>295</td>
<td>5 2 27 2 1 67</td>
<td></td>
</tr>
<tr>
<td>ML T PL Y</td>
<td>43</td>
<td>112</td>
<td>2 2 31 6 5 47</td>
<td></td>
</tr>
<tr>
<td>GR UP 2</td>
<td>63</td>
<td>203</td>
<td>1 1 16 2 2 67</td>
<td></td>
</tr>
<tr>
<td>BP 3</td>
<td>26</td>
<td>97</td>
<td>40 4 4 4 33</td>
<td></td>
</tr>
<tr>
<td>BP 6</td>
<td>43</td>
<td>144</td>
<td>1 1 8 2 2 29</td>
<td></td>
</tr>
<tr>
<td>SS P 2</td>
<td>40</td>
<td>131</td>
<td>1 1 22 2 2</td>
<td></td>
</tr>
<tr>
<td>CUBN</td>
<td>45</td>
<td>117</td>
<td>3 1 59 5 3 66</td>
<td></td>
</tr>
<tr>
<td>PLY</td>
<td>86</td>
<td>235</td>
<td>6 3 59 5 3 66</td>
<td></td>
</tr>
<tr>
<td>G N P L T 1</td>
<td>54</td>
<td>132</td>
<td>3 2 30 4 3 41</td>
<td></td>
</tr>
<tr>
<td>R T D 2</td>
<td>28</td>
<td>201</td>
<td>2 2 25 2 1 1 6</td>
<td></td>
</tr>
<tr>
<td>S P D I</td>
<td>73</td>
<td>232</td>
<td>5 2 28 8 6 58</td>
<td></td>
</tr>
<tr>
<td>E X S M</td>
<td>35</td>
<td>134</td>
<td>1 1 8 3 3 9</td>
<td></td>
</tr>
<tr>
<td>S E 35</td>
<td>37</td>
<td>85</td>
<td>2 2 12 1 1 1 10</td>
<td></td>
</tr>
<tr>
<td>M E B S</td>
<td>44</td>
<td>94</td>
<td>2 2 11 1 1 1 25</td>
<td></td>
</tr>
<tr>
<td>D G T 3</td>
<td>51</td>
<td>108</td>
<td>2 2 22 5 2 37</td>
<td></td>
</tr>
<tr>
<td>J E L P</td>
<td>74</td>
<td>152</td>
<td>1 1 21 6 3 56</td>
<td></td>
</tr>
<tr>
<td>M V S</td>
<td>66</td>
<td>168</td>
<td>2 2 28 4 4 54</td>
<td></td>
</tr>
<tr>
<td>Q S F</td>
<td>37</td>
<td>74</td>
<td>2 2 13 1 1 1 23</td>
<td></td>
</tr>
<tr>
<td>S I M Q</td>
<td>98</td>
<td>235</td>
<td>7 3 75 4 2 45</td>
<td></td>
</tr>
<tr>
<td>M X R A</td>
<td>92</td>
<td>288</td>
<td>4 3 61 10 6 76</td>
<td></td>
</tr>
<tr>
<td>M E S T</td>
<td>101</td>
<td>276</td>
<td>6 3 69 8 3 60</td>
<td></td>
</tr>
<tr>
<td>Q A T R</td>
<td>93</td>
<td>177</td>
<td>6 3 48 6 5 66</td>
<td></td>
</tr>
<tr>
<td>P E C R</td>
<td>137</td>
<td>353</td>
<td>6 3 78 12 4 119</td>
<td></td>
</tr>
<tr>
<td>E L I 1</td>
<td>84</td>
<td>217</td>
<td>1 1 31 10 4 64</td>
<td></td>
</tr>
</tbody>
</table>
are executed with only one set of data and the selected data is such that the execution follows the normal conditional branches. The iterations in the program are executed a minimum of five times.

Intrinsic parallelism of a given program is the ratio of the sequential execution time ($\tau_s$) to parallel execution time ($\tau_p$). The sequential and parallel execution times are obtained by setting the operation times of identity and merge instructions to zero, and the execution time of other instructions are either selected from Table 2.4 or Table 2.5. The assignment of zero time for identity and merge operations allows a comparison with the execution time on an equivalent sequential machine which required no synchronization and also eliminates the effect of known overhead identity operations generated by the compiler. The instruction execution times in Table 2.4 include the memory fetch time as part of the operation, whereas the execution times given in Table 2.5 include only operation time. The intrinsic parallelism and resource requirements corresponding to minimum parallel execution time are obtained by providing more than enough functional units of each type to eliminate resource contention. The sequential and parallel execution times along with the intrinsic parallelism are shown in Table 2.7. The ratio $\tau_s/\tau_p$ represents the intrinsic parallelism in the high level program using instruction execution time without memory fetch,
Table 2.7. Intrinsic parallelism for two sets of execution times

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Parallelism Based on Add Time of 1 Unit</th>
<th>Parallelism Based on Add Time of 4 Units</th>
<th>Percentage Change</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\tau_s$, $\tau_p$, $\tau_s'/\tau_p$</td>
<td>$\tau_s'$, $\tau_p'$, $\tau_s'/\tau_p'$</td>
<td></td>
</tr>
<tr>
<td>WEWB</td>
<td>41, 22, 1.86</td>
<td>196, 111, 1.76</td>
<td>5.37</td>
</tr>
<tr>
<td>PRPF</td>
<td>947, 295, 3.21</td>
<td>2284, 703, 3.24</td>
<td>0.93</td>
</tr>
<tr>
<td>MLTPLY</td>
<td>367, 157, 2.33</td>
<td>1216, 571, 2.12</td>
<td>9.01</td>
</tr>
<tr>
<td>GRUP2</td>
<td>457, 270, 1.69</td>
<td>1750, 984, 1.77</td>
<td>4.73</td>
</tr>
<tr>
<td>BP3</td>
<td>700, 203, 3.44</td>
<td>3676, 1083, 3.39</td>
<td>1.45</td>
</tr>
<tr>
<td>BP6</td>
<td>103, 58, 1.77</td>
<td>444, 239, 1.85</td>
<td>4.51</td>
</tr>
<tr>
<td>SSP2</td>
<td>207, 117, 1.76</td>
<td>771, 461, 1.67</td>
<td>5.11</td>
</tr>
<tr>
<td>CUBN</td>
<td>542, 228, 2.37</td>
<td>1646, 746, 2.21</td>
<td>7.17</td>
</tr>
<tr>
<td>PLY</td>
<td>265, 112, 2.36</td>
<td>833, 393, 2.11</td>
<td>10.59</td>
</tr>
<tr>
<td>GNFLTl</td>
<td>305, 111, 2.74</td>
<td>1273, 448, 2.84</td>
<td>3.64</td>
</tr>
<tr>
<td>RTD2</td>
<td>586, 254, 2.31</td>
<td>2809, 1135, 2.47</td>
<td>7.36</td>
</tr>
<tr>
<td>SPDI</td>
<td>618, 253, 2.50</td>
<td>2272, 871, 2.60</td>
<td>4.00</td>
</tr>
<tr>
<td>ESM</td>
<td>361, 156, 2.31</td>
<td>1341, 552, 2.38</td>
<td>3.03</td>
</tr>
<tr>
<td>SE35</td>
<td>206, 130, 1.58</td>
<td>730, 480, 1.52</td>
<td>3.79</td>
</tr>
<tr>
<td>MEBS</td>
<td>171, 83, 2.06</td>
<td>598, 315, 1.89</td>
<td>8.25</td>
</tr>
<tr>
<td>DGT3</td>
<td>180, 77, 2.33</td>
<td>610, 266, 2.29</td>
<td>1.71</td>
</tr>
<tr>
<td>JELF</td>
<td>158, 64, 2.47</td>
<td>678, 321, 2.11</td>
<td>14.57</td>
</tr>
<tr>
<td>MVS</td>
<td>178, 85, 2.09</td>
<td>561, 252, 2.22</td>
<td>6.22</td>
</tr>
<tr>
<td>QSF</td>
<td>159, 87, 1.82</td>
<td>551, 316, 1.74</td>
<td>4.39</td>
</tr>
<tr>
<td>SIMQ</td>
<td>413, 200, 2.07</td>
<td>1339, 641, 2.08</td>
<td>0.48</td>
</tr>
<tr>
<td>MXRA</td>
<td>288, 160, 1.80</td>
<td>854, 546, 1.56</td>
<td>13.33</td>
</tr>
<tr>
<td>MEST</td>
<td>534, 275, 1.94</td>
<td>1837, 1014, 1.81</td>
<td>6.70</td>
</tr>
<tr>
<td>QATR</td>
<td>554, 202, 2.74</td>
<td>2328, 803, 2.89</td>
<td>5.47</td>
</tr>
<tr>
<td>PECCR</td>
<td>145, 55, 2.63</td>
<td>466, 174, 2.67</td>
<td>1.52</td>
</tr>
<tr>
<td>ELII</td>
<td>41, 16, 2.56</td>
<td>173, 72, 2.40</td>
<td>6.25</td>
</tr>
</tbody>
</table>
and the ratio $t_s'/t_p'$ is the measure of intrinsic parallelism when the instruction execution time includes memory fetch. Percentage change, indicates the effect of including memory fetch in the instruction execution time. The percentage change in absolute value of intrinsic parallelism due to memory fetch range from 0.48 (SIMQ) to 14.4 (JELF). Without any attempt to optimize the programs, the measured parallelism ranged from 1.58 to 3.44, when the fetch time is not part of instruction execution time, and ranged from 1.52 to 3.39 when the fetch time is part of instruction execution time.

The instruction execution times in Table 2.5 is used to obtain the results in the rest of this section. To study the optimizing effect of forward substitution transformation within statements of the high level program and the inhibiting effect of sequential file operation on parallel execution time, the identity and the merge operators are assigned zero execution times. Table 2.8 shows the parallelism in the high level program. The column $1 - t_p'/t_p$ indicates the reduction in parallel execution time obtained by applying the forward substitution transformation (including a balance of the resulting expression tree) to a block of assignment statements. The decrease in parallel execution time ranges from 0 for WEWB to 4.0 for BP3. Except for BP3 and SE35, the reduction in parallel execution times are less than 10 percent. A substantial reduction in parallel execution time is possible, if
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Original Program</th>
<th>High Level Forward Substitution</th>
<th>Input/Output Ignored</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( t_s )</td>
<td>( t_p )</td>
<td>( t_s / t_p )</td>
</tr>
<tr>
<td>WEEB</td>
<td>41</td>
<td>22</td>
<td>1.86</td>
</tr>
<tr>
<td>PRPF</td>
<td>947</td>
<td>295</td>
<td>3.21</td>
</tr>
<tr>
<td>MLTPLY</td>
<td>367</td>
<td>157</td>
<td>2.33</td>
</tr>
<tr>
<td>GRUP2</td>
<td>457</td>
<td>270</td>
<td>1.69</td>
</tr>
<tr>
<td>BP3</td>
<td>700</td>
<td>203</td>
<td>3.44</td>
</tr>
<tr>
<td>BP6</td>
<td>103</td>
<td>58</td>
<td>1.77</td>
</tr>
<tr>
<td>SSP2</td>
<td>207</td>
<td>117</td>
<td>1.76</td>
</tr>
<tr>
<td>CUBN</td>
<td>542</td>
<td>228</td>
<td>2.37</td>
</tr>
<tr>
<td>PLY</td>
<td>265</td>
<td>112</td>
<td>2.36</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>305</td>
<td>111</td>
<td>2.74</td>
</tr>
<tr>
<td>RFD2</td>
<td>586</td>
<td>254</td>
<td>2.31</td>
</tr>
<tr>
<td>SPD1</td>
<td>618</td>
<td>253</td>
<td>2.50</td>
</tr>
<tr>
<td>EXSM</td>
<td>361</td>
<td>156</td>
<td>2.31</td>
</tr>
<tr>
<td>SE35</td>
<td>206</td>
<td>130</td>
<td>1.58</td>
</tr>
<tr>
<td>MEBS</td>
<td>171</td>
<td>83</td>
<td>2.06</td>
</tr>
<tr>
<td>DGT3</td>
<td>180</td>
<td>77</td>
<td>2.33</td>
</tr>
<tr>
<td>JELF</td>
<td>158</td>
<td>64</td>
<td>2.47</td>
</tr>
<tr>
<td>MVS</td>
<td>178</td>
<td>85</td>
<td>2.09</td>
</tr>
<tr>
<td>QSF</td>
<td>159</td>
<td>87</td>
<td>1.82</td>
</tr>
<tr>
<td>SIMQ</td>
<td>413</td>
<td>200</td>
<td>2.07</td>
</tr>
<tr>
<td>MXRA</td>
<td>288</td>
<td>160</td>
<td>1.80</td>
</tr>
<tr>
<td>MEST</td>
<td>534</td>
<td>275</td>
<td>1.94</td>
</tr>
<tr>
<td>QATR</td>
<td>554</td>
<td>202</td>
<td>2.74</td>
</tr>
<tr>
<td>PECD</td>
<td>145</td>
<td>55</td>
<td>2.63</td>
</tr>
<tr>
<td>ELII</td>
<td>41</td>
<td>16</td>
<td>2.56</td>
</tr>
</tbody>
</table>
forward substitution produces longer expression trees with operators which allow tree balancing. Even though forward substitution produced longer expressions, the operators in the expression did not permit the formation of a balanced tree as reflected by the zero reduction in the parallel execution time and the increase in the number of operations for the program WEWB. The ratio $\tau_s'/\tau_s$ is an indication of the redundancy of operations introduced by the forward substitution transformation. The amount of redundancy introduced ranges from 1.01 for MXRA to 3.48 for RTD2. The redundancy factor is a function of the number of dependent statements that can be eliminated in the basic block after forward substitution. The redundancy factor increases with the decrease of dependent statements that can be eliminated. Care should be taken in applying forward substitution transformation to programs to be executed on a feedback data flow interpreter.

To measure the effect of sequential file operation on concurrent execution of programs, the output statements in the high level program were eliminated and the input statements were replaced by statements which assign input data values to identifiers in the input list. This makes all the input data values available simultaneously, except in the case of a subscripted identifier. For the subscripted identifier the values are appended to form a structure before they can be used any further. The ratio $\tau_s''/\tau_p$ in Table 2.8 shows the
intrinsic parallelism without the sequential input/output file operations. The intrinsic parallelism varies from 1.60 for MXRA to 4.08 for PECR. For programs BP3, JELF and MXRA, there was a reduction in the intrinsic parallelism when the I/O effect was removed. This indicates the contribution of sequential overlapped I/O operation to the intrinsic parallelism. For all the programs the parallel execution time $\tau_p''$ is less than the corresponding parallel execution time $\tau_p$ for the original program. The value $1 - \frac{\tau_p''}{\tau_p}$ indicates the decrease in parallel execution time and this ranges from 8 to 60 percent.

Table 2.9 shows the intrinsic parallelism, the execution speedup, and the overhead necessary to maintain data determinacy and to avoid deadlock for the twenty-five programs. The ratio $\frac{\tau_s'}{\tau_p'}$ (execution speedup) is obtained by assigning an execution time of unity to identify and merge operators. The execution speedup ranges from 2.2 for SE35 to 4.97 for PECR. The increase in the ratio $\frac{\tau_s'}{\tau_p'}$ is due to the substantial increase in the sequential execution time ($\tau_s'$) compared to the parallel execution time ($\tau_p'$). The relatively small increase in the parallel execution time is due to the fact that the overhead operators execute in the same time step. The overhead required to provide deadlock-free determinate concurrent processing is represented by $\frac{\tau_s'}{\tau_s}$ for the feedback data flow interpreter. This overhead ranges from 1.16 for BP3 to 3.12 for GNFLT1. The overhead factor is a function of the
## Table 2.9. Intrinsic Parallelism and Speedup

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Execution Time Without Overhead Operators</th>
<th>Execution Time With Overhead Operators</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\tau_s$ $\tau_p$ $\tau_s/\tau_p$ $\tau_s'$ $\tau_p'$ $\tau_s'/\tau_p'$ $\tau_s'/\tau_s$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WEWEB</td>
<td>41  22  1.86  60  24  2.50  1.46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRPF</td>
<td>947 295  3.21 1377 319  4.31 1.45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLTPLV</td>
<td>367 157  2.33  642 208  3.00  1.74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GRUP2</td>
<td>457 270  1.69 1299 482  2.69  2.84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP3</td>
<td>700 203  3.44  812 213  3.812 1.16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP6</td>
<td>103  58  1.77  232  75  3.09  2.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSP2</td>
<td>207 117  1.76  427 172  2.48  2.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUBN</td>
<td>542 228  2.37  823 285  2.88  1.51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLY</td>
<td>265 112  2.36  531 165  3.21  2.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GNFLT1</td>
<td>305 111  2.74  954 251  3.80  3.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTD2</td>
<td>586 254  2.307 931 301  3.09  1.58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPD1</td>
<td>618 253  2.50 1313 415  3.20  2.10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXSM</td>
<td>361 156  2.31  596 197  3.02  1.65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SE35</td>
<td>206 130  1.58  374 170  2.20  1.81</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEBS</td>
<td>171  83  2.06  323 118  2.73  1.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGT3</td>
<td>180  77  2.33  367 110  3.33  2.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JELF</td>
<td>158  64  2.47  327 103  3.10  2.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>178  85  2.09  319 126  2.50  1.80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QSF</td>
<td>159  87  1.82  315 123  2.56  1.98</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIMQ</td>
<td>413 200  2.06  844 293  2.88  2.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MXRA</td>
<td>288 160  1.80  613 245  2.50  2.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEST</td>
<td>534 275  1.94 1186 398  2.97  2.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QATR</td>
<td>554 202  2.74  969 282  3.43  1.74</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PECR</td>
<td>145  55  2.63  438  88  4.97  3.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELII</td>
<td>41   16  2.56  75   24  3.13  1.82</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
program structure. The presence of conditional and iterative constructs increase the overhead operators and hence the overhead factor.

The resource requirements consist of data path, data memory and processor requirements. The resource requirements are obtained for minimum parallel execution time by assigning unit execution times to the identity and merge (overhead) operators. The results presented in the following tables are selected to show the highlights of simulator output; the complete set of simulator output data appears in the Appendix.

Table 2.10 shows the data path requirements for a selected subset of the programs. The selected results show the range of values produced by executing the programs on the feedback data flow simulator. In each entry the first figure is the maximum resource requirement encountered during any time step of the simulated execution. The second figure gives the average resource requirements over the complete execution and the third figure is the standard deviation from this average. The row labeled composite in Table 2.10 and Table 2.11 summarizes the results for all the test programs. The first figure is the average of the maximum resource requirements. The second figure is the average of the twenty-five averages, and the third is the standard deviation associated with the overall averages.
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Instructions Fetched</th>
<th>Acknowledgements</th>
<th>Destinations</th>
<th>Control Tokens</th>
<th>Structure Memory References</th>
<th>Elementary Memory References</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRPF</td>
<td>34/4.1/3.6</td>
<td>48/8.8/6.8</td>
<td>51/6.5/6.7</td>
<td>34/3.3/6.8</td>
<td>50/3.5/5.2</td>
<td>46/6.1/5.3</td>
</tr>
<tr>
<td>PECR</td>
<td>27/4.9/4.4</td>
<td>50/11.8/10.2</td>
<td>42/7.4/7.6</td>
<td>33/4.8/7.9</td>
<td>18/2.8/3.5</td>
<td>38/6.5/6.3</td>
</tr>
<tr>
<td>RTD2</td>
<td>13/2.6/2.8</td>
<td>37/5.4/7.3</td>
<td>31/3.9/5.1</td>
<td>24/1.5/4.9</td>
<td>3/0.3/0.8</td>
<td>14/4.7/3.5</td>
</tr>
<tr>
<td>EXSM</td>
<td>6/2.9/1.6</td>
<td>13/6.0/3.4</td>
<td>14/4.4/2.8</td>
<td>21/1.9/5.3</td>
<td>12/1.7/2.5</td>
<td>24/4.8/4.4</td>
</tr>
<tr>
<td>SE35</td>
<td>11/2.1/1.3</td>
<td>20/4.7/3.7</td>
<td>14/3.2/2.3</td>
<td>21/1.6/4.9</td>
<td>11/1.4/2.2</td>
<td>19/3.2/2.5</td>
</tr>
<tr>
<td>BP3</td>
<td>9/3.5/1.9</td>
<td>20/6.2/4.0</td>
<td>22/5.5/4.1</td>
<td>22/1.0/4.7</td>
<td>3/0.1/0.6</td>
<td>21/7.7/3.9</td>
</tr>
<tr>
<td>Composite</td>
<td>13.5/3.0/0.6</td>
<td>27.3/6.4/1.5</td>
<td>23.8/4.5/1.0</td>
<td>21.9/2.3/0.8</td>
<td>15.2/1.7/1.0</td>
<td>21.5/4.4/1.4</td>
</tr>
</tbody>
</table>
Table 2.11. Representative Data Memory Requirements

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Real Cells</th>
<th>Integer Cells</th>
<th>Structure Cells</th>
<th>Boolean Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRPF</td>
<td>55/36.1/15.4</td>
<td>44/15.6/9.3</td>
<td>35/29/4/8.5</td>
<td>2/0.01/0.1</td>
</tr>
<tr>
<td>PECR</td>
<td>21/12.6/6.6</td>
<td>25/11.9/7.1</td>
<td>15/7.0/5.9</td>
<td>0/0/0</td>
</tr>
<tr>
<td>RTD2</td>
<td>17/13.2/5.2</td>
<td>4.2.5/1.2</td>
<td>2/0.1/0.4</td>
<td>2/0.72/0.53</td>
</tr>
<tr>
<td>EXSM</td>
<td>31/16.1/8.6</td>
<td>6/3.5/1.2</td>
<td>21/9.9/6.4</td>
<td>2/0.06/0.29</td>
</tr>
<tr>
<td>SE35</td>
<td>20/10.5/5.8</td>
<td>8/6.0/1.5</td>
<td>17/8.9/4.6</td>
<td>0/0/0</td>
</tr>
<tr>
<td>BP3</td>
<td>25/18.9/6.2</td>
<td>0/0/0</td>
<td>2/0.1/0.3</td>
<td>0/0/0</td>
</tr>
<tr>
<td>MXRA</td>
<td>27/17.8/7.9</td>
<td>14/11.1/1.5</td>
<td>21/14.9/6.9</td>
<td>3/0.19/0.54</td>
</tr>
<tr>
<td>Composite</td>
<td>25.2/14.6/8.3</td>
<td>12.7/8.0/5.2</td>
<td>15.1/8.8/7.3</td>
<td>1.04/0.21/0.20</td>
</tr>
</tbody>
</table>

Data path requirements The instruction fetch indicates the traffic between the instruction memory and the functional units. The average number of instruction fetch varies from 2 to 5 and the standard deviation ranges from 1.3 to 4.4. The maximum number of instructions fetched at any time step in the execution of the program varies from 8 to 34. The significantly higher maximums are largely attributable to three operators: merge, identity and cons. The merge operator is provided to eliminate data non-determinancy by serving as a staging place for data values. Each identifier in an iteration construct has a merge operator and most of these operators execute simultaneously, thereby pushing up the maximum number of instructions fetched. The compiler generates
identity operators to follow the merge operators to distribute the data values to the operators inside the body of the iteration construct. The identity operator also serves as a single gating point for the gated identifier, thereby reducing the amount of signal flow. The cons operator is used to initialize identifiers which are assigned constant value in the program. The higher maximum instruction fetch occurs during the moments of time when the merges, identifiers or cons execute. Note that the average number of instruction fetch is very close to the speedup obtained with overhead, shown in Table 2.9. Any deviation present is due to the presence of instructions with non-unity execution time.

The number of acknowledgments are the back signals sent to the predecessor instructions which supplied the operands (including gate value) to the fetched instructions. The average number of acknowledgment ranges from 3 to 12 and the maximum number of acknowledgment varies from 13 to 15. The peak value occurs when identity instructions used to stage values for gating are enabled simultaneously by the arrival of control tokens from the conditional for the iteration construct. The average acknowledgment is approximately twice the number of fetches, indicating the dominance of the binary operators.

The functional unit sends the address of the result produced by the operation to successor instructions defined
by the destination field. The number of result tokens sent to successor instructions represents the traffic due to the destinations (data tokens). The average number of destinations ranges from 3 to 8 and the peak traffic again occurs when the identity instructions distribute data values to instructions in the body of the iteration construct. The peak value varies from 13 to 51.

Control tokens are the number of gate values sent to instructions by the relational functional unit. Peak loads in control token occur when the decider of the iterative construct sends signals to merge and identity instructions in the iteration construct.

Structure memory reference indicates the number of references made to the structure type data cells. These references are largely due to array operations. Another contributor to structure memory references is the input instruction. Memory reference made do not include the traffic due to the reference count management of the data memory cells. The elementary memory references consist of the number of references made to all elementary data types. The average elementary memory references vary from 3.1 to 7.7.

Data memory requirements Table 2.11 shows the data memory requirements for a selected subset of the programs. The real, integer, structure and boolean cells represent the
number of data cells of each type allocated for the execution of the program at any time. These are the data cells whose reference counts are not zero at any point in time. The average value of real cells needed ranged from 4.1 to 36.1. The average number of integer and structure cells varied from 0 to 22.4 and 0.1 to 29.4, respectively. This is significantly less than the amount of memory required using traditional block memory management methods.

**Processor requirements** The maximum functional units of each type required to execute each of the twenty-five programs appear in the Appendix. Table 2.12 shows only the functional units for which the maximum demand was greater than two. For the purpose of comparison the maximum number of fetches is also given in Table 2.12.

The intrinsic parallelism and the low data memory requirements show the relative advantage of the feedback data flow interpreter over an equivalent sequential machine. The cost of these advantages is illustrated in Table 2.10 by the extensive flow of informations within the network which interface the functional units and memories.

**Conclusions**

This chapter illustrates how simulation has been used in evaluating the performance of a feedback data flow processor capable of scalar operations only. The intrinsic parallelism for the twenty-five small programs is only modest compared to
Table 2.12. Maximum functional unit requirements (SEL-select, NEG-negate)

<table>
<thead>
<tr>
<th>Program</th>
<th>Max Instr</th>
<th>Merge</th>
<th>Ident</th>
<th>Cons</th>
<th>Maximum Functional Units Needed</th>
<th>Other major requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEB</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>9</td>
<td>4(/)</td>
<td></td>
</tr>
<tr>
<td>PRPF</td>
<td>34</td>
<td>13</td>
<td>10</td>
<td>10</td>
<td>10(*), 8(+), 4(SEL), 3(**), 3(&gt;)</td>
<td></td>
</tr>
<tr>
<td>MLTPLY</td>
<td>14</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>6(SEL), 5(+), 3(*)</td>
<td></td>
</tr>
<tr>
<td>GRUP2</td>
<td>15</td>
<td>12</td>
<td>13</td>
<td>8</td>
<td>3(NEG)</td>
<td></td>
</tr>
<tr>
<td>BP3</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>4(*), 3(+)</td>
<td></td>
</tr>
<tr>
<td>BP6</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>SSP2</td>
<td>10</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>5(SEL), 3(+), 3(-)</td>
<td></td>
</tr>
<tr>
<td>CUBN</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>5(SEL), 4(+), 3(-)</td>
<td></td>
</tr>
<tr>
<td>PLY</td>
<td>16</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>4(SEL), 3(*)</td>
<td></td>
</tr>
<tr>
<td>GNFLETl</td>
<td>12</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>4(+)</td>
<td></td>
</tr>
<tr>
<td>RTD2</td>
<td>13</td>
<td>5</td>
<td>13</td>
<td>2</td>
<td>4(*), 4(**)</td>
<td></td>
</tr>
<tr>
<td>SPDI</td>
<td>18</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>5(+), 3(*), 3(NEG)</td>
<td></td>
</tr>
<tr>
<td>EXSM</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6(SEL)</td>
<td></td>
</tr>
<tr>
<td>SE35</td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4(SEL)</td>
<td></td>
</tr>
<tr>
<td>MEBS</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>5(SEL), 3(+)</td>
<td></td>
</tr>
<tr>
<td>DGT3</td>
<td>11</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4(SEL), 4(-)</td>
<td></td>
</tr>
<tr>
<td>JELF</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>17</td>
<td>10</td>
<td>6</td>
<td>4</td>
<td>9(SEL), 4(*), 4(+), 4(-)</td>
<td></td>
</tr>
<tr>
<td>QSF</td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4(SEL), 3(*)</td>
<td></td>
</tr>
<tr>
<td>SIMQ</td>
<td>12</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>4(+), 3(SEL), 3(-)</td>
<td></td>
</tr>
<tr>
<td>MXRA</td>
<td>11</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>4(SEL), 3(NEG), 3(=)</td>
<td></td>
</tr>
<tr>
<td>MEST</td>
<td>15</td>
<td>11</td>
<td>12</td>
<td>11</td>
<td>3(SEL)</td>
<td></td>
</tr>
<tr>
<td>QATR</td>
<td>11</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4(+)</td>
<td></td>
</tr>
<tr>
<td>PECR</td>
<td>27</td>
<td>14</td>
<td>14</td>
<td>7</td>
<td>4(+), 3(SEL)</td>
<td></td>
</tr>
<tr>
<td>ELII</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
a sequential processing capability. The data memory requirement is significantly less than that of a system which uses a traditional block memory allocation feature. The processor requirement is dominated by the overhead operators. The main drawback seems to be the cost, in terms of information flow in the network connecting the memory and functional units, thereby indicating the need for better memory organization and modifications to the processor to achieve a reduction in the data path requirements.

The simulation studies do not measure the potential parallelism which could be exploited by the concept of streams. However, it appears that only four of the programs PLY, SPDI, MLTP and CUBN have a significant amount of such exploitable parallelism. By assuming the availability of vector functional units, this exploitable parallelism is estimated by graph analysis techniques in Chapter II.
CHAPTER III. ANALYSIS OF PROGRAM GRAPH

Introduction

System performance can be predicted without building the real system by using either simulation or analysis techniques. In Chapter II, the simulation principle was used to study the performance of a feedback data flow processor. Even though simulation techniques predict system performance at a step closer to the real system, the effort and cost may become prohibitive and moreover the structural relationship among system components may not be easily visible to a casual observer. On the other hand, analytical models not only show explicitly the structural relationship but also provide a clear understanding of the analyzed system. The usefulness and accuracy of a model under a given situation will depend on how closely the model matches the modeled system and the modeling tools used in extracting the results.

As a modeling tool both Petri nets and directed graphs are powerful [3, 5, 10, 12, 17, 19, 24, 31, 34], yet flexible enough to represent asynchronous concurrent systems. They model in a natural way processes which are inherently parallel and restricted by precedence relationship.

Although the directed graph model is used to predict the intrinsic parallelism present in high level programs, the Petri net model could have been used instead of the directed graph.
In this chapter, the section called Definition and Notations is an introduction to graph theory, the section on Background Material reviews previous work and the section on Acyclic Directed Graph for Model I Architecture contains the performance analysis assuming a feedback data flow processor capable of operations on elementary data types and individual structure components as commonly found in algebraic languages [5].

Graph Representation for Model II Architecture presents the performance analysis assuming a second model architecture which is an enhancement of the first in that vector operations are added to the basic feedback data flow processor. The last section in this chapter extends the analysis to include a Stream Read feature.

Definitions and Notations

Before presenting the directed graph model and performance analysis, the formal definition of the directed graph [13,29] together with the related terminology used in this thesis is introduced.

A graph, G, is a three tuple \(<N, A, F>\) where

where \(N\) is a non-empty set of unique nodes 
\[\{n_1, n_2, \ldots, n_k\}\].

\(A\) is a non-empty set of unique arcs 
\[\{a_1, a_2, \ldots, a_m\}\].
F is a function from the set A of arcs to the set $\mathbb{N} \times \mathbb{N}$.

$F: A \rightarrow \mathbb{N} \times \mathbb{N}$

If, for a given ordered pair $(n_i, n_j)$, there are $k$ distinct arcs in $A$ for which $F(a) = (n_i, n_j)$ then there are $k$ arcs from node $n_i$ to node $n_j$ and the graph is called a multigraph. Figure 3.1 is an example of a multigraph.

Example

$$\begin{array}{c}
\text{a} \\
n_2 \\
n_3 \\
\end{array}$$

Figure 3.1. A multigraph

For a given arc $a = (n_i, n_j)$, the node $n_i$ is called the initial node and $n_j$ the final node. A simple graph is a multigraph where the function, $F$, is one-to-one.

A Path, $P$, of graph $G$ is a sequence of arcs $(a_1, a_2, \ldots a_\ell)$ such that for every pair $(a_i, a_{i+1})$ of consecutive arcs in $P$ the final node of the arc $a_i$ and the initial node of the second arc $a_{i+1}$ coincide.

A finite sequence of arcs $a_1, a_2, \ldots a_\ell$ is said to constitute a path of length $\ell$. The nodes $n_1$ and $n_{\ell+1}$ are said to be the initial and final nodes of the path respectively. If $n_1 = n_{\ell+1}$ then the path is said to be a cycle and if $a_1 = a_{\ell+1}$ but otherwise all the nodes in the path are distinct, then the
path is called a **proper cycle**. A path in which all the nodes on the path are distinct is a **proper path**.

Example:

\[
\begin{align*}
\{n_1, n_2, n_3\} & \text{ is a proper cycle} \\
\{n_4, n_5, n_6\} & \text{ is a proper path of length } = 2
\end{align*}
\]

![Graph with a cycle](image)

**Figure 3.2.** Graph with a cycle

In a cycle free (acyclic) graph, if there exists a proper path from \(a_i\) to \(a_j\)

then \(a_i\) is the **ancestor** of \(a_j\) and \\
\(a_j\) is the **descendant** of \(a_i\).

If the path length is 1 then \(a_i\) is the direct ancestor of \(a_j\) and \(a_j\) is the direct descendant of \(a_i\).

In Figure 3.2

\(n_6\) is a descendant of \(n_4\) and \\
\(n_6\) is the direct descendant of \(n_5\).

A node \(n_j\) is **reachable** from another node \(n_i\) in a graph \(G = <N, A, F>\) if \(n_i = n_j\) or there is a path from \(n_i\) to \(n_j\).

A subgraph \(G'\) of a graph \(G\) is a subset of the set of nodes of \(G\) together with all the arcs connected to these nodes.
Figure 3.3b shows a subgraph. Two subgraphs $G'$ and $G''$ of a program graph $G$ are said to be disjoint if they do not share a common node.

A transition sequence is a subgraph which contains the initial and final nodes of the graph such that the final node is reachable from the initial node.

Figure 3.3. A graph and an associated subgraph

Figure 3.4. A graph and two of its transition sequences
Figure 3.4b and c represent two transition sequences. If $\sigma$ is the transition sequence, then the length of the transition sequence $L(\sigma)$ is the number of nodes in $\sigma$. In Figure 3.4c, $L(\sigma) = 5$.

A path which contains no cycles is acyclic and a graph in which all paths are acyclic is an acyclic graph. The model used in this thesis is based on acyclic directed graph.

Background Material

Graphs have been used extensively to model computations in computer science. The best known among all the graphs is the flowchart which represents the sequential control in the computation. Even though flowcharts are an important branch of graph theory, the review presented here is for models which have a direct bearing on the study of concurrent processing. Some models are completely interpreted, some partially interpreted and others uninterpreted. Depending on the model, they are used to measure the performance, determinancy, proper termination, and equivalence or other characteristics of multiprocessing systems.

Karp and Miller's [17] model, called the computation graph, is a directed graph in which nodes denote operations and branches represent storage elements. The storage elements are first-in-first-out queues. Each branch is assigned four non-negative integers which define the execution rule for a
node in the graph. Therefore, the computation represented by the computation graph is deterministic.

The directed acyclic bilogic graph [3, 10] was developed to represent computations on a variable structure computer. The nodes in the graph represent statements in the program and the arcs represent the precedence among the statements. Each node is assigned either a EOR or AND type logic. A legal graph in this model has a unique initial and terminal vertex. This model is used primarily to determine an a priori assignment and sequencing of computations in a multiprocessor system.

The program graph [31] is a direct extension of the previous model. This model is completely interpreted where the nodes represent operations and the arcs represent the storage elements and transmission of data and/or control. Each of the arcs on a node carry control information and, depending on the information residing in each of the arcs incident into and out of the node, the computation is initiated. The graph is formed by interconnecting the corresponding control and data arcs of the nodes correctly. The program graph represents a deterministic computation. Necessary and sufficient conditions to prevent execution ending in a deadlock state, along with transformations to get an equivalent graph for the program graph is also presented.

Among other graph theoretical models, the directed acyclic
graph [34] is used for automatic sequencing of processes on a given number of autonomous processing units. In the event of the number of processes being greater than the available number of processors, then the process with the largest number of successors is processed first. If all of the processes have the same number of successors, then the tie is broken in favour of the process with the smallest execution time.

Acyclic Directed Graph for Model-I Architecture

A model for computation processes called timed program graph (TPG) is presented in this section. The timed program graph explicitly indicates the data dependency among the different parts of the process in a natural way. The model I architecture is a feedback data flow processor capable of operations on elementary data types and individual structure components [7, 26].

Machine level graph representation

The data flow program is represented as a timed program graph which is a variation of the graph model introduced by Martin and Estrin [23, 24]. The nodes in the timed program graph represent the basic operations supported by the data flow processor described in Chapter II and the arcs represent the data dependency among the operations in the data flow program.

All non-trivial programs have iterative constructs which introduce cycles in the graph model. The presence of
cycles in the graph increases the complexity of estimating the transition sequences compared to an acyclic directed graph. The cycles can be eliminated by transforming the cyclic graph to a transitive directed graph using the transformations presented in [23, 24]. For a well-formed program, the transitive directed graph is formed by removing the cycle and modifying the execution time of all the nodes inside the cycle by a multiplicative factor corresponding to the number of times the cycle will be traversed during program execution. In high level programs, the number of times the loop executes and the probability of the conditional taking one branch or another is dependent on input data or values calculated during the execution of the program. Therefore assigning values to these parameters during the estimation process demands an a priori knowledge of the values of the parameters. In the timed program graph model, the above parameters are assigned variable names. Also, the nodes in the graph are assigned execution time, where the execution time is the time to execute a given operation normalized with respect to the add function time. Any node in the graph representing the synchronization operation or an operation for which there is no corresponding operation in sequential processing is either eliminated or assigned an execution time of zero. The resulting timed program graph, $G$, can be characterized by a four-tuple.
\[ G = \langle N, A, T, B \rangle \]

where

- \( N \) is a non-empty set of unique nodes
  \( \{n_1, n_2, \ldots, n_k\} \)
- \( A \) is a non-empty set of unique arcs
  \( \{a_1, a_2, \ldots, a_m\} \)
- \( F \) is a function from set \( A \) of arcs to the set \( N \times N \)
  \( F: A \rightarrow N \times N \)
- \( T \) is a function from set \( N \) of nodes to the set \( \{0, 1, 2, \ldots\} \) the set of natural numbers
  \( T: N \rightarrow \{0, 1, 2, \ldots\} \)
- \( B \) is a function assigning branching probabilities to the arcs
  \( B: A \rightarrow \{x: 0 \leq x \leq 1\} \)

A few examples are presented to illustrate the above principles. A segment of high level program and the corresponding program graph is shown in Figure 3.5. Notice that the program graph for this program segment is similar to the data flow program representation of Dennis [5] except that input is modeled as a basic machine operation. The machine level program graph models program execution on a feedback data flow processor without introducing any error in the measurement. For large programs the gain in accuracy of the analysis may outweigh the amount of work introduced in terms of the number of nodes and paths in the graph.
s1 input a, b, c, d file = f format = 4F(6)
s2 y := a*a + b+b;
s3 g := a*b;
s4 v := y*y*(g+b);
s5 u := v*v + y/c;
s6 w := u*d;

a) Segment of High Level Program

b) Machine Level Program Graph

Figure 3.5. Machine level and approximate program graph representation
c) Approximate Timed Program Graph

\[ \tau_s(G') = 23, \quad \tau_p(G') = 11 \]

\[ IP(G') = \frac{23}{11} = 2.1 \]

Figure 3.5 (Continued)
For the program graph in Figure 3.5b, the parallel execution time, assuming an execution time of all operators to be unity, is the maximum of the ten transition sequences and the sequential execution time is the sum of the operation time of the operators in the program graph. The ten transition sequences produced are:

\[
\begin{align*}
\sigma_1 &= 1, 2, 4, 5, 7, 8, 10, 11, 24 \\
\sigma_2 &= 1, 2, 4, 5, 7, 8, 10, 12, 23, 24 \\
\sigma_3 &= 1, 2, 4, 5, 7, 9, 19, 22, 23, 24 \\
\sigma_4 &= 1, 2, 4, 6, 14, 16, 20, 21, 22, 23, 24 \\
\sigma_5 &= 1, 2, 4, 6, 16, 20, 21, 22, 23, 24 \\
\sigma_6 &= 1, 2, 4, 6, 15, 17, 19, 22, 23, 24 \\
\sigma_7 &= 1, 2, 4, 6, 15, 17, 18, 20, 21, 22, 23, 24 \\
\sigma_8 &= 1, 2, 3, 14, 16, 20, 21, 22, 23, 24 \\
\sigma_9 &= 1, 2, 3, 13, 17, 19, 22, 23, 24 \\
\sigma_{10} &= 1, 2, 3, 13, 17, 18, 20, 21, 22, 23, 24
\end{align*}
\]

The parallel execution time \((T_p) = \max \{|\sigma_1|, |\sigma_2|, \ldots, |\sigma_{10}|\} = |\sigma_7|\).

That is, \(T_p = \text{sum of execution times of nodes in } \sigma_7 = 11\) and \(T_s = 23\). Note that node 24 is a unique terminal node added to satisfy the definition of transition sequence and it does not have a counterpart in high level programs and hence is assigned an execution time of zero. Even though the program graph in
Figure 3.5 is a multigraph, the timed program graph is a simple graph with the multiplicity eliminated. To obtain the transition sequence and hence the longest path in the graph, path algorithm may be used, after producing a consistent labeling of the graph using Marimont's algorithm [28]. The execution times of these algorithms are of order \((n + a)\), where \(n\) is the number of nodes and \(a\) is the number of arcs in the graph. For the program in Figure 3.5 the intrinsic parallelism (IP) is the ratio \(\tau_s(G)/\tau_p(G)\), which is 2.1.

**Approximate graph representation**

The machine level graph representation does not introduce any errors in the calculation of intrinsic parallelism if the branching probabilities and the loop execution parameters are known a priori. But the amount of work involved may increase rapidly as the program size increases, which may make the cost of estimation prohibitive. A significant reduction in the work can be achieved by reducing the number of paths in the program graph. This reduction in the number of paths can be obtained by defining a new set of nodes \(N'\) which represent expressions, assignment statements, procedure invocations and input/output processing implied by individual components in I/O lists as they appear in the high level program. The set of arcs \(A'\) now denote data dependencies among nodes \(N'\). Figure 3.5c illustrates this resulting approximate timed directed graph \(G' = <N', A', T, B>\) associated with the program.
segment in Figure 3.5a.

Calculation of parallel execution time for the program segment now requires traversal of only 15 arcs connecting 9 nodes compared to 30 arcs connecting 24 nodes. The sequential and parallel execution times assigned to an individual node appear as \((\tau_s : \tau_p)\) beside the node on the program graph. The sequential execution time of a node is calculated as the sum of the execution times of the implied base machine operations. The parallel execution time of a node is calculated as the parse tree height, produced by the compiler, based on execution times of the base machine operations.

**Graph representation and timing of high level constructs**

In the approximate program graph model the nodes represent more than one base machine operation. Therefore the node execution time will depend on the type of high level construct it represents. The execution time is formed as a function of the number of loop iterations and conditional path probabilities for well-formed high level constructs. Throughout this section, \(\tau_s(name)\), stands for the sequential execution time of the operation represented by the name and \(t_{name}\) represent the execution time of the base machine operation. The high level constructs and their execution times are given below:
1. Assignment Statement (AS)

**high level**

\[ V := E \]

**graph representation**

\[
\begin{array}{c}
\ldots \\
\text{AS} \\
\ldots \\
\end{array}
\]

\[
\text{inset (AS)} = \text{inset (V)} \cup \text{inset (E)}
\]

\[
\text{outset (AS)} = \text{outset (V)}
\]

**Execution Time**

**sequential**

\[ \tau_s (AS) = \text{sum of machine level operations} \]

**parallel**

\[ \tau_p (AS) = \text{height of parse tree} \]

**Example**

\[ D = A \times B + Q/C \]

Assuming unit execution time for all the operations,

\[ \tau_s (AS) = 3 \]

\[ \tau_p (AS) = 2 \]

2. Conditional (C)

**high level**

\[ \text{if } P \text{ then } B \]
inset (C) = inset (P) U inset (B)
outset (C) = outset (B)
\( \alpha_1, \alpha_2 \) is the branching probabilities of generating a true or false value.

**Execution Time**

**sequential**
\[ \tau_s(C) = \tau_s(P) + \alpha_1 \cdot \tau_s(B) \]

**parallel**
\[ \tau_p(C) = \tau_p(P) + \alpha_1 \cdot \tau_p(B) \]

Here B stands for the body of the conditional which may contain one or more high level constructs.

*if P then B_1 else B_2*
Graph

\[
\text{inset (C)} = \text{inset (P)} \cup \text{inset (B)} \cup \text{inset (B_1)} \\
\text{outset (C)} = \text{outset (B)} \cup \text{outset (B_1)}
\]

\(\alpha_1, \alpha_2\) are the branching probabilities

**Execution Time**

**sequential**

\[
\tau_s(C) = \tau_s(P) + \alpha_1 \tau_s(B_1) + \alpha_2 \tau_s(B_2)
\]

**parallel**

\[
\tau_p(C) = \tau_p(P) + \alpha_1 \tau_p(B_1) + \alpha_2 \tau_p(B_2)
\]

**Example**

\[
\text{if } A > E \text{ then } Q := (A+D) \cdot E \\
\text{else } Q := (D-A) \cdot E
\]

\[
\tau_s(P) = \tau_p(P) = 1 \\
\tau_s(B_1) = \tau_p(B_1) = 2 \\
\tau_s(B_2) = \tau_p(B_2) = 2
\]

\[
\tau_s(C) = \tau_p(C) = 1 + 2 \cdot \alpha_1 + 2 \cdot \alpha_2
\]
\[ \tau_s(C) = \tau_p(C) = 1 + 2 \cdot \alpha_1 + 2 \cdot \alpha_2 \]

3. Repetition (R)

**high level**

```
while P do B
```

**Graph**

```
inset (R) U outset (R)
```

```
inset (P) U inset (B)
```

```
outset (R) = outset (B)
```

**Execution Time**

**sequential**

\[ \tau_s(R) = (N_R + 1) \tau_s(P) + N_R \tau_s(B) \]

**parallel**

\[ \tau_p(R) = (N_R + 1) \tau_p(P) + N_R \tau_p(B) \]

**Example**

```
s1 while I <= N do
s2 J := (W+Q) + (L*L);
s3 I := I+1
end;
```
\[ \tau_s(P) = \tau_p(P) = 1 \]
\[ \tau_s(S_2) = 3 \]
\[ \tau_p(S_2) = 2 \]
\[ \tau_s(S_3) = \tau_p(S_3) = 1 \]

The timed program graph is the graph resulting after applying the transitive transformation which eliminates cycles.

**high level**

\[ \text{repeat } B \text{ until } P \]

**Graph**

\[ \text{inset (} P \text{) U outset (} R \text{)} \]

\[ \text{inset (} R \text{) = inset (} B \text{) U inset (} P \text{)} \]

\[ \text{outset (} R \text{) = outset (} B \text{)} \]

**Execution Time**

**sequential**

\[ \tau_s(R) = N_R (\tau_s(B) + \tau_s(P)) \]

**parallel**

\[ \tau_p(R) = N_R (\tau_p(B) + \tau_p(P)) \]

**Example**

\[ \text{repeat} \]
\[ s2 \quad J := (J+Q) + (W*W); \]
\[ s3 \quad I := I+1 \]
\[ \text{until } I > N; \]
Non-recursive Procedure

This section on procedure consists of procedure definition and procedure application.

Procedure Definition (Proc-Def)

**high level**

\[
\text{procedure } \text{Proc.name} \left( \text{in}(I_1, \ldots, I_m), \text{out}(O_1, \ldots, O_k) \right)
\]

**Graph**

\[
\begin{align*}
I_1 & \quad \ldots \quad I_m & \text{inset (Proc-Def)} \\
\text{Proc-Def} & \quad & \\
O_1 & \quad \ldots \quad O_k & \text{outset (Proc-Def)}
\end{align*}
\]

inset (Proc-Def) = \{I_1, \ldots, I_m\}

outset (Proc-Def) = \{O_1, \ldots, O_k\}

The node Proc-Def in the graph represents the body of the procedure definition formed from the high level constructs of the target language, the prologue to select the actual parameters from input structure to initiate the procedure body and the epilogue to form the result structure to be sent back to the calling
procedure. The number of machine operations to select the actual input parameters is a function of the number of actual parameters specified in the `in` portion of the procedure definition. In general, the number of select operation is, $|\text{in}(I_1, \ldots, I_m)|$, equal to $m$, where $m$ is the number of input parameters. Similarly the number of append machine operations required to form the result structure is, $|\text{out}(O_1, \ldots, O_k)|$, equal to $k$, where $k$ is the number of `out` parameters in the procedure definition.

**Execution Time**

**sequential**

\[
\tau_s(\text{prologue}) = m \cdot t_{select} + t_{identity}
\]

\[
\tau_p(\text{prologue}) = t_{select} + t_{identity}
\]

where the identity operator is the one which starts the prologue of the procedure definition. The corresponding time for appending results to the output structure and returning the value to the calling program (epilogue) are:

\[
\tau_s(\text{epilogue}) = \tau_p(\text{epilogue}) = k \cdot t_{append} + t_{identity}
\]

Note that $t_{select}$, $t_{append}$ and $t_{identity}$ correspond to the time required to execute the base machine operations select, append and identity. Therefore the time to execute procedure definition is given by
sequential
\[ \tau_s(Proc-Def) = \tau_s(Prologue) + \tau_s(B) + \tau_s(Epilogue) \]

parallel
\[ \tau_p(Proc-Def) = \tau_p(Prologue) + \tau_p(B) + \tau_p(Epilogue) \]

**Procedure Application** (Proc -Apply)

**high level**

Proc.Name \( \langle in(I_1, \ldots, I_m), out(O_1, \ldots, O_k) \rangle \)

**Graph**

\[ \begin{array}{c}
I_1 \ldots I_m \\
\text{inset (Proc-Apply)} \\
\text{Proc-Apply} \\
O_1 \ldots O_k \\
\text{outset (Proc-Apply)}
\end{array} \]

\[ \text{inset (Proc-Apply)} = \{I_1, \ldots, I_m\} \]
\[ \text{outset (Proc-Apply)} = \{O_1, \ldots, O_k\} \]

In the graph, the node Proc-Apply represents the formation of the actual procedure parameters required to initiate the procedure definition, the execution of the procedure definition once the procedure is initiated and the distribution of the results of the procedure execution to successor instructions signaling the termination of the procedure application.

**Execution Time**

The procedure initiation consists of forming the input structure with the input parameters and the execution time is \( \tau_s(\text{initiate}) = \tau_p(\text{initiate}) = m \)
$t_{append} + t_{apply}$. Similarly, the procedure termination time $\tau_s(terminate) = k \cdot t_{select} + t_{identity}$ and $\tau_p(terminate) = t_{select} + t_{identity}$. Therefore, the procedure apply time is,

**sequential**

\[
\tau_s(Proc\text{-}Apply) = \tau_s(initiate) + \tau_s(Proc\text{-}Def) + \tau_s(terminate)
\]

**parallel**

\[
\tau_p(Proc\text{-}Apply) = \tau_p(initiate) + \tau_p(Proc\text{-}Def) + \tau_p(terminate)
\]

5. **Input/Output (I/O)**

**high level**

\[
\text{input/output } V_1, V_2, \ldots, V_m \text{ file = file-name} \]

\[
\text{format } = (f_{1^1}, \ldots, f_{k_1}, \ldots, f_{1^m}, \ldots, f_{k_m})
\]

The high level input/output statement may consist of a combination of elementary and structured data values along with control formats. The format list consists of $m$ data specification formats and the rest are control formats. The high level input/output statement is represented as a set of low level read/write operations corresponding to each occurrence of a $V_i$ in the input/output list. Note that it is possible to read structured data using an implied repetition construct and the execution time for
reading a set of data will be treated later on. Also there is a set of control format primitives called readedit/writedit corresponding to the control formats in the format list.

Since the feedback data flow architecture supports only sequential file operations each read/write operation is sequenced on the presence of the file. Also, each member in the input list is represented as a node in the graph model, except when the operation is on structured data. The graph model and the execution time for file primitives are illustrated below:

**Control Formats**

readedit file = file-name format = \( f^i_j \)

**Graph**

\[ \text{inset (Readedit)} = \{\text{file-name, } f^i_j\} \]

Readedit

\[ \text{outset (Readedit)} = \{\text{file-name}\} \]

**Execution Time**

\[ \tau_s(\text{Readedit}) = \tau_p(\text{Readedit}) = t_{\text{readedit}} \]

writedit file = file-name format = \( f^i_j \)

**Graph**

\[ \text{inset (Writedit)} = \{\text{file-name, } f^i_j\} \]

Writedit

\[ \text{outset (Writedit)} = \{\text{file-name}\} \]
Execution Time

\[ \tau_S(\text{Writedit}) = \tau_P(\text{Writedit}) = t_{\text{writeit}} \]

If the \( V_i \) in the input/output list represents an elementary value, then a single \( f_j \) from the format list is associated with the elementary value to form the read/write machine level file operation.

\[ \text{read } V_i \text{ file.} = \text{file-name} = F \text{ format} = f_j \]

Graph

inset (Read) = \{file-name, \( f_j \)\}

outset (Read) = \{file-name, \( V_i \)\}

Execution Time

The read operation removes characters from the head of the input file and converts them to an internal data type according to the specified format. The read operation forms a structure with the data value and the file as components. This is followed, at the machine level, by operations to select the data value and the file name for further processing. Therefore the time to perform the read function is:

**sequential**

\[ \tau_S(\text{Read}) = t_{\text{read}} + 2 * t_{\text{select}} \]

**parallel**

\[ \tau_P(\text{Read}) = t_{\text{read}} + t_{\text{select}} \]
write \( V_i \) \( \text{file} = \text{file-name} \) \( \text{format} = f_j \)

Graph

inset (Write) = \{file-name, \( V_i \)\}

outset (Write) = \{file-name\}

**Execution Time**

The write operation takes a data value, a file and a format as its input parameters and copies the data value on to the file according to the specified format. After the write operation is over the updated file is passed on to the next output operation. Therefore the execution time for write operation is:

\[ \tau_s(\text{Write}) = \tau_p(\text{Write}) = t_{\text{Write}}. \]

If \( V_i \) in the input/output list represents a structure, then the general form is \( (V_i, I_1, I_2, \ldots, I_n) \) \( I_1 = E_{11} \) to \( E_{12}, \ldots, I_n = E_{n1} \) to \( E_{n2} \) with an associated format of the form \( f_{1}^{i}, f_{2}^{i}, \ldots, f_{k_{i}}^{i} \). In the format list one \( f_{j}^{i} \) is a data format and remaining \( (k_{i}-1) \) are control formats. The input/output of the above form implies \( n \) nested loops of form 3 where the body \( B \) now consists of a read and zero or more readedits or, a write and zero or more writedits along with necessary structuring (append/select)
operations.

Example

input A, B, C file = inf format = F(8,4), F(8,5), F(8,7)

Graph

output A, B, C file = opf format = F(8,6), F(8,5), F(8,7)

Graph
input \((A(I) \ I = 1 \text{ to } N)\) file = inf format = F(5,2),

skip

Graph

In the graph the nodes represent:

1. the predicate
2. increment of loop index
3. read
4. append to form structure A
5. readedit.

6. Synchronization Constructs

Fork \((F)\)

To represent initiation of concurrent activities in the graph model, a fork node is introduced to indicate the availability of operands. Zero execution time is
assigned to the fork operator, because there is no equivalent operation in the high level language for sequential processing.

Graph

\[
\text{inset (Fork)} = \{\text{Union of the outsets of all the immediate predecessor nodes}\}
\]

\[
\text{outset (Fork)} = \text{inset (Fork)}
\]

\[
\tau_s(\text{Fork}) = \tau_p(\text{Fork}) = 0
\]

Join (J)

The join node is the inverse of the fork construct and models the collection of outsets of concurrently executing predecessor nodes of the graph model. The join node is used in repetitive construct to enforce the feedback effect of the data flow processor and also when more than one node is active simultaneously in each branch of the conditional.

Graph

\[
\text{inset (Join)}
\]

\[
\text{outset (Join)}
\]
inset (Join) = \{\text{Union of the outsets of all the concurrently active immediate predecessor nodes}\}

\text{outset (Join) = inset (Join)}

\text{Execution Time}
\[
\tau_s(\text{Join}) = \tau_p(\text{Join}) = 0
\]

\text{OR}

The OR node is used to select one of two alternative insets in a conditional construct.

\text{Graph}

\begin{center}
\begin{tikzpicture}
  \node[draw,circle] (OR) at (0,0) {OR};
  \node[draw,circle] (inset) at (0,-2) {inset (OR)};
  \node[draw,circle] (outset) at (0,-4) {outset (OR)};
  \draw[->] (OR) -- (inset);
  \draw[->] (inset) -- (outset);
\end{tikzpicture}
\end{center}

\text{inset (OR) = \{Outset of lefthand node or outset of righthand node\}}

\text{outset (OR) = inset (OR)}

\text{Execution Time}
\[
\tau_s(\text{OR}) = \tau_p(\text{OR}) = 0
\]

\text{Performance Analysis}

A timed program graph, \(G' = <N', A', T, B>\), of a given high level program can be formed using the graph representation of the previous section. The intrinsic parallelism (IP) in the program can be approximated by the ratio of the sequential to the
parallel characteristic timing equations, $\tau_s(G')$ and $\tau_p(G')$, respectively. The characteristic timing equations are functions of the program structure, the underlying processor organization, and the type of transformations, if any, applied to the original program. The effect of the applied transformation can be incorporated by forming a new timed program graph for each transformation applied to the program. The following assumptions, which reflect the effect of processor organization on the estimation of characteristic equation, are made in estimating the performance characteristics.

1. Adequate computing resources are available to avoid resource contention.
2. The execution time used in the estimation of characteristic equation includes the operation execution time and fetch time for operands.
3. The memory contention between operations executed in parallel in fetching operands and storing results are ignored.
4. The time taken to fetch the instruction, decode and the time spent in communication channels are neglected.
5. The operation time of synchronizing operators are made zero to eliminate their effect on intrinsic parallelism.
Based on these assumptions, a procedure to estimate the sequential and parallel execution times is presented. The sequential and the parallel execution times calculated from the timed program graph is called the characteristic timing equations. The presence of data dependent predicate operations introduce interminancy as to which branch is to be taken at execution time. This problem is overcome by assigning unique branching probability variables to each conditional construct in the high level program. The iteration parameter is handled in a similar fashion by assigning loop parameter variable to the high level iteration constructs. Therefore, the final characteristic equations appear as a function of the loop parameters, branching probabilities and the maximum function which selects the path with maximum execution time among concurrent transition sequences between any two given nodes in the timed program graph. To estimate the longest transition sequence, Marimont's algorithm may be used to achieve a consistent labeling before using the longest path algorithm [28]. The longest transition sequence defines the characteristic timing equation for parallel execution \((\tau_p(G'))\).

The characteristic timing equation for sequential execution \((\tau_s(G'))\) is the sum of the node execution
times of the timed program graph. The intrinsic parallelism, \( \text{IP}(G') = \tau_s(G')/\tau_p(G') \), is a function of the branching probabilities and the number of iterations.

Figure 3.6 shows an example of a high level program, its timed program graph and the characteristic timing equations for high level program below:

1. \( I := 1 \);
2. \( \textbf{while } I <= N \textbf{ do} \)
3. \( V(I) := A(I) * B(I); \)
4. \( Q(I) := V(I) * C(I); \)
5. \( J := 1; \)
6. \( \textbf{while } J <= N \textbf{ do} \)
7. \( E(I,J) := F(I,J) + K(I,J); \)
8. \( D(I,J) := E(I,J) * 10.0; \)
9. \( J := J + 1 \)
10. \( \textbf{end}; \)
11. \( I = I+1 \)
12. \( \textbf{end}; \)

Observe that the node, \( F \), representing a fork operation signifies the concurrency among the different parts of the program and also the operator join represents the effect of the underlying feedback data flow processor, by not allowing the initiation of the next iteration until the first initiation terminates.
Figure 3.7 shows the tree formed for statements 3 and 7 in the high level program with the machine operation times in parentheses. The parallel and the sequential execution time for node 3 and 7 are also shown in Figure 3.7. The execution time for node 3 and 7 are multiplied by the iteration parameters in Figure 3.6 to form the timed program graph. In the timed program graph, the node execution times are presented as \((\tau_s : \tau_p)\). The sequential characteristic timing equation, \(\tau_s(G')\), is the sum of the node execution times of the timed program graph and the parallel characteristic timing equation, \(\tau_p(G')\), is the maximum of the transition sequences between initial node 1 and the terminal node J.

\[
\tau_p(G') = \max \{1 + (4 N_1+1) + 4 N_1, 1 + 4(N_1+1) + 8 N_1 + 8 N_1, 1 + (4 N_1+1) + N_1 + 4 N_1(N_2+1) + 4 N_1 N_2, 1 + (4 N_1+1) + N_1 + 4 N_1(N_2+1) + 4 N_1 N_2, 1 + (4 N_1+1) + N_1 + 4 N_1(N_2+1) + 16 N_1 N_2 + 18 N_1 N_2\}
\]

\[
= 5 + 4 N_1 + \max \{16 N_1, 5 N_1 + 38 N_1 N_2\}
\]

A single figure of merit for \(IP(G') = 73/38\) is obtained by letting all \(N_i\) approach infinity at an equal rate in the ratio for \(IP(G')\). This value is a measure of the parallelism under the assumption that the innermost loop totally dominates the computation. A more accurate estimate of the parallelism can be obtained through intelligent assignment of values to \(N_i\).
b) Characteristic timing equations for high level program

\[ \tau_s(G') = 5 + 13N_1 + 73N_1N_2 \]

\[ \tau_p(G') = 5 + 4N_1 + \max\{16N_1, 5N_1 + 38N_1N_2\} \]

\[ \lim IP(G') = 73/38 \]

Figure 3.6. Performance analysis for a sample program
$\tau_s(3) = t_{\text{append}} + t_{*} + t_{\text{select}} + t_{\text{select}}$

$= 1 + 6 + 1 + 1 = 9$

$\tau_p(3) = t_{\text{append}} + t_{*} + t_{\text{select}}$

$= 1 + 6 + 1 = 8$

$\tau_s(7) = t_{\text{append}} + t_{*} + t_{+} + t_{+} + (t_{\text{select}} + t_{*} + t_{+}) \times 2$

$= 1 + 6 + 4 + 4 + (1 + 6 + 4) \times 2 = 37$

$\tau_p(7) = t_{+} + t_{\text{select}} + t_{*} + t_{+}$

$= 4 + 1 + 6 + 4 = 16$

**Figure 3.7. Parse tree**
and \( \alpha_i \).

The actual parallelism and the predicted parallelism for a sample of 25 Fortran and Algol programs selected from a variety of sources is shown in Figure 3.8. The actual parallelism (actual IP) is obtained by executing the program for a given set of input data on the data flow simulator. The predicted parallelism (predicted IP) was found from the function \( \text{IP}(G') \) by letting the number of loop iterations approach infinity and selecting the longest transition sequence (assigning 0 or 1 to the branching probabilities) based on the machine operation times given in Table 2.4. While the numeric values in Figure 3.8 cannot, in any sense, be regarded as absolute, they are highly indicative of the parallelism available in high level programs for the feedback data flow processor. The single figure of merit, as predicted parallelism, is in many cases, extremely close to the actual parallelism. In 15 of the 25 cases, the error is less than 10\% and in 22 of 25 cases, the error is less than 20\%. This indicates that the characteristic timing equations produced at compile time are highly useful in approximating parallelism in real programs.

Graph Representation for Model II Architecture

Parallelism in model I architecture is due to the presence of scalar concurrency among statements and operations. A potentially large factor of parallelism has been ignored since arrays have been viewed as structures (rather than streams of
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Actual IP</th>
<th>Predicted IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEWB</td>
<td>1.76</td>
<td>1.54</td>
</tr>
<tr>
<td>PRPF</td>
<td>3.24</td>
<td>2.55</td>
</tr>
<tr>
<td>MLTP</td>
<td>2.12</td>
<td>2.23</td>
</tr>
<tr>
<td>GRUP2</td>
<td>1.77</td>
<td>1.62</td>
</tr>
<tr>
<td>BP3</td>
<td>3.39</td>
<td>2.19</td>
</tr>
<tr>
<td>BP6</td>
<td>1.85</td>
<td>1.37</td>
</tr>
<tr>
<td>SSP2</td>
<td>1.67</td>
<td>1.72</td>
</tr>
<tr>
<td>CUBN</td>
<td>2.21</td>
<td>2.12</td>
</tr>
<tr>
<td>PLY</td>
<td>2.11</td>
<td>2.12</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>2.84</td>
<td>2.36</td>
</tr>
<tr>
<td>RTD2</td>
<td>2.47</td>
<td>2.51</td>
</tr>
<tr>
<td>SPDI</td>
<td>2.60</td>
<td>2.61</td>
</tr>
<tr>
<td>EXSM</td>
<td>2.38</td>
<td>2.03</td>
</tr>
<tr>
<td>SE35</td>
<td>1.52</td>
<td>1.69</td>
</tr>
<tr>
<td>MEBS</td>
<td>1.89</td>
<td>2.00</td>
</tr>
<tr>
<td>DGT3</td>
<td>2.29</td>
<td>1.87</td>
</tr>
<tr>
<td>JELF</td>
<td>2.11</td>
<td>2.10</td>
</tr>
<tr>
<td>MVS</td>
<td>2.22</td>
<td>2.28</td>
</tr>
<tr>
<td>QSF</td>
<td>1.74</td>
<td>1.59</td>
</tr>
<tr>
<td>SIMQ</td>
<td>2.08</td>
<td>1.81</td>
</tr>
<tr>
<td>MXRA</td>
<td>1.56</td>
<td>1.37</td>
</tr>
<tr>
<td>MEST</td>
<td>1.81</td>
<td>1.74</td>
</tr>
<tr>
<td>QATR</td>
<td>2.89</td>
<td>2.78</td>
</tr>
<tr>
<td>PECR</td>
<td>2.67</td>
<td>2.54</td>
</tr>
<tr>
<td>ELII</td>
<td>2.40</td>
<td>2.52</td>
</tr>
</tbody>
</table>

Figure 3.8. Measured and predicted parallelism for a sample of 25 programs

scalars). A measure of this additional parallelism can be approximated by introducing vector functional units within the context of a data flow processor. This is
only approximate since chaining of vector operations is ignored. An additional increase in intrinsic parallelism can be given when vector operations are applied even to vectors of short length [16]. To get an initial feel for the intrinsic parallelism when vector operations are present, estimates of intrinsic parallelism in high level programs using the model II architecture are obtained. The model II architecture is an enhancement of model I in that a vector processing capability is assumed in addition to scalar processing.

Compiler considerations

The enhancement of the processing capabilities by adding vector function unit implies that the compiler has to be intelligent enough to recognize vector operations and produce corresponding data flow code. In the context of this chapter, the recognition and code generation of vector operations are not directly relevant and hence the principle used in vectorizing high level program loop by decomposition [22] only is reviewed. It is possible to increase intrinsic parallelism by applying more vectorizing transformations [20] to the high level programs. But this line of action is not pursued here.

In order to obtain the characteristic timing equations for model II architecture, any loop involving array computation is examined for iteration independence and if it is, then the loop is transformed into a parallel construct. If the loop is iteration dependent an attempt is made to eliminate dependency
by forward substitution, introduction of temporary array names, and conversion of scalers into vectors. After applying the above transformations if the data dependency is removed, then the loop is decomposed without violating the precedence constraints and the analysis is repeated on each partition, otherwise the loop is unsuitable for vector processing.

Two examples are given to illustrate the vectorizing techniques. The high level program segment shown below cannot be vectorized directly because statement 4 is data dependent on statement 3.

1. I := 2;
2. while I <= N do
3. B(I) := A(I) + C(I+1);
4. C(I) := B(I+1);
5. I := I+1
6. end;

By assigning the values in vector B to another vector, the data dependency between statement 3 and 4 is eliminated. Now the loop can be vectorized as shown below.

1. DO PAR I = 2 to N
2. D(I) := B(I);
3. B(I) := A(I) + C(I+1);
4. C(I) := D(I+1);
5. END;
The next example illustrates how loop decomposition can be used to gain speedup in program execution in a feedback data flow processor.

1. I := 1;
2. while I <= N do
3.   A(I) := 1.0;
4.   K := 1;
5.   while K <= N do
6.     B(I,K) := 0.0;
7.     K := K + 1
8.   end;
9.   I := I + 1
10. end;
11. J := 1;
12. while J <= N do
13.   A(J) := A(J) * C(J);
14.   J := J + 1
15. end;

The above program segment can be vectorized without applying any transformation at all. But, the loop consisting of statements 12 to 15 has to wait for the arrival of the vector A. By decomposing the doubly nested loop as shown below, it is possible that the loop (9-11) may execute in parallel with the doubly nested loop (4-8) in the event of loop (1-3) producing a result earlier than the doubly nested
loop.

1. DO PAR I = 1 to N
2. A(I) := 1.0
3. END
4. DO PAR I = 1 to N
5. DO PAR K = 1 to N
6. B(I,K) := 0.0
7. END
8. END
9. DO PAR J = 1 to N
10. A(J) := A(J)*C(J)
11. END

**Execution time for vector operations**

The execution time for vector operations depends on the complexity of the operation, the organization of the pipelined functional unit and the clock cycle of the processor and the vector length. The execution times of the vector functional units obtained are based on the following assumptions:

a. The control needed once the vector operation is initiated and the vector operation parameter specifiers are part of the processor.

b. The vector operation instruction carries the destination specifiers for vector parameters.

c. Calculation of effective address of operands, field
length, etc. are performed before the vector operation begins.

d. The vector operation parameters such as the starting location of the operands, the constant increment, the length of the vectors, etc. are held in registers.

The execution time for a vector operation on the basis of the above assumptions is, \( \tau^V = t_s^V + (N-1) t_w^V \) [30] where,

- \( t_s^V \) - is the set up and the flush time of the pipe
- \( t_w^V \) - is the maximum execution time of any segment in the pipeline
- \( N \) - is the length of the vector.

The setup time is the time taken to load vector operation parameters to registers from memory and the flush time includes the time to decode, address calculation, operand fetch and pairing of the first pair of operands until it is output. The factor \( t_s^V \) will have less significant effect on the vector operation execution time as the vector length increases.

**Performance analysis**

The performance analysis is again based on the approximate graph model except that the execution time of vector operation replaces the scalar operation time. Also, the nodes corresponding to the loop conditional and index increment no longer appear in the timed program graph, if vector operation replaces iteration construct. Figure 3.9 represents the high level
parallel version of the program on page 95, along with the timed program graph. The effect of limited vector functional units is modeled by multiplying the execution time of the node by a resource factor of the form \( \frac{N}{K^V} \), where \( N \) is the number of similar vector operations represented by the graph node and \( K^V \) is the number of available function units of type \( v \). The limited availability of vector functional units introduces a more complex heterogeneous scheduling problem, for which there is no known general solution, when data independent paths compete for the limited processing resources.

The timed program graph is a simple model, in that the full vector processing capabilities such as chaining of vector operations [16, 33] or streaming of scalars [38] are not included. The use of only a subset of the vectorizing transformations along with the assumed serialized execution of potentially competing operation in estimating the characteristic execution time gives a conservative estimate for the intrinsic parallelism for the model II architecture.

To obtain a single figure of merit from the characteristic timing equations for model II architecture, a simple vector capability model shown in Figure 3.10 is used. The setup time \( t_s^V \) and the maximum pipeline segment time \( t_w^V \) are obtained in terms of memory cycle time. Assuming \( N_1 = N_2 = 10, K^+ = K^- = 1, t_s^+ = t_s^- = 7, \) and \( t_w^+ = t_w^- = 2 \), the intrinsic parallelism for the program in Figure 3.9 is, \( IP(G^\prime) = \frac{t_s(G^\prime)}{t_p^V(G^\prime)} = 14 \). If
1. DO PAR I = 1 to N
2. V(I) := A(I)*B(I);
3. Q(I) := V(I)*C(I)
4. END PAR;
5. DO PAR I = 1 to N, J = 1 to Ng
6. E(I,J) := F(I,J) + K(I,J);
7. D(I,J) := E(I,J)*10.0
8. END PAR;

a) Parallel form of high level program

\[ t_s + (N_1-1)t^* \]

\[ t_s + (N_1-1)t^* \]

\[ \tau_s(G') = 5 + 13 N_1 + 73 N_1 N_2 \]

\[ \tau_p(V(G'')) = 2 \left( t_s + (N_1-1)t^* \right) + \left( \frac{N_1}{K^*} \right) \left( t_s + (N_2-1)t^* \right) \]

b) Graph representation and characteristic timing equations

Figure 3.9. High level program and its timed program graph for Model II Architecture
enough number of add and multiply vector functional units are assumed, to avoid any conflict, then the characteristic timing equations for the program in Figure 3.9 is $\tau_p(G') = \max \{2(t_s^* + (N_1 - 1) t_w^*), (t_s^* + t_s^* + (N_2 - 1)(t_w^* + t_w^*)\}$. Assuming $N_1 = N_2 = 10$, $t_s^* = t_w^* = 7$, and $t_s^* = t_w^* = 2$, $IP(G') = \frac{\tau_s(G')}{\tau_p(G')} = 152$. Even though this figure looks attractive, in reality the bandwidth and memory access conflict may limit the number of vector operations that can be executed concurrently. Figure 3.11 shows the dominant terms of the characteristic equation, $\tau_p^v(G')$, for four of the twenty-five sample programs which were found amenable to vectorization. Assuming $N_i = 10$, $K^v = 1$, $t_s^{as} = 1$, $t_s^v = 7$, and $t_w^v = 2$ for all $i$ and $v$, the intrinsic parallelism using the dominant terms range from 2.1 to 12.3.

By assuming vector functional units, we have attempted to capture a measure of the parallelism exploitable by streams of tokens in the data flow program. The measures obtained are probably conservative since we assumed no chaining of individual vector operations. On the other hand, streams of tokens would take advantage of maximum chaining since operations are at the scalar level.
a) Vector assignment: $t_s^{as} = 1, t_w^{as} = 2$

b) Vector operation: $t_s^{v} = 1 + 1 + n + 1 = n + 3$

$\quad t_s^{v} = 2$

Figure 3.10. Model of simple vector capability
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Dominant Term of $T_s(G)$</th>
<th>Dominant Term of $T_P^v(G)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLY</td>
<td>$34N_1N_2N_3$</td>
<td>$N_1 \left{ \frac{N_3}{K^+} \right} + (t_s^+ + (N_2-1)t_w^+) \left{ \frac{N_3}{K^+} \right}$ + $(t_s^+ + (N_2-1)t_w^+) \left{ \frac{N_3}{K^\prime} \right}$</td>
</tr>
<tr>
<td>SPDI</td>
<td>$19N_1N_2 + 36N_3N_4$</td>
<td>$(t_s^+ + (N_1-1)t_s^+) \left{ \frac{N_2}{K^s} \right} + (t_s^+ + (N_3-1)t_3^+) \left{ \frac{N_4}{K^\prime} \right}$</td>
</tr>
<tr>
<td>MLTP</td>
<td>$58N_1N_2$</td>
<td>$27N_0 + t_w^+ (N_1+N_2+N_4+N_5) + t_w^+ (2N_1+N_4+N_5) + t_w^{as} (N_3+N_4)$</td>
</tr>
<tr>
<td>CUBN</td>
<td>$187N_1$</td>
<td>$86N_1 + 2N_1 t_w^-$</td>
</tr>
</tbody>
</table>

Figure 3.11. Dominant term of characteristic equations for four sample programs
Stream Read and Program Structure

There are applications where the program is executed for multiple sets of input data. In high level sequential programs, to input multiple sets of data are many times accommodated by enclosing the whole program is enclosed within an iterative construct. When this high level program is translated to data flow language and executed on a feedback data flow processor, the feedback inhibits the execution of the body of the loop for the next set of data until the first set of data is processed.

Within the feedback constraint, the program execution can be speeded up either by providing a copy of the program for each one of the input data set or by replacing the input statement by a stream input statement.

The first method is applicable only if the result produced does not depend on data values from other data sets. Also, memory requirements increase along with the increase in the number of data sets to be processed. On the other hand, the stream input construct divides the program structure into segments based on the natural feedback constraints imposed by the feedback data flow processor, although the stream input construct still operates on sequential file structure.

The introduction of stream input makes the program structure behave like a pipeline, where the segments are determined by the feedback effect. In some cases, the high level program may have to be modified, such as moving the
input statement outside the iteration construct, to make use of the stream input construct.

Semantically the stream input is similar to the input constructs except that it continuously obtains input data until the end of file is encountered and suitably routes the data values to their destinations.

The characteristic timing equation, \( \tau_p(GS) \), using the stream input is based on the assumption that the program structure behaves like a pipeline processor. Therefore,

\[
\tau_p(GS) = S + (D-1)p_w \quad \text{and} \quad \tau_s(GS) = D * \tau_s(G')
\]

where,

- \( D \) - is the number of data sets processed
- \( S = \sum_{i=1}^{n} p_i \)
- \( p_w = \max(p_1, p_2, \ldots, p_n) \)
- \( p_i \) - is the execution time of the \( i^{th} \) segment in the program
- \( n \) - is the number of segments in the program.

A high level program along with the timed program graphs with input and stream input constructs are as shown in Figure 3.12. The high level program for stream input is the same as Figure 3.11a except that the statement 7 is changed to

\[ \text{stream input } N, S, B, C, D \text{ file } = \text{ inf } \text{ format } = F(5,2), F(5,2), f(5,2), F(5,2), \text{ skip}. \]

The timed program graph, GS, is the
longest transition sequence of the timed program graph $G'$.

The node execution times in Figure 3.12 b and c are calculated assuming unit execution time for the base machine operations. In the timed program graph, $G_S$, the nodes inside the loop are reduced to one node because the feedback constrain of the data flow processor will not allow the next set of data to enter until the previous one is through processing. Therefore iterative and conditional construct acts as a segment in the program pipe and hence is represented as a single node and the node execution time is the time of the longest transition sequence between the initial and terminal nodes of the iterative construct.

Assigning a value of 10 to $N_1$, $N_2$, and $D$, the characteristic timing equations produce:

\[
\begin{align*}
\tau_s(G') &= 143, \quad \tau_p(G') = 95, \\
\tau_s(G_S) &= 1430 \\
\tau_p(G_S) &= 554
\end{align*}
\]

\[
\begin{align*}
\text{IP}(G') &= \frac{\tau_s(G')}{\tau_p(G')} = \frac{143}{95} = 1.50 \\
\text{IP}(G_S) &= \frac{\tau_s(G_S)}{\tau_p(G_S)} = \frac{1430}{554} = 2.58
\end{align*}
\]

In addition to the increased program execution speed, the production rate when the program uses stream input to process a set of data on a feedback data flow processor, is about 51 units of time compared to 143 units of time required for the sequential execution. The production rate may, in some cases, be increased by minimizing the bottleneck.
decomposing the segment into smaller segments without violating precedence relations.

Programs containing nested loops or programs where the conditional encloses the whole body of the program as in validity checking for input data or where the entire program is inside an all enclosing loop, are not well-suited for stream input processing. By incorporating the validity check for input data in stream input construct, programs which contain conditionals to check validity of data can be eliminated and made suitable for stream input processing. The stream input, when used to process a number of input data sets makes the program structure behave like a pipeline. The pipeline behaviour of program not only increases the program execution speed but also increases the production rate if the program is executed in production mode and it is possible to predict the production rate using the timed program graph.

Source of Error in Approximate Timed Program Graph

The machine level timed program graph model represents the behavior of a high level program executing on a feedback data flow processor without introducing any error. But the number of nodes in the graph and hence the cost of estimating the performance may become prohibitive as the program size grows. Therefore, the approximate timed graph model was introduced and this approximate representation introduces errors in the calculation of performance characteristic.
`PROCEDURE example`

```
begin
  real A, B, C, D, Q, R, W;
  integer I, K, J, N;
  real array E(1:10);
  file inf, ouf;

  input N, A, B, C, D file = inf
    format = F(5,2), F(5,2), F(5,2), F(5,2)

  Q := A*B;
  R := C*D;
  W := Q+R;
  I := 1;

  while I <= N do
    F(I) := (W+B)/I = C*D;
    I := I+1
  end;

  J := 1;

  while J <= N do
    W := F(I)*W/J;
    J := J+1
  end;

  output W file = ouf format = F(10,6)
end;
```

Figure 3.12. High Level program and timed program graph
\( (N_1 + 1) : (N_1 + 1) \)

\( T_g(G') = 23 + (7N_1 + 5N_2) \)

\( T_p(G') = 15 + (5N_1 + 3N_2) \)

b) Timed program graph

Figure 3.12 (Continued)
c) Timed program graph, GS, with stream input

\[ t_s(GS) = D \times (23 + 7N_1 + 5N_2) \]

\[ t_p(GS) = (15 + 5N_1 + 3N_2) + (D-1)\max \{(5N_1+1),(3N_2+1)\} \]

Figure 3.12 (Continued)
In the approximate graph model, the node representing high level constructs is assumed to execute only when it receives all its operands. But the computation represented by the node may consist of subcomputations which could have been executed earlier. This disappearance of the partial overlap in computation between data dependent nodes introduces an error into the parallel execution time of the overall program. The example in Figure 3.13 illustrates this effect. A similar effect happens in the evaluation of selectors for arrays with more than one dimension.

There is a certain amount of computational overlap in the case of the repeat construct, which is not modeled by the approximate timed program graph model. This overlap is due to the synchronization taking place after execution as opposed to the while construct where the synchronization takes place before the body executes. Therefore, the repeat construct in the program introduces some error in the estimation of characteristic timing equation $\tau_p(G')$.

1. $A := 2.0$;
2. $B := 4.0$;
3. $C := 5.0$;
5. $E := A*2 + B*C + D$;
6. $F := E*A$

a) High level program

Figure 3.13. Program graph and high level program
b) Machine level graph model

\[ t_p(G) = 6 \]
\[ t_s(G) = 12 \]

\[ t_p(G') = 8 \]
\[ t_s(G') = 12 \]

c) Approximate graph model

Figure 3.13 (Continued)
The error introduced due to the overlap in a repeat construct is illustrated with an example. Figure 3.14 shows the high level program segment and the data flow graph. Figure 3.15 shows the operation execution sequence for five iterations. The operator execution times are assumed to be unity. The superscript in Figure 3.15 indicates the iteration number for a feedback data flow interpreter. For five iterations the data flow interpreter takes 19 steps but the estimated parallel execution time from the timed program graph is 25 step, thereby introducing a sizable error in the parallel execution time.

The last source of error is in the representation of conditional constructs in the graph model. In the feedback data flow processor the values produced inside the conditional are passed on to the successor or gated out of existence. But in the graph model all the values produced inside the conditional are collected before sending to successor constructs. In the graph, this action is represented by the OR node. This delay in forwarding operands to successor increases the parallel execution time and thereby introduces an error in the calculation of $\tau_p(G')$.

The characteristic timing equations contain two sets of parameters $\{a_i\}$ and $\{N_i\}$ whose values are either input data dependent or calculated during program execution. The IP(G') is calculated by assigning 0's and 1's to the $a_i$'s, and it is possible that the branch taken may not be the one which the
A := 0;
N := 5;
C := 10;
D := 8;
B := 2;
I := 1;

1. repeat
2.   A := A+D*A;
3.   C := A*C+B;
4.   I := I+1
5. until I > N:

Figure 3.14. High level program and the data flow graph
Figure 3.15. The operator execution sequence of the data flow graph and timed program graph

program executes, thus introducing inaccuracy. The predicted intrinsic parallelism is obtained by letting all \( N_1 \)'s approach infinity at an equal rate in \( IP(G') \). This amounts to assuming
that the innermost loop totally dominates the computation. The amount of error introduced can be minimized by allowing the user to provide value to the parameters and using the entire characteristic timing equations in the estimation if $IP(G')$.

Conclusion

The timed program graph can be used to model execution of high level programs on a feedback data flow processor with scalar as well as vector processing capabilities. Even though this model introduces errors in the estimation of the characteristic timing equation for parallel execution, the estimated intrinsic parallelism for a sample of 25 Fortran and Algol programs were very close to the values obtained through the simulated execution of the above mentioned programs. The estimated figures of merit for a processor with vector capability is encouraging and this model can hopefully be extended to represent more complex operation such as chaining and stream oriented computations.
CHAPTER IV. INTEGRATION OF PROGRAM ANALYSIS IN COMPILER

Introduction

Chapter III described the program graph analysis to estimate intrinsic parallelism in a high level program. Even though the graph model was used to estimate intrinsic parallelism, it may be extended to produce estimates of system requirements at compile time. The program graph model may also be used to produce process partitioning into subprocesses for distributed processing, generating test cases for program testing and program verification [11, 19]. In this chapter attention is mainly on the generation of the timed program graph within the context of an existing data flow compiler. The timed program graph is an approximate representation of the data flow graph. The section on Overview of Compiler presents an overview of an existing compiler which produces data flow code. Compiler Generated Tables describes what is used in the formation of the timed program graph. The section on High Level Constructs, Execution Time and Program Graph presents the high level constructs and their program graphs along with the calculation of the execution times. The section on Generation of Timed Program Graph explains the procedure to form the timed program graph. The last section presents a high level program, the tables generated by the compiler for this program and the timed program graph.
Overview of Compiler

The three stages in the data flow compiler are parsing, flow analysis and code generation. The timed program graph is formed using the tables generated during the parsing and flow analysis stages. The compiler performs a top-down parse using the recursive descent technique. The parser builds the symbol table using the tokens produced by the lexical analyzer.

Whenever the parser recognizes certain syntactic units, entries are made in the IFT (Intermediate Form Table). The parser also builds the parse tree for various expressions and places a pointer to the tree in the IFT. Each syntactic unit in the program has an entry in the IFT and for each entry in the IFT, an inset and an outset is formed. The inset and the outset indicate the left and right context identifiers. When a value for an identifier is produced, a list is formed indicating the place where this identifier is used. When the value for an identifier is used, a list indicating the location where this identifier was produced is formed. Therefore, a two-way linking is formed between the location where an identifier was produced and where it is used. The parser only generates a partial IFT with links to program statements.

The second step, flow analysis, completes the IFT by obtaining the uses and definition of all the identifiers used in the program. Uses is a list of locations in the program where a given identifier is used and the definition is the location
where an identifier is assigned a value. Once the uses and
the definition of identifiers are known, the data dependency
among statements can be established. For certain constructs,
the uses and the definition of identifiers are collected at
the innermost level first and then propagated outwards. The
inset and outset lists for each entry in the IFT are only
partially formed during the parsing phase, because the inset
and outset entries of type If, Then, Else, Repeat, and While,
are the union of the insets and the outsets of the statements
in the bodies of the above constructs. Therefore, during
the flow analysis, every entry that appears in an inset is
examined to complete the use and definition lists. All the
information needed to form the program graph is available
at the end of the flow analysis.

Compiler Generated Tables

The table of central importance in forming the program
graph is the IFT. The parser as it recognizes certain
syntactic units creates IFT table entries by collecting the
relevant information for that syntactic unit. Figure 4.1
shows the templates for compiler-constructed tables. A brief
discussion of each field and its use follows.

Statement type

The statement type is used extensively in the formation
of the program graph nodes. This field contains one of the
Figure 4.1. Compiler generated tables
compiler recognized syntactic units given below:

Procedure Declaration

Procedure
End

Function Declaration

Function
End

Input

Output

Assignment

If - Then - Else

If
Condition
Then
Close (for Then)
Else
Close (for Else)
Close (for If)

If - Then

If
Condition
Then
Close (for Then)
Close (for If)
While - Do
  While
  Condition
  Close (for While)
Repeat - Until
  Repeat
  Condition
  Close (for Repeat)
Procedure Call

**Program line number**

This is the physical line number of the high level program statement. This field is not used in the formation of the program graph.

**Number within line number**

Major syntactic elements within a particular line are numbered and this field is not used in generating the program graph.

**Expression tree**

During the parsing step, an expression tree of relevant information is built as an intermediate form of the program text. The expression tree field is a pointer to the parse tree for the expression. A null pointer is placed in this field for table entries with no expression tree.
Inset

The inset is a list of identifiers that form the input to a statement (used in righthand context). The inset field contains a pointer to the inset list. For entries in the table that represent an If, While, and Repeat, the inset consists of the union of insets of all the statements in the body of the construct.

Number in inset

Represents the cardinality of the inset list.

Outset

This field is a pointer to a list of identifiers whose values are produced by this statement. For an entry of type If, While, Repeat in the IFT table, the outset list is the union of all the outsets of the statements in the body of a particular construct.

Number in outset

This field represents the number of identifiers in the outset list. This indicates the number of use lists.

Forward link

This link is the sequential ordering of the entries in the IFT consistent with the ordering of the program text.

Backward link

With the exception of the close entry in conditional and
iteration constructs, the backward links point to the previous entries in the IFT. In the case of the close entry, the backward link points to the first entry associated with the corresponding conditional or iteration construct. It is this link that is helpful in extending the program graph from a conditional or an iteration construct.

Other compiler-produced tables used in the formation of the program graph are the inset list, the outset list, and the use list. The inset list contains the following pointer fields:

**Next**

The next field is a pointer to the next entry in the inset list. If there is only one identifier in the inset list this pointer is set to null.

**Text**

The text field is a pointer to the entry of the IFT in whose inset list this identifier belongs. The content of this field is used as the node number in the program graph.

**Symbol table entry**

This field is a pointer to the symbol table entry representing the identifier name forming part of the inset list.

**Use list**

The use list field is a pointer to the use list which has two fields, next and domain. The next field points to the
next entry in the use list. The domain field is a pointer to an inset entry of an embedded construct (if any) that uses this variable. The use list field is only applicable for the IFT entries of type Procedure, Function, If, Then, Else, While and Repeat. For any other type a null pointer is placed in this field. This field indicates the use of an identifier within the body of the high level construct. The program graph for the constructs Procedure, Function, If, Then, Else, While and Repeat are generated using the use lists.

Definition field

This field is a pointer to an entry in an inset or an outset list. This pointer points to an outset list if the definition of the identifier comes from a construct at the same level as the one in which this use appears. It points to an enclosing inset list, if the definition of the identifier comes from a construct whose level is higher than the construct where the use of this identifier appears. For the second case, the definition has been passed in through the inset of the enclosing construct.

The outset list consists of next, text, symbol table entry, and use list fields. The first three fields are similar to the inset list fields. The fourth field, use list, is a pointer to a list similar to the use list in the inset list if the identifier being defined has a subsequent use. This field denotes the use of this identifier outside this
high level construct.

For example, assume that the identifier X has already been defined and X is in the inset of the IFT entry corresponding to A₁. Let A₀ be the IFT entry at which the definition has taken place. Let α be the inset field pointer for the use of X and β be the outset field pointer where X is defined. When forming the inset list, a search is made to find the place where X was defined. In this case, the value β is returned because β represents the outset list which contains the identifier X. This value of β is placed in the definition field of the inset list of X. In entry β, a use list v is started or extended (if this is not the first use list) and in the domain field of the use list starting at v, the pointer α is placed denoting that one of the uses of the identifier X produced at β is α. This linking of uses and definition of an identifier is shown in Figure 4.2.

![Diagram of Inset List, Outset List, and Use List](image)

Figure 4.2. Linking uses and definition of an identifier
High Level Construct, Execution Time and Program Graph

The IFT contains all the information required to produce the timed program graph. This information may be present directly in the IFT fields or may be a pointer to other tables which contain the required information. In this section, a description of the contents of the IFT fields is presented. Also, the expression tree, if applicable, the execution time in terms of the base machine operations and the program graph are given. Certain fields in the IFT, such as the forward link (which always points to the next entry in the IFT) and the number in inset and outset (which represent the cardinality of the inset and outset lists, respectively) are not included in the description. The program line and the number within line number fields are not used in the generation of the timed program graph; hence they are ignored in the following discussion.

Procedure

Statement Type - Proc.

The statement type field contains Proc which is a syntactic unit recognized by the compiler.

Expression tree - t

The expression tree field has a pointer to the expression tree, if the tree is present. Otherwise it has a null pointer. The name in the parse tree is actually a pointer to the symbol table entry that corresponds to this name.
Inset - A pointer (α) to the inset list of parameters which have an in or inout attribute. The total number of identifiers in the inset list is placed in the IFT number in inset field.

Outset - A pointer (β) to the outset list of parameters which carry an out or inout attribute. The number of identifiers in the outset list can be obtained from the number in outset field.

Backward link - This field contains a null pointer because a separate table is formed for each procedure.

Execution time - The time taken to execute the node represented by Proc is the sum of the times taken to select all the input parameters in the input structure, to form the result structure corresponding to the outset list and by the identity operation which is part of the procedure activation.
The number of inset and outset list identifiers are directly available in the IFT table. Hence, the execution times are

\[
\tau_s(\text{Proc}) = (\text{number in inset}) \times t_{\text{select}} + \tau_s(B) +
\]
\[
(\text{number in outset}) \times t_{\text{append}} + t_{\text{identity}}
\]
\[
\tau_p(\text{Proc}) = t_{\text{select}} + (\text{number in outset}) \times t_{\text{append}} +
\]
\[
\tau_p(B) + t_{\text{identity}}
\]

where

- \(t_{\text{select}}\) - is the time taken to execute the select base machine operation.
- \(\tau_s(B)\) - is the sequential execution time to execute the body of the procedure.
- \(\tau_p(B)\) - is the parallel execution time to execute the body of the procedure.
- \(t_{\text{append}}\) - is the time to execute the append base machine operation.
- \(t_{\text{identity}}\) - is the time to execute the identity base machine operation.

- (number in inset) - represents the content of the number in inset field in the IFT.
- (number in outset) - represents the content of the number in outset field in the IFT.

In the rest of the chapter, a representation similar to the one given above is used.
Processing method  After creating a node to represent Proc, the successor nodes are formed by following the inset field pointer to the use list. The use list indicates the statement(s) where the identifier represented by the use list appears. Hence, either a node is formed if one does not exist already and linked to the Proc node or the existing node is linked to the Proc node.

Program graph

![Diagram of program graph]

End

The only fields of interest in the IFT are the statement type and the forward link.

Statement type - End

Forward link - If there is another procedure in the high level program, then the forward link contains a pointer to the IFT entry of the next procedure. If there is no other procedure present, the forward link will be null.

Execution time

\[
\tau_p(\text{End}) = \tau_s(\text{End}) = 0
\]

Processing method  All free arcs in the program graph are terminated on this node representing the End. Then the forward link is followed to form a node to start a new graph
when this field is not null. If the forward link is null, then no more processing is needed.

Program graph

\[ \bigcup \tau_s(\text{End}) : \tau_p(\text{End}) \]

Function

Statement type - Funct

Expression tree - t

This field contains a pointer to the expression tree:

\[ t \rightarrow \text{Funct} \]

name of function

in

name_1 name_i formal parameters

Inset - The inset field is a pointer to the inset list which consists of all the identifiers in the formal parameter list.

Outset - The outset field has a pointer to the outset list. The outset list is made up of the function name.

Backward link - Null

This field contains a null pointer because each function has a separate IFT.

Execution time The derivation of execution time is similar to that for a procedure. The only difference is that functions have only one identifier in the outset list.

\[ \tau_s(\text{Funct}) = (\text{number in inset}) \times t_{\text{select}} + t_{\text{append}} + t_{\text{identity}} + \tau_s(B) \]
\[ \tau_p(\text{Funct}) = t_{\text{select}} + t_{\text{append}} + t_{\text{identity}} + \tau_p(B) \]

**Processing method**  
The function entry in the IFT corresponds to a new function definition, hence a new graph is started with the first node representing the Funct entry in IFT. Then, the graph is extended from this node by creating as many successor nodes as there are use lists in the inset list, eliminating duplicate nodes.

**Program graph**

![Program graph diagram]

**End** - (for Function)

**Statement type** - End

**Backward link** - The backward link points to the start of this syntactic construct, Funct, in the IFT.

**Execution time**

\[ \tau_S(\text{End}) = \tau_p(\text{End}) = 0 \]

**Processing method**  
The node corresponding to End is the terminal node in the graph and all the free arcs are joined to this node.

**Program graph**

![Program graph diagram]
Input

Statement type - Input

Expression tree - t

The expression tree is one of two forms, depending on whether the identifier in the input list is a simple or a structure. In case the identifier is a structure, the expression tree contains a subtree for the evaluation of the selector of the structure identifier. The format is actually a pointer to an appropriate character string in the constant table.

**Simple Identifier**

```
  t  ┌── Input
    │   └── file name
    │       ┌── format └── name
```

**Structure Identifier**

```
  t  ┌── Input
    │   └── file name
    │       ┌── format └── name of array
    │               └── expression tree for subscript
```

**Inset** The inset list depends upon the type of the identifier in the input list.
Simple identifier  The inset list is null.

Structure identifier  The inset list consists of all those identifiers which appear as selectors and the name of the structure identifier.

Outset  The outset list has the name of the identifier that is being assigned a value and the file name.

Execution time

Simple identifier

\[ \tau_s(\text{Input}) = \tau_{\text{read}} + 2 * \tau_{\text{select}} \]
\[ \tau_p(\text{Input}) = \tau_{\text{read}} + \tau_{\text{select}} \]

Structure identifier

\[ \tau_s(\text{Input}) = \tau(\text{ST}) + \tau_{\text{read}} + 2*\tau_{\text{select}} + \tau_{\text{append}} \]
\[ \tau_p(\text{Input}) = \tau(\text{PT}) + \tau_{\text{read}} + \tau_{\text{select}} + \tau_{\text{append}} \]

where

\( \tau(\text{ST}) \) is the sum of the node execution times of the subscript calculation subtree.

\( \tau(\text{PT}) \) is the longest transition sequence of the subscript calculation subtree.

Processing method  To extend the program graph from the Input node, the outset field in the IFT is used. This field points to the outset list of the Input entry. Nodes are formed using the use lists in the outset list eliminating any duplicate nodes.
**Program graph**

![Program Graph](image)

**Output**

Statement type - Output

Expression tree - \( t \)

Inset The inset field carries a pointer to the inset list which consists of all the identifiers in the input list and the file name.

Outset The outset field has a pointer to the outset list which consists of only the file name.

**Execution time**

\[
\begin{align*}
\tau_s(\text{Output}) &= t_{\text{write}} + \tau(\text{ST}) \\
\tau_p(\text{Output}) &= t_{\text{write}} + \tau(\text{PT})
\end{align*}
\]

where,

\( \tau(\text{ST}) \) is the sum of the node execution times of the expression tree.

\( \tau(\text{PT}) \) is the longest transition sequence of the expression
tree.

**Processing method** The program graph is extended from the Output node by using the use list of the outset list.

**Program graph**

```
Output \rightarrow \tau_S(\text{Output}) : \tau_P(\text{Output})
```

**Assignment**

**Statement type** - Assign

**Expression tree** - $t$

The expression tree formed depends on the type of the identifier on the left-hand side of the assignment operator.

**Simple identifier**

```
\quad \text{t} \quad \rightarrow \quad \text{Assign} \\
\quad \quad \downarrow \quad \quad \downarrow \quad \quad \downarrow \\
\quad \quad \text{Simple identifier} \quad \quad \text{Expression tree} \\
```

**Structured identifier**

```
\quad \text{t} \quad \rightarrow \quad \text{Assign} \\
\quad \quad \downarrow \quad \quad \downarrow \quad \quad \downarrow \\
\quad \quad \text{Name of the structure} \quad \text{Expression tree} \\
```

Expression tree for selector
**Inset** The inset field points to the inset list which is formed from the identifiers appearing on the righthand side of the assignment operator and the selector identifier(s), if the lefthand side is a structure identifier.

**Outset** The outset field has a pointer to the outset list which consists of the identifier on the lefthand side of the assignment operator.

**Execution time**
- $\tau_s(Assign) = \text{sum of all the node execution times.}$
- $\tau_p(Assign) = \text{longest transition sequence of the parse tree.}$

**Processing method** The program graph is extended from the Assign node by creating distinct nodes using the use list in the outset list of the Assign entry in the IFT.

**Program graph**

![Program graph]

**If**

**Statement type** - If

**Expression tree** - null

**Inset** The inset field points to the inset list which consists of the union of all the inset list identifiers of the Condition, Then and Else syntactic units of the conditional
construct.

**Execution time**

\[ \tau_s(\text{If}) = \tau_p(\text{If}) = 0 \]

**Processing method** The forward link is used to extend the program graph by forming a node corresponding to the entry pointed to by the forward link in the IFT.

**Program graph**

![Diagram of a program graph with a node labeled If \( \tau_s(\text{If}) : \tau_p(\text{If}) \)]

**Condition - (for If)**

**Statement type - Cond**

**Expression tree - t**

![Expression tree diagram with a relational or logical operator and two operand trees]

**Inset** The inset field points to the inset list which consists of all the identifiers in the conditional expression.

**Outset** The outset field contains a null pointer.

**Execution time**

\[ \tau_s(\text{Cond}) = \text{sum of the node execution times of the expression tree.} \]

\[ \tau_p(\text{Cond}) = \text{longest transition sequence of the expression} \]
Processing method

The program graph is extended from the condition node by following the forward link and creating a node corresponding to this entry in the IFT. Also a branching probability parameter is associated with the arc formed.

Program graph

\[
\text{Cond} \quad \tau_s(\text{Cond}) : \tau_p(\text{Cond})
\]

\[\alpha_i\]

Then

Statement type - Then

Expression tree - null

Inset The inset field points to the inset list which is the union of inset lists of all statements in the body of Then.

Outset The outset field points to the outset list formed from the union of all the outset lists of the statements in Then.

Execution time

\[
\tau_s(\text{Then}) = \tau_p(\text{Then}) = 0
\]

Processing method The inset list consists of all the identifiers used in the Then body. The program graph is extended by forming nodes using the use lists of the inset lists. Again duplicate nodes are eliminated from the program graph.
Program graph

\[
\text{Then} \quad \tau_s(\text{Then}) : \tau_p(\text{Then})
\]

\[
\begin{array}{ccc}
& l & \cdots & k \\
\end{array}
\]

Close - (for Then)

Statement type - Close

Expression tree - null

Inset - null

Outset - null

Execution time

\[
\tau_s(\text{Close}) = \tau_p(\text{Close}) = 0
\]

Processing method

The close syntactic unit forms the end for the statements in the body of Then. Hence, all the nodes with free arcs inside the body of Then are terminated on this node corresponding to the Close. The graph is extended by forming a node pointed to by the forward link of the close entry in the IFT. There are two possible ways of processing depending on the type of entry in the IFT pointed to by the forward link. If that statement type is Else then the backward link is used to get to the corresponding If entry in the IFT. The backward link of Then points to the conditional which drives the Then and the Else. Therefore a node to represent the Else is formed and linked to the conditional
node. The arc joining the conditional node to the Else node is assigned a branching probability. If the forward link points to a Close statement type then a node is formed and linked to the node being processed. This indicates the absence of the Else part. The program graph represents the second case.

Program graph

Else

Statement type - Else
Expression tree - null

Inset The inset field points to the inset list which consists of the union of all the inset lists of the statements in the body of the Else.

Outset The outset field points to the outset list which consists of the union of all the outset lists of the statements in the body of Else.

Execution time
\[ \tau_s(Else) = \tau_p(Else) = 0 \]

Processing method The partial program graph is extended from the Else node by forming successor nodes using the
use lists in the inset list. This creates nodes for all the statements in the body of Else.

**Program graph**

![Program graph diagram]

- **Close** - (for Else)
  - **Statement type** - Close
  - **Expression tree** - null
  - **Inset** - null
  - **Outset** - null
  - **Execution time**
    \[ \tau_s(\text{Close}) = \tau_p(\text{Close}) = 0 \]

**Processing method** All the free arcs in the body of Else are terminated on the close node. Then a new node is formed using the forward link. This new node represents the Close node for the If-Then-Else construct. Therefore, the Close nodes corresponding to the Then and Else syntactic units are connected to this new node formed. The backward links can be used to identify the Close syntactic units corresponding to the Then and Else part.
Program graph

```

Program graph

CLOSE

\[ T_s(Close) = T_p(Close) = 0 \]

or

Close - (for If)

Statement type - Close
Expression tree - null
Inset - null
Outset - null
Execution time
\[ T_s(Close) = T_p(Close) = 0 \]

Processing method This, Close, syntactic unit signifies the end of the If-Then-Else construct. Hence, all successor nodes of the conditional construct are extended from this node. To extend the partial graph, the backward link to the If entry is followed to obtain the outset list, a node is formed for each one of the use lists in the outset lists, eliminating all duplicate nodes. The Close node is linked to all the new nodes formed.
Program graph

Example for If-Then-Else with unit operation time

if \( A > 0 \) then begin
  \( C := C \times A; \)
  \( A := W \times A \)
end
else begin
  \( C := C + A - 5; \)
  \( Q := W/2 + A \)
end;
While

**Statement type** - While

**Expression tree** - null

**Inset** The inset field points to the inset list which consists of the union of all the inset lists of the statements in the body of While.

**Outset** The outset field points to the outset list which consists of the union of all the outset lists of the statements in the body of While.

**Execution time**

\[ \tau_s(\text{While}) = \tau_p(\text{While}) = 0 \]

**Processing method** The partial program graph is extended from the While node by following the forward link and forming a node corresponding to the condition of the While construct.

```
While \( \tau_s(\text{While}): \tau_p(\text{While}) \)
```

**Condition**

**Statement type** - Cond

**Expression tree** - \( t \)

```
t
```

- Relational or logical operator
- tree for operand 1
- tree for operand 2 (if needed)
Inset  Inset field points to the inset list which consists of all identifiers which appear in the expression tree.

Outset - null

Execution time

\[ \tau_s(\text{Cond}) = \text{sum of the node execution times of the expression tree.} \]

\[ \tau_p(\text{Cond}) = \text{longest transition sequence of the expression tree.} \]

Processing method  The statements in the body can execute only after the conditional executes. Therefore, the program graph is extended from this node by forming new nodes corresponding to the use lists on the inset list of the While entry in the IFT. To get to the inset lists the backward link of the Cond entry is used.

Program graph

\[
\begin{align*}
\text{Cond} & \quad \tau_s(\text{Cond}):\tau_p(\text{Cond}) \\
1 & \quad \cdots \\
& \quad k
\end{align*}
\]

Close - (for While)

Statement type - Close

Expression tree - null

Inset - null

Outset - null
Execution time
\[ \tau_s(\text{Close}) = \tau_p(\text{Close}) = 0 \]

Processing method All the free arcs inside the body of the While-do construct are terminated on the node corresponding to the Close syntactic unit. The partial program graph is extended from this node by forming nodes corresponding to the use lists in the outset lists of the While entry in the IFT. The backward link in the Close entry in the IFT is used to obtain the outset lists.

Program graph

Example for While-do

1. \textbf{while} \quad I \leq N \textbf{do}
2. \quad \textbf{Q := A*B;}
3. \quad \textbf{C := Q+C;}
4. \quad \textbf{I := I+1}
5. \quad \textbf{end;
**Repeat**

**Statement type** - Repeat

**Expression tree** - null

**Inset**  Inset field points to the inset list which consists of the union of all the inset lists of the statements in the body of the Repeat-Until construct.

**Outset**  Outset field points to the outset list which is the union of all the outset lists of the statements in the body of the Repeat-Until.
Execution time
\[ \tau_{s}(\text{Repeat}) = \tau_{p}(\text{Repeat}) = 0 \]

Processing method The partial program graph is extended from the Repeat node by forming nodes corresponding to the use lists in the inset list of the Repeat entry in the IFT.

Program graph

Condition

Statement type - Cond

Expression tree - \( t \)

Inset The inset field points to the inset list which consists of all the identifiers in the expression tree.

Outset - null

Execution time

\[ \tau_{s}(\text{Cond}) = \text{sum of the execution times of the nodes in the expression tree.} \]

\[ \tau_{p}(\text{Cond}) = \text{longest transition sequence of the expression tree.} \]
Processing method: The node formed corresponding to the condition is linked to the Close node of the Repeat-Until construct.

**Program graph**

![Program graph](image)

Close - (for Repeat)

**Statement type** - Close

**Expression tree** - null

**Inset** - null

**Outset** - null

**Execution time**

\[ \tau_s(\text{Close}) = \tau_p(\text{Close}) = 0 \]

Processing method: All the free arcs inside the body of Repeat-Until are terminated on this node. Then, the partial program graph is extended by forming nodes corresponding to the use lists in the outset lists of the Repeat entry in the IFT.

**Program graph**

![Program graph](image)
Example for Repeat-Until

1. repeat
2. Q := A*B;
3. C := Q+C;
4. I := I+1
5. until I>N;

Note that the node execution times are multiplied by a loop execution parameter (n) in the program graphs for While-do and Repeat-Until. This transforms the cycle graph into a transitive graph.

Call

Statement type - Call

Expression tree - t
An identifier is either a simple or a subscripted identifier. In the case of the subscripted identifiers a subtree of the form shown below is part of the expression tree.

```
Subscripted Identifier
   ↓
Expression for Selector
```

**Inset** The inset field points to the inset list which consists of all input parameters including those appearing in the selector expressions.

**Outset** The outset field points to the outset list which consists of all output parameters, including subscripted identifiers.
Execution time

\[ \tau_S(\text{Call}) = (\text{number in inset}) \times t_{\text{append}} + t_{\text{apply}} + \tau_S(\text{Procedure}) + t_{\text{identity}} + (\text{number in outset}) \times t_{\text{select}} + \text{sum of the node execution times of selector.} \]

\[ \tau_P(\text{Call}) = (\text{number in inset}) \times t_{\text{append}} + t_{\text{apply}} + \tau_P(\text{Procedure}) + t_{\text{identity}} + t_{\text{select}} + \text{max \{longest transition sequences of the selectors\}} \]

Processing method The partial program graph is extended from the Call node by creating nodes corresponding to the use lists in the outset list.

Program graph

\[ \text{Call} \quad \tau_S(\text{Call}):\tau_P(\text{Call}) \]

Generation of Timed Program Graph

The formation of the timed program graph using the program graphs presented in Chapter IV is described here. The estimation of execution times for each node can be obtained either when the parse tree is formed or when the program graph is formed. A table of execution times for the base machine operations is needed to form the node execution times.
The timed program graph is formed from the IFT which contains all the information needed either directly or as a pointer to the relevant information in other tables. Other tables of importance in forming the timed program graph are the inset lists, the outset lists, and the use lists (see Figure 4.1). The inset list contains the identifiers used in right-hand contexts in a high level construct. The outset list consists of the identifiers used in left-hand contexts (defined) in a high level construct and the use list indicates the use of the identifier in other high level constructs. These three lists along with the forward link and backward link play a central role in determining the data dependencies among the high level constructs.

The syntactic units recognized by the compiler can be divided into three groups based on the way they are used to form the timed program graph. The first group of syntactic units (i.e. the Input, Output, Assignment and Call (procedure invocation)) uses the use list of the outset list to extend the timed program graph. The second group (i.e. Procedure definition, Function definition, Repeat, While, If, Then, and Else) uses the use lists on the inset list to form the data dependencies for the statements inside the body of the construct and uses the use lists on the outset list to extend the program graph outside the body of the construct being processed. The third group (i.e. Close and End) is used to form
special nodes in the program graph.

A timed program graph is formed for each of the procedures in the high level program. The first node in the timed program graph corresponds to the procedure definition. Once this node is formed the sequential and parallel execution times are calculated using the parse tree for that procedure definition. Next, the successor nodes are formed using the use lists on the inset lists. The procedure node is saved to form links to nodes representing statements which are assigned constant values. The next step is to use the forward link to get to the next IFT entry. If a node corresponding to this entry in the IFT is not already present (as is the case when the procedure definition is the main procedure or the statement assigns a constant value to an identifier) a node is formed and the Procedure definition node is linked to it. The execution time is calculated from the parse tree and assigned to this node. The next step in the processing depends on the statement type that is being pointed to by the forward link of the IFT entry just processed.

If the statement type is either Repeat or While, an iteration parameter is generated to represent the number of times the loop executes. The execution time of every node formed, until the corresponding Close entry is encountered, is multiplied by the iteration parameter. To form the program graph for the Repeat construct, the inset list is used. The
inset field in the IFT points to the first inset list. If there is more than one element in the inset list then they are linked using the next field of the element. A null pointer in the next field indicates that this is the last inset list. Each one of the inset lists may have more than one element. If more than one element exists a linked list is formed using the next field in the use list. The outset list is also organized in a similar way. To extend the program graph from the Repeat node, a node is formed using the first use list in the first inset list. Then another node is formed corresponding to the next use list if the next field of the first use list is not null and the node formed does not exist already. This process is repeated until the next field in the use list is null. Now, the next field in the element is checked. If it is not null, more nodes are formed using the use lists as before. This is repeated until the next field in the inset list is null. A null pointer indicates the end of processing for the Repeat node. The outset list processing is similar to the inset list. The content of the number in inset field indicates the number of the inset lists for an IFT entry.

If the statement type is While, the program graph is extended by forming a node corresponding to the forward link of the While entry in the IFT (this node corresponds to the predicate). To form nodes to represent the statements in the body of the While, the graph is extended from the conditional
node by using the inset list of the While. The inset list is obtained by following the backward link in the IFT for the conditional. If the program graph formed is a multigraph, all arcs except one are removed to change it to a simple graph. This can be done easily by not forming any arc between the node being processed and an already existing node.

If the statement type is If then the branching probability parameter has to be generated. The program graph is extended from the IF node by using the forward link and forming a node corresponding to the conditional. From the conditional node a successor node is formed and the branching probability generated is assigned to the connecting arc. This successor node corresponds to the Then entry in the IFT. After saving the conditional node, the graph is extended using the forward link. The nodes corresponding to the statements in the Then body are formed, eliminating multigraphs, using the inset lists of the Then entry in the IFT. If the statement type is Else then the program graph is extended by using the inset list of the Else entry in the IFT.

If the processing node belongs to group two, then the outset list is used to extend the graph. Successor nodes are formed using the use lists in the outset list of the processing node. The process of forming the successor nodes is terminated once all the use lists in the outset lists are visited. Again no arcs are formed when a node formed in one
step is referenced again in the same step.

If the statement type is Close and its backward link points to an If, While or Repeat syntactic unit, then the program graph is extended from the Close node using the outset lists of the If, While or Repeat. If the backward link of Close points to an Else syntactic unit then the free arcs within the body of the Else are terminated on this node and the graph is extended using the forward link. If the backward link points to a Then syntactic unit and the forward link points to an Else syntactic unit then a node corresponding to the Else is formed and is linked to the conditional node. If the forward link points to a Close then the graph is extended using the forward link. If the statement type is End and the forward link is null then the processing is terminated. Otherwise a new graph is started and the processing is continued using the forward link which points to a new procedure definition. Processing of nodes in the partial program graph is done by following the forward link.

Example

In this section a high level program, the compiler generated tables (IFT, inset list, outset list and use list), the parse trees, and the timed program graph formed using the IFT are presented.

The high level language program for which the timed program graph is formed is shown in Figure 4.3. Figure 4.4
represents the parse trees for the constructs in the IFT and the corresponding sequential and parallel execution times, assuming unit operation time for the base machine operations. Figure 4.5 is the compiler generated IFT. The timed program graph formed from the IFT, inset list (Figure 4.6), outset list (Figure 4.7) and the use list (Figure 4.8) is shown in Figure 4.9.

The node numbers in the timed program graph in Figure 4.9 correspond to the serial numbers in the IFT. The first entry in the IFT is the starting point to form the timed program graph. The statement type of this entry is Proc and it belongs to group two constructs. The node with Number 1 is formed and using the pointer \( t_1 \) the execution time is estimated and assigned to this node. Since the inset list is null as indicated by the inset field in the IFT and this being the main procedure, the program graph is extended by following the forward link. This points to the fifth entry in the IFT and hence a node with number five is formed. Before processing node five, the starting node is saved for later use. The statement type corresponding to node five is Input and is part of group one. Therefore, the outset list is used to form the successor node(s). The field number in outset indicates the number of elements in the outset lists. Using the pointer, \( \beta_3 \), in the outset field the outset list is located. The use list field, \( v_{25} \), in the outset list (\( \beta_3 \)) points to the use list. The use list's domain field indicates the location
program mapout
6. begin
7. real xn, yn, sum, sum1, sum2, s;
8. integer I, N;
9. file inf, opf;
10. procedure summation (in (s1,s2,s3), out (s4))
12. begin
14. real s1, s2, s3, s4;
16. s4 := s1 + s2 + s3
17. end;
19. input xn, yn file = inf format = F(5,3), F(5,3);
20. I := 1;
21. N := 5;
22. sum1 := 0.0
23. sum2 := 0.0
24. sum := 0.0
26. while I <= N do
27. sum := sum + 2.0 * xn * yn;
28. if I - I/2 * 2 = ) then begin
29. sum1 := sum1 + xn*yn;
30. sum2 := sum2
31. end
32. else begin
33. sum1 := sum1;
34. sum2 := sum2 + xn/2.0 * yn/2.0
35. end
36. I := I+1
37. end;
39. summation (in{sum, sum1, sum2}, out{s});
40. output s file = opf format = f(5.3)
41. end

Figure 4.3 High Level Program
Figure 4.4 Compiler constructed parse trees for the high level program
Figure 4.4 (Continued)
Figure 4.4 (Continued)
<table>
<thead>
<tr>
<th>Serial Statement</th>
<th>Program Number</th>
<th>Tree</th>
<th>Inset Number</th>
<th>Outset Number</th>
<th>Forward Link</th>
<th>Backward Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Proc</td>
<td>4</td>
<td>t₁</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Proc</td>
<td>10</td>
<td>t₂</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Assign</td>
<td>16</td>
<td>t₃</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>End</td>
<td>17</td>
<td>α₇</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Input</td>
<td>19</td>
<td>t₅</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Input</td>
<td>19</td>
<td>t₆</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Assign</td>
<td>20</td>
<td>t₇</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>Assign</td>
<td>21</td>
<td>t₈</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>Assign</td>
<td>22</td>
<td>t₉</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>Assign</td>
<td>23</td>
<td>t₁₀</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>Assign</td>
<td>24</td>
<td>t₁₁</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>While</td>
<td>26</td>
<td>t₁₃</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>Cond</td>
<td>26</td>
<td>t₁₇</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>14</td>
<td>Assign</td>
<td>27</td>
<td>t₁₄</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>15</td>
<td>If</td>
<td>28</td>
<td>t₁₅</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>16</td>
<td>Cond</td>
<td>28</td>
<td>t₁₆</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>Then</td>
<td>28</td>
<td>t₁₇</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>18</td>
<td>Assign</td>
<td>29</td>
<td>t₁₈</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>19</td>
<td>Assign</td>
<td>30</td>
<td>t₁₉</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>20</td>
<td>Close</td>
<td>32</td>
<td>α₃₆</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>21</td>
<td>Else</td>
<td>32</td>
<td>α₃₈</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>22</td>
<td>Assign</td>
<td>33</td>
<td>α₄₂</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>23</td>
<td>Assign</td>
<td>34</td>
<td>α₄₃</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>24</td>
<td>Close</td>
<td>35</td>
<td>α₄₆</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>25</td>
<td>Close</td>
<td>35</td>
<td>α₄₉</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>26</td>
<td>Assign</td>
<td>36</td>
<td>α₄₈</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>27</td>
<td>Close</td>
<td>37</td>
<td>α₄₉</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>28</td>
<td>Close</td>
<td>37</td>
<td>α₅₀</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>29</td>
<td>Call</td>
<td>39</td>
<td>α₅₃</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>30</td>
<td>Output</td>
<td>40</td>
<td>α₅₆</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>31</td>
<td>End</td>
<td>41</td>
<td>α₅₉</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

Figure 4.5. Intermediate form table (IFT) for the high level program produced by the compiler
Figure 4.6. Compiler Generated inset list for the high level program
Figure 4.6 (Continued)
Figure 4.6 (Continued)
Figure 4.7. Compiler generated outset list for the high level program
Figure 4.8. Compiler generated use list for the high level program
Figure 4.8 (Continued)
Figure 4.9. Timed Program Graph Generated from Compiler Produced Tables
where this identifier is used. The inset list where this identifier is used is at $\alpha_{13}$.

The text field entry in the inset list pointed to by $\alpha_{13}$ is the successor of node five. Therefore, a node with text field contents (12) is formed and linked to node five. The null pointer in the next field of the use list pointed to by $\alpha_{25}$ indicates that this identifier is not used in any other location. Now, the next field in the outset list is checked, which points to the outset list at $\beta_4$. The same procedure as above is followed to form the node 6 in the graph. The execution time for node five is formed using the tree pointer $t_5$. Once the processing of node five is done, the forward link is used to extend the graph. Note that the forward link points to an already existing node. The processing at this new node is based on the statement type of the node. Processing of node 6 is similar to node 5 except that the use list field for the outset list at $\beta_6$ is null. This indicates that it is not used anywhere else and therefore becomes a free arc. This node is saved so that it can be linked to the last node of the construct. In this case, the last node corresponds to the procedure definition (node 31). IFT entries 7, 8, 9, 10 and 11 need special processing because they assign constant values to identifiers. The inset field contains a null pointer to indicate this. Here, nodes 7, 8, 9, 10 and 11 are formed and the starting node (1) is
linked to these nodes. Formation of successor nodes from these nodes is similar to group one constructs.

Node 12 corresponds to the While syntactic unit. The program graph (for the statements in the body of the While) is formed after forming the node for the conditional. To form the node for the conditional, the forward link is used. Once this conditional node (13) is formed, the successor nodes are formed from it using the inset list of the While node (12). The backward link of entry 13 in the IFT points to the While. The inset list pointer is a₁₀ and there are seven inset lists. The first use list is v₄ and this produces the node 26, which is the text field entry in the inset list at a₄₈. The next field of v₄ points to v₅₂ and node 15 corresponding to this use list is formed. This use list is linked to v₄₅ and the text field points to the processing node itself, hence no arc is formed. The next inset list to be processed is at a₁₁ and the use list is v₅. Again no new node is formed because node 13 already exists and no arc is formed since it refers to itself. Using the use list v₆ on the inset list a₁₂, a new node 14 is formed as a successor to node 13. If the rest of the use lists are used, they refer to already existing nodes and hence no arcs need be formed thus eliminating the multiple arcs between an immediate predecessor and an immediate successor.

Node 28 represents the Close syntactic unit of the While-
Do construct. To extend the program graph from this node, the outset list is used because the backward link points to a While (the 12th entry in the IFT). Successor node 29 is formed using the outset list at β12. Processing is continued by following the forward link. The timed program graph formed from the compiler generated tables is shown in figure 4.9.

The characteristic timing equations for the timed program graph in Figure 4.9 are:

\[ \tau_s(G') = \text{sum of the sequential node execution times} = \{l+1+l+1+4+4+n+n+4n+3n} \]

\[ + \alpha_1(2n + n) + \alpha_2(4n + n) + 10 + 1 \}

\[ = 24 + 9n + 31n \alpha_1 + 5n \alpha_2 \]

\[ \tau_p(G') = \{\text{parallel execution time} \}
\]

\[ = \{\text{max [1,1,1,1,1,3,3+3] + max [n+n, n+3n, n+4n + \alpha_1(n+2n) + \alpha_2(n+4n)] + n+1} \}

\[ = 15 + 5n + 3n \alpha_1 + 5n \alpha_2. \]

The characteristic timing equations show the program structure of the high level program. The sequential execution time, \( \tau_s(G') \), shows the static structure of the entire program, while the parallel execution time, \( \tau_p(G') \), shows only the program structures which are sequenced by the data dependency. The iterative program structure can be identified by the \( n_1 \) terms in the characteristic equations. The order of the \( n_1 \)
terms indicate the nesting level of the iterative program structure. In a similar way the $\alpha_i$ terms indicate the structure introduced by the conditional construct and the order of $\alpha_i$ terms again show the nesting depth of the conditional constructs. The characteristic timing equations for the timed program graph in Figure 4.9 indicate that there is one iterative construct and a conditional construct enclosed within the loop. The order of both $n$ and $\alpha_i$ show that the nesting level for the conditional as well as the iterative construct is only one.
CHAPTER V. CONCLUSIONS

The software simulation of a feedback data flow processor capable of only scalar operations is used to obtain the resource requirements and execution speedup for a set of twenty-five high level programs. The simulated execution of the high level programs demonstrates the feasibility of translating the existing high level programs to a data flow language and executing them on a data flow processor. The timed program graph model is introduced as an alternate and inexpensive method to estimate the performance characteristics at a higher level than the simulation method. The graph model is used to obtain the intrinsic parallelism for both the scalar and the vector processing systems. The characteristic timing equations generated from the approximate program graph consist of the sequential and parallel execution times. For the scalar architecture, the estimated numeric values of intrinsic parallelism are close to the actual measures obtained through the simulated execution of the actual data flow programs, thereby demonstrating the utility of these equations. The numeric values for intrinsic parallelism obtained from the characteristic equations for vector architecture in addition to scalar operations demonstrate large factors of execution speedup. These figures are obtained under the assumptions that potentially competing nodes are executed sequentially and that there is only one vector processing unit of each type.
Finally, a method to generate the timed program graph at compile time from the compiler generated tables is presented.

The measured intrinsic parallelism indicates only the scalar concurrency. Hence, the gain in execution speedup is only modest compared to the sequential processing system. Higher execution speedup may be obtained by applying one or more of the following methods.

1. Use the more complex transformation given in [20] to the high level program.

2. Use a translation technique to expose stream oriented computations [8, 38].

3. Use the loop unravelling processor [2].

The peak data path and memory requirements encountered in the execution of programs can only be met by introducing new and efficient memory organization and multiple instruction fetches. The data path requirements can be reduced by selectively eliminating some of the feedbacks without introducing data nondeterminancy. The numeric values for the data path, memory, and processor requirements may be used as a guideline in the selection of machine parameters for future data driven processors.

To exploit the full potential of the data processor for general purpose computing applications more research and much experimentation is needed. The execution speedup can
be increased significantly in some programs by utilizing the concept of streams. However, stream processing requires higher data rates and increased memory traffic and data path requirements. Hence, further research is needed to study the trade-off in terms of increased cost and the speedup that can be obtained. Another related field of study is the memory organization to meet the processing demands. Other areas of research include a study to establish the relation between the program structure and the machine parameters; the effect of removing feedback on execution speedup and the effect of multiprogramming on resource utilization in a data driven processor.

The estimate of resource requirements that can be obtained by extending the timed program graph can be used in job scheduling. The overlapped execution that takes place when the Repeat construct executes is not estimated accurately using the timed program graph. This introduces a certain amount of error in the estimation of the parallel execution time. The accuracy of the estimated intrinsic parallelism can be improved if a straightforward and simple method can be devised to estimate the execution overlap at compile time.
REFERENCES


ACKNOWLEDGMENTS

I am indebted to Dr. Arthur E. Oldehoeft for suggesting the research topic, for the many hours of assistance, for his advice and encouragement throughout my stay at Iowa State University. I take this opportunity to thank Professor Roy J. Zingg for his help and the constructive criticism during the preparation of this thesis.

I wish to thank Professor Roy F. Keller, Professor Robert J. Lambert and Distinguished Professor Arthur V. Pohm for serving as committee members.

I thank the members of the data flow group for their help and Mr. K. Vijayakumar for proof reading the thesis.

I am grateful to the Lutheran Work Federation for supporting my masters program.

Finally, I would like to thank Professor P. S. Srinevasan, Department of Electrical Engineering, Regional Engineering College, Calicut, India, for his constant encouragement and for being a model teacher during my undergraduate years.

The work done in this thesis was supported in part by the National Science Foundation and in part by the Science and Humanities Research Institute and the Engineering Research Institute of Iowa State University.
APPENDIX

Resource Requirements

The data path, memory and processor requirements for the twenty-five high level programs are presented in the Appendix. In each entry the first figure is the maximum resource requirement encountered during any time step of the simulated execution. The second figure gives the average resource requirement over the complete execution (for the processor requirement, the average is over those periods when a demand existed), and the third figure is the standard deviation from this average.
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Instruction Fetch</th>
<th>Acknowledgment</th>
<th>Destination</th>
<th>Control Tokens</th>
<th>Structure Memory Reference</th>
<th>Elementary Memory Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEWB</td>
<td>10/2.41/2.4</td>
<td>13/3.63/3.3</td>
<td>15/3.29/3.5</td>
<td>6/0.50/1.7</td>
<td>3/4.02/1.0</td>
<td>14/4.4/3.4</td>
</tr>
<tr>
<td>PRPF</td>
<td>34/4.1/3.6</td>
<td>48/8.8/6.8</td>
<td>51/6.5/6.7</td>
<td>34/3.3/6.8</td>
<td>50/3.5/5.2</td>
<td>46/6.1/5.3</td>
</tr>
<tr>
<td>MLTP</td>
<td>14/3.0/2.2</td>
<td>32/6.5/4.8</td>
<td>28/4.8/4.6</td>
<td>26/2.3/5.0</td>
<td>13/1.9/2.3</td>
<td>23/4.7/3.6</td>
</tr>
<tr>
<td>GRUP2</td>
<td>15/2.7/2.7</td>
<td>36/5.8/7.0</td>
<td>25/3.6/4.4</td>
<td>24/2.2/5.6</td>
<td>21/1.1/2.6</td>
<td>33/3.6/3.6</td>
</tr>
<tr>
<td>BP3</td>
<td>9/3.5/1.9</td>
<td>20/6.2/4.0</td>
<td>22/5.5/4.1</td>
<td>22/1.0/4.7</td>
<td>3/0.0/0.6</td>
<td>21/7.7/3.9</td>
</tr>
<tr>
<td>BP6</td>
<td>12/2.8/2.6</td>
<td>36/5.9/7.7</td>
<td>25/4.0/4.9</td>
<td>24/2.0/4.9</td>
<td>3/0.5/1.0</td>
<td>11/4.0/2.9</td>
</tr>
<tr>
<td>SSP2</td>
<td>10/2.4/1.5</td>
<td>20/5.2/4.1</td>
<td>22/3.5/3.0</td>
<td>24/1.8/5.1</td>
<td>9/1.3/1.8</td>
<td>17/3.3/3.1</td>
</tr>
<tr>
<td>CUBN</td>
<td>9/2.8/1.7</td>
<td>18/6.1/3.6</td>
<td>17/4.5/3.4</td>
<td>16/1.7/4.6</td>
<td>26/4.2/5.8</td>
<td>20/4.1/3.6</td>
</tr>
<tr>
<td>PLY</td>
<td>16/3.2/2.2</td>
<td>33/7.0/5.3</td>
<td>23/4.8/4/2</td>
<td>23/2.7/5.2</td>
<td>22/2.6/3.2</td>
<td>14/3.8/3.3</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>12/3.7/2.4</td>
<td>26/7.3/5.0</td>
<td>27/5.2/5.2</td>
<td>18/2.7/5.1</td>
<td>10/0.8/1.4</td>
<td>23/5.8/4.5</td>
</tr>
<tr>
<td>RTD2</td>
<td>13/2.6/2.8</td>
<td>37/5.4/7.3</td>
<td>31/3.9/5.1</td>
<td>24/1.5/4.9</td>
<td>3/0.3/0.8</td>
<td>14/4.7/3.5</td>
</tr>
<tr>
<td>SPDFI</td>
<td>18/3.1/2.3</td>
<td>32/6.9/5.3</td>
<td>25/4.7/4.8</td>
<td>17/2.8/5.3</td>
<td>32/3.2/6.4</td>
<td>21/4.3/3.3</td>
</tr>
<tr>
<td>EXSM</td>
<td>6/2.9/1.6</td>
<td>13/6.0/3.4</td>
<td>14/4.4/2.8</td>
<td>21/1.9/5.3</td>
<td>12/1.7/2.5</td>
<td>24/4.8/4.4</td>
</tr>
<tr>
<td>SE35</td>
<td>11/2.1/1.3</td>
<td>20/4.7/3.7</td>
<td>14/3.2/2.3</td>
<td>21/1.6/4.9</td>
<td>11/1.4/2.2</td>
<td>19/3.2/2.5</td>
</tr>
<tr>
<td>SE35</td>
<td>11/2.1/1.3</td>
<td>20/4.7/3.7</td>
<td>14/3.2/2.3</td>
<td>21/1.6/4.9</td>
<td>11/1.4/2.2</td>
<td>19/3.2/2.5</td>
</tr>
<tr>
<td>MEBS</td>
<td>9/2.6/2.0</td>
<td>25/5.6/5.0</td>
<td>18/3.9/3.9</td>
<td>19/2.9/4.3</td>
<td>10/1.6/2.0</td>
<td>17/3.8/4.1</td>
</tr>
<tr>
<td>DGT3</td>
<td>11/3.2/2.0</td>
<td>22/7.3/5.8</td>
<td>19/4.8/4.0</td>
<td>30/2.6/6.1</td>
<td>12/1.9/1.9</td>
<td>22/4.4/4.4</td>
</tr>
<tr>
<td>JELF</td>
<td>9/3.1/1.9</td>
<td>22/6.9/5.5</td>
<td>14/4.6/3.8</td>
<td>18/2.4/5.2</td>
<td>10/1.1/1.6</td>
<td>19/4.9/3.3</td>
</tr>
<tr>
<td>MBS</td>
<td>17/2.5/2.5</td>
<td>27/5.2/4.4</td>
<td>27/3.8/4.7</td>
<td>16/1.7/3.7</td>
<td>15/2.2/3.0</td>
<td>21/3.1/4.1</td>
</tr>
<tr>
<td>QSF</td>
<td>8/2.5/1.4</td>
<td>20/5.2/3.4</td>
<td>12/3.5/2.3</td>
<td>19/1.9/4.9</td>
<td>10/1.8/2.2</td>
<td>16/3.2/2.3</td>
</tr>
<tr>
<td>SIMQ</td>
<td>12/2.8/2.4</td>
<td>30/6.5/5.4</td>
<td>26/4.5/5.0</td>
<td>22/2.3/5.1</td>
<td>34/3.1/5.7</td>
<td>43/4.2/4.8</td>
</tr>
<tr>
<td>MXRA</td>
<td>11/2.4/1.9</td>
<td>26/5.0/4.8</td>
<td>24/3.5/3.9</td>
<td>24/1.8/4.2</td>
<td>10/1.6/1.8</td>
<td>14/2.9/2.5</td>
</tr>
<tr>
<td>MEST</td>
<td>15/3.0/2.9</td>
<td>34/7.0/6.9</td>
<td>34/4.9/5.7</td>
<td>24/2.5/6.2</td>
<td>23/2.2/3.8</td>
<td>16/4.5/3.3</td>
</tr>
<tr>
<td>QATR</td>
<td>11/3.3/2.0</td>
<td>25/7.5/4.8</td>
<td>17/5.1/3.7</td>
<td>24/2.7/5.4</td>
<td>17/1.1/3.0</td>
<td>19/6.1/3.6</td>
</tr>
<tr>
<td>PECR</td>
<td>27/4.9/4.4</td>
<td>50/11.8/10.2</td>
<td>42/7.4/7.6</td>
<td>33/4.8/7.9</td>
<td>18/2.8/3.5</td>
<td>38/6.5/6.3</td>
</tr>
<tr>
<td>EL11</td>
<td>8/2.9/1.9</td>
<td>17/5.4/4.0</td>
<td>13/3.9/3.5</td>
<td>18/1.6/4.4</td>
<td>3/0.8/1.2</td>
<td>13/4.5/3.7</td>
</tr>
</tbody>
</table>

Data Path Requirements for 25 Programs
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Real Cells</th>
<th>Integer Cells</th>
<th>Structure Cells</th>
<th>Boolean Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEWB</td>
<td>11/6.54/3.0</td>
<td>1/0/0.95/0.2</td>
<td>2/0.17/0.6</td>
<td>0/0/0</td>
</tr>
<tr>
<td>PRPF</td>
<td>55/36.1/15.4</td>
<td>44/15.6/9.3</td>
<td>35/29.4/8.6</td>
<td>2/0.01/0.1</td>
</tr>
<tr>
<td>MLTP</td>
<td>20/13.7/4.9</td>
<td>17/9.5/4.2</td>
<td>16/13.0/3.8</td>
<td>0/0/0</td>
</tr>
<tr>
<td>GRUP2</td>
<td>32/24.1/7.5</td>
<td>11/9.5/1.1</td>
<td>26/19.8/6.6</td>
<td>2/1.1/0.81</td>
</tr>
<tr>
<td>BP3</td>
<td>25/18.9/6.2</td>
<td>0/0/0</td>
<td>2/0.1/0.3</td>
<td>0/0/0</td>
</tr>
<tr>
<td>BP6</td>
<td>15/10.2/4.6</td>
<td>1/0.6/0.5</td>
<td>2/0.2/0.6</td>
<td>0/0/0</td>
</tr>
<tr>
<td>SSF2</td>
<td>16/8.2/3.8</td>
<td>11/7.6/2.5</td>
<td>13/6.6/3.1</td>
<td>0/0/0</td>
</tr>
<tr>
<td>CUBN</td>
<td>59/29.2/17.0</td>
<td>18/11.8/5.5</td>
<td>36/21.1/10.1</td>
<td>0/0/0</td>
</tr>
<tr>
<td>PLY</td>
<td>15/8.1/3.5</td>
<td>12/6.4/3.1</td>
<td>12/6.9/4.0</td>
<td>2/0.01/0.15</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>25/16.6/8.6</td>
<td>11/7.5/2.3</td>
<td>13/8.9/3.2</td>
<td>2/0.6/0.91</td>
</tr>
<tr>
<td>RTD2</td>
<td>17/13.2/5.2</td>
<td>4/2.5/1.2</td>
<td>2/0.1/0.4</td>
<td>2/0.72/0.53</td>
</tr>
<tr>
<td>SPDI</td>
<td>25/4.1/7.2</td>
<td>35/22.4/6.4</td>
<td>49/20.4/13.4</td>
<td>21/0.51/0.79</td>
</tr>
<tr>
<td>EXSM</td>
<td>31/16.1/8.6</td>
<td>6/3.5/1.2</td>
<td>21/9.9/6.4</td>
<td>2/0.06/0.29</td>
</tr>
<tr>
<td>SE35</td>
<td>20/10.5/5.8</td>
<td>8/6.0/1.5</td>
<td>17/8.9/4.6</td>
<td>0/0/0</td>
</tr>
<tr>
<td>MEBS</td>
<td>16/6.9/4.4</td>
<td>12/8.9/3.7</td>
<td>11/5.1/3.9</td>
<td>0/0/0</td>
</tr>
<tr>
<td>DGT3</td>
<td>22/9.9/6.5</td>
<td>8/6.2/1.8</td>
<td>16/8.3/4.7</td>
<td>0/0/0</td>
</tr>
<tr>
<td>JELF</td>
<td>16/8.9/4.8</td>
<td>4/2.3/1.0</td>
<td>10/5.0/4.1</td>
<td>2/0.39/0.61</td>
</tr>
<tr>
<td>MVS</td>
<td>15/8.8/5.0</td>
<td>25/12.5/8.5</td>
<td>11/5.7/2.6</td>
<td>0/0/0</td>
</tr>
<tr>
<td>QSIF</td>
<td>18/9.9/4.6</td>
<td>5/3.5/1.2</td>
<td>17/8.3/4.7</td>
<td>0/0/0</td>
</tr>
<tr>
<td>SIMQ</td>
<td>31/18.0/8.3</td>
<td>25/13.6/5.2</td>
<td>13/9.3/5.8</td>
<td>2/0.85/0.33</td>
</tr>
<tr>
<td>MXRA</td>
<td>27/17.8/7.9</td>
<td>14/11.1/1.5</td>
<td>21/14.9/6.9</td>
<td>3/0.19/0.54</td>
</tr>
<tr>
<td>MEST</td>
<td>60/32.2/16.8</td>
<td>15/11.9/2.1</td>
<td>10/5.7/1.4</td>
<td>3/0.12/0.51</td>
</tr>
<tr>
<td>QATR</td>
<td>29/21.1/7.1</td>
<td>13/9.6/3.5</td>
<td>6/4.1/1.6</td>
<td>2/0.66/0.51</td>
</tr>
<tr>
<td>PECR</td>
<td>21/12.6/6.6</td>
<td>25/11.9/7.1</td>
<td>15/7.0/5.9</td>
<td>0/0/0</td>
</tr>
<tr>
<td>EL11</td>
<td>8/4.3/2.5</td>
<td>2/1.2/0.5</td>
<td>2/0.3/0.7</td>
<td>0/0/0</td>
</tr>
</tbody>
</table>

Data Memory Requirements for the 25 Programs
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Identity</th>
<th>Merge</th>
<th>Cons</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEWB</td>
<td>4/1.33/0.47</td>
<td>4/2.50/1.50</td>
<td>9/5.00/4.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PRPF</td>
<td>10/1.88/1.89</td>
<td>13/2.81/2.43</td>
<td>10/2.31/2.82</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MLTP</td>
<td>5/2.11/1.74</td>
<td>7/2.22/1.62</td>
<td>8/2.66/1.88</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>GRUP2</td>
<td>13/3.91/3.43</td>
<td>12/1.62/1.66</td>
<td>8/2.00/1.89</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>BP3</td>
<td>1/1.00/0.00</td>
<td>7/2.00/2.18</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>BP6</td>
<td>10/5.72/3.46</td>
<td>10/2.62/2.19</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SSP2</td>
<td>8/3.42/1.63</td>
<td>4/1.41/0.68</td>
<td>5/3.33/1.25</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>CUBN</td>
<td>5/3.16/1.96</td>
<td>6/1.46/0.88</td>
<td>7/1.67/1.89</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PLY</td>
<td>6/2.34/1.71</td>
<td>5/2.23/1.54</td>
<td>8/2.18/2.21</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>4/1.69/1.06</td>
<td>8/2.10/1.99</td>
<td>4/2.73/0.83</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>RTD2</td>
<td>13/7.04/5.36</td>
<td>5/1.78/1.29</td>
<td>2/1.50/0.50</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SPDI</td>
<td>8/2.70/1.95</td>
<td>9/2.23/1.96</td>
<td>10/2.23/2.42</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>EXSM</td>
<td>5/2.39/1.44</td>
<td>4/1.55/0.92</td>
<td>5/5.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SE35</td>
<td>4/3.00/1.33</td>
<td>4/1.34/0.68</td>
<td>4/3.50/0.50</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MEBS</td>
<td>4/2.59/1.43</td>
<td>7/1.56/1.37</td>
<td>6/6.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>DGT3</td>
<td>5/2.50/1.50</td>
<td>6/1.78/1.28</td>
<td>4/4.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>JELF</td>
<td>8/3.33/2.21</td>
<td>4/1.62/0.90</td>
<td>5/3.33/1.69</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MBS</td>
<td>6/1.95/1.29</td>
<td>10/1.78/1.45</td>
<td>4/3.00/1.41</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>QSF</td>
<td>4/2.87/1.41</td>
<td>7/1.50/1.01</td>
<td>4/4.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SIMQ</td>
<td>9/2.56/2.06</td>
<td>9/1.85/1.64</td>
<td>9/1.89/2.51</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MRRA</td>
<td>9/3.17/2.13</td>
<td>9/1.64/1.55</td>
<td>5/2.14/1.36</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MEST</td>
<td>12/2.91/3.57</td>
<td>11/2.15/2.45</td>
<td>11/2.74/2.36</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>QATR</td>
<td>5/1.91/1.38</td>
<td>5/2.34/1.43</td>
<td>5/3.50/1.22</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PECR</td>
<td>14/2.41/2.83</td>
<td>14/2.85/2.52</td>
<td>7/3.25/1.98</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>ELI1</td>
<td>5/2.09/1.31</td>
<td>1/1.00/0.00</td>
<td>5/3.50/1.50</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>Program Name</td>
<td>Write</td>
<td>+</td>
<td>-</td>
<td>*</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>WEWB</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PRPF</td>
<td>1/1.00/0.00</td>
<td>8/1.41/1.04</td>
<td>2/1.12/0.33</td>
<td>10/1.25/0.98</td>
</tr>
<tr>
<td>MLTP</td>
<td>1/1.00/0.00</td>
<td>5/1.38/0.76</td>
<td>1/1.00/0.00</td>
<td>3/1.53/0.56</td>
</tr>
<tr>
<td>GRUP2</td>
<td>1/1.00/0.00</td>
<td>2/1.05/0.22</td>
<td>2/1.44/0.49</td>
<td>2/1.44/0.49</td>
</tr>
<tr>
<td>BP3</td>
<td>1/1.00/0.00</td>
<td>3/1.20/0.45</td>
<td>2/1.22/0.41</td>
<td>4/1.40/0.80</td>
</tr>
<tr>
<td>BP6</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>2/1.25/0.43</td>
<td>2/1.09/0.30</td>
</tr>
<tr>
<td>SSP2</td>
<td>1/1.00/0.00</td>
<td>3/1.18/0.46</td>
<td>3/1.17/0.53</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>CUBN</td>
<td>1/1.00/0.00</td>
<td>4/1.67/0.80</td>
<td>3/1.29/0.49</td>
<td>2/1.25/0.43</td>
</tr>
<tr>
<td>PLY</td>
<td>-</td>
<td>2/1.32/0.47</td>
<td>2/1.69/0.46</td>
<td>3/1.21/0.61</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>1/1.00/0.00</td>
<td>4/1.53/0.75</td>
<td>1/1.00/0.00</td>
<td>2/1.21/0.41</td>
</tr>
<tr>
<td>RTD2</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>2/1.29/0.46</td>
<td>4/2.10/1.07</td>
</tr>
<tr>
<td>SPDI</td>
<td>1/1.00/0.00</td>
<td>5/1.28/0.61</td>
<td>2/1.25/0.43</td>
<td>3/1.58/0.53</td>
</tr>
<tr>
<td>EXSM</td>
<td>1/1.00/0.00</td>
<td>2/1.34/0.48</td>
<td>2/1.28/0.45</td>
<td>2/1.21/0.41</td>
</tr>
<tr>
<td>SE35</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>2/1.03/0.18</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MEBS</td>
<td>1/1.00/0.00</td>
<td>3/1.28/0.51</td>
<td>2/1.13/0.33</td>
<td>2/1.31/0.46</td>
</tr>
<tr>
<td>DGT3</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>4/1.83/1.12</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>JELF</td>
<td>1/1.00/0.00</td>
<td>2/1.31/0.46</td>
<td>1/1.00/0.00</td>
<td>2/1.12/0.32</td>
</tr>
<tr>
<td>MVS</td>
<td>1/1.00/0.00</td>
<td>4/1.42/0.91</td>
<td>4/1.63/0.99</td>
<td>4/1.21/0.77</td>
</tr>
<tr>
<td>QSF</td>
<td>1/1.00/0.00</td>
<td>2/1.12/0.33</td>
<td>1/1.00/0.00</td>
<td>3/1.12/0.47</td>
</tr>
<tr>
<td>SIMQ</td>
<td>1/1.00/0.00</td>
<td>4/1.36/0.65</td>
<td>3/1.29/0.63</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MXRA</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>2/1.12/0.33</td>
<td>2/2.00/0.00</td>
</tr>
<tr>
<td>MEST</td>
<td>-</td>
<td>2/1.15/0.36</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>QATR</td>
<td>1/1.00/0.00</td>
<td>4/1.59/0.83</td>
<td>2/1.23/0.42</td>
<td>2/1.08/0.28</td>
</tr>
<tr>
<td>PECR</td>
<td>1/1.00/0.00</td>
<td>4/1.50/0.76</td>
<td>1/1.00/0.00</td>
<td>2/1.09/0.29</td>
</tr>
<tr>
<td>ELII</td>
<td>1/1.00/0.00</td>
<td>2/1.50/0.50</td>
<td>1/1.00/0.00</td>
<td>2/1.39/0.49</td>
</tr>
<tr>
<td>Program Name</td>
<td>/</td>
<td>.**</td>
<td>&lt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>----------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>WEEWB</td>
<td>4/2.00/1.42</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PRPF</td>
<td>2/1.04/0.20</td>
<td>3/1.09/0.38</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MLTP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>GRUP2</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>BP3</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>BP6</td>
<td>2/1.25/0.43</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SSP2</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>CUBN</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PLY</td>
<td>2/1.33/0.47</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>2/1.04/0.20</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>RTD2</td>
<td>1/1.00/0.00</td>
<td>4/2.79/1.04</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SPDI</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>2/1.04/0.19</td>
</tr>
<tr>
<td>EXSM</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SE35</td>
<td>2/1.09/0.28</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MEBS</td>
<td>2/1.50/0.50</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>DGT3</td>
<td>2/1.83/0.37</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>JELF</td>
<td>1/1.20/0.40</td>
<td>-</td>
<td>2/1.14/0.35</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MVS</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>QSF</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SIMQ</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MXRA</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MEST</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>QATR</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PE CR</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>ELII</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>Program Name</td>
<td>Select</td>
<td>Append</td>
<td>Logarithm</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>--------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>WEWB</td>
<td>1/1.00/0.00</td>
<td>2/2.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PRPF</td>
<td>-</td>
<td>4/1.33/0.61</td>
<td>2/1.07/0.25</td>
<td>-</td>
</tr>
<tr>
<td>MLTP</td>
<td>-</td>
<td>6/1.81/0.72</td>
<td>2/1.09/0.29</td>
<td>-</td>
</tr>
<tr>
<td>GRUP2</td>
<td>-</td>
<td>2/1.50/0.50</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>BP3</td>
<td>-</td>
<td>2/2.00/0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BP6</td>
<td>1/1.00/0.00</td>
<td>2/2.00/0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SSP2</td>
<td>-</td>
<td>5/1.56/0.85</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>CUBN</td>
<td>-</td>
<td>5/1.79/0.69</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>PLY</td>
<td>-</td>
<td>4/1.63/0.84</td>
<td>2/1.03/0.16</td>
<td>-</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>1/1.00/0.00</td>
<td>2/1.33/0.47</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>RTD2</td>
<td>-</td>
<td>2/2.00/0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SPD1</td>
<td>-</td>
<td>2/1.14/0.35</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>EXSM</td>
<td>-</td>
<td>6/1.82/0.81</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>SE35</td>
<td>-</td>
<td>4/1.70/0.71</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>MBES</td>
<td>-</td>
<td>5/2.32/0.98</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>DGT3</td>
<td>-</td>
<td>4/1.90/0.61</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>JELF</td>
<td>1/1.00/0.00</td>
<td>2/2.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>MVS</td>
<td>-</td>
<td>9/1.84/1.56</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>QSF</td>
<td>-</td>
<td>4/1.73/0.71</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>SIMQ</td>
<td>1/1.00/0.00</td>
<td>3/1.49/0.61</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>MKRA</td>
<td>-</td>
<td>4/1.66/0.57</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>MEST</td>
<td>-</td>
<td>3/1.32/0.51</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>QATR</td>
<td>-</td>
<td>2/1.17/0.38</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>PEcr</td>
<td>1/1.00/0.00</td>
<td>3/1.56/0.61</td>
<td>2/1.12/0.31</td>
<td>-</td>
</tr>
<tr>
<td>EL11</td>
<td>-</td>
<td>2/2.00/0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Program Name</td>
<td>&gt;</td>
<td>&gt;=</td>
<td>And</td>
<td>Or</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td>----</td>
<td>------</td>
<td>----</td>
</tr>
<tr>
<td>WEWB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PRPF</td>
<td>3/1.09/0.34</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>MLTP</td>
<td>2/1.03/0.17</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRUP2</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>BP3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BP6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SSP2</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CUBN</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PLY</td>
<td>2/1.03/0.18</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>RTD2</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>SPD1</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>EXSM</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>SE35</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MEBS</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DGT3</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JELF</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>MVS</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>QSF</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SIMQ</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MXRA</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MEST</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>QATR</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PECR</td>
<td>2/1.05/0.23</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ELi1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Program Name</td>
<td>SQRT</td>
<td>Negate</td>
<td></td>
<td>Cosine</td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>--------</td>
<td>---</td>
<td>---------</td>
</tr>
<tr>
<td>WEDB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PRPF</td>
<td>1/1.00/0.00</td>
<td>3/1.25/0.66</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MLTP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GRUP2</td>
<td>-</td>
<td>3/1.50/0.87</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>BP3</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>BP6</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>SSP2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CUBN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PLY</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>GNFLT1</td>
<td>-</td>
<td>2/1.50/0.50</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>RTD2</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SPD1</td>
<td>-</td>
<td>3/1.33/0.75</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>EXSM</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>SE35</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MEBS</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DGT3</td>
<td>-</td>
<td>-</td>
<td>2/1.83/0.37</td>
<td>-</td>
</tr>
<tr>
<td>JELF</td>
<td>2/1.83/0.37</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>MVS</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>QSF</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SLMQ</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>MXRA</td>
<td>-</td>
<td>3/2.00/1.00</td>
<td>3/1.33/0.75</td>
<td>-</td>
</tr>
<tr>
<td>MEST</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>QATR</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
</tr>
<tr>
<td>PECR</td>
<td>-</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
</tr>
<tr>
<td>ELII</td>
<td>2/2.0/0.00</td>
<td>-</td>
<td>1/1.0/0.00</td>
<td>-</td>
</tr>
<tr>
<td>Program Name</td>
<td>Sine</td>
<td>Absolute</td>
<td>Apply</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>----------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>WEWB</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>PRPF</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>MLTP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>GRUP2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>BP3</td>
<td>3/1.32/0.52</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>BP6</td>
<td>2/1.33/0.47</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>SSP2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>CUBN</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>PLY</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td></td>
</tr>
<tr>
<td>GNFLT1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>RTD2</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>SPDI</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>EXSM</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>SE35</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>MEBS</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>DGT3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>JELF</td>
<td>2/2.00/0.00</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>MVS</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>QSF</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>SIMQ</td>
<td>-</td>
<td>2/1.25/0.43</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>MXRA</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>MEST</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>QA6TK</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>1/1.00/0.00</td>
<td></td>
</tr>
<tr>
<td>PE6CR</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>EL21</td>
<td>-</td>
<td>1/1.00/0.00</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>