An Automatic Approach To Implementing DSP Algorithms On Parallel Processors

by

Alan John Goddard

A Thesis submitted for the degree of Doctor of Philosophy

CITY UNIVERSITY
Centre for Information Engineering
April, 1990.

Contents

Chapter 1. Introduction

1.1 Processor architectures 1-2
1.1.1 Von Neumann machine 1-2
1.1.2 Array machines 1-3
1.1.3 Parallel von Neumann machines 1-5
1.1.4 Non von Neumann machines 1-6
1.1.5 Dataflow machines 1-7
1.1.6 Graph reduction machines 1-7
1.2 Interconnection Networks 1-8
1.2.1 Bus interconnection 1-8
1.2.2 Crossbar interconnection 1-9
1.2.3 Omega interconnection 1-9
1.2.4 Static interconnection 1-10
1.3 Programming languages 1-12
1.3.1 Array and vector programming 1-12
1.3.2 Multiprocessor programming 1-13
1.3.3 Non-von Neumann programming 1-14
1.4 Implementation strategies 1-15
1.5 Thesis outline 1-19

Chapter 2. Discrete algorithms and their graphs

2.1 The characteristics of discrete algorithms 2-1
2.1.1 Discreteness 2-1
2.1.2 Competence-performance trade-off 2-1
2.1.3 Inputs and outputs 2-2
2.1.4 Memory 2-2
2.1.5 Complexity 2-2
2.1.6 Real-time, deterministic, synchronising systems 2-3
2.1.7 Granularity 2-4
2.1.8 Task primitives 2-4
2.2 Data flow graphs 2-5
2.2.1 Nodes and arcs 2-5
2.2.2 Paths and acyclic paths 2-6
2.2.3 Initiating and terminating DAGs 2-7
2.2.4 Dependence and independence 2-8
2.2.5 Path costs and the critical path 2-9
2.2.6 Earliest and latest start times and float 2-10
2.3 Summary 2-11

Contents

Chapter 3. DFDL design aspects

3.1 Language grammar 3-1
3.2 Single-assignment 3-3
3.3 Program flow 3-4
3.4 Past values 3-5
3.5 Error handling 3-5
3.6 Data types 3-5
3.7 Scope and use of variables 3-6
3.8 Program units 3-6
3.9 Summary 3-7

Chapter 4. DFDL definition and syntax

4.1 BNF notation 4-1
    4.1.1 Production 4-1
    4.1.2 Alternative 4-1
    4.1.3 Repetition 4-2
4.2 Program facilities 4-2
    4.2.1 Continuation 4-2
    4.2.2 Comments 4-2
4.3 Lexical types 4-3
    4.3.1 Alphabetic characters (letters) 4-3
    4.3.2 Numeric characters (digits) 4-3
    4.3.3 Special characters 4-3
    4.3.4 Delimiters 4-3
    4.3.5 Identifiers 4-3
    4.3.6 Reserved words 4-4
    4.3.7 Integer numbers 4-4
    4.3.8 Real numbers 4-4
    4.3.9 Real errors and overflow 4-5
4.4 Syntax 4-5
    4.4.1 Operators and functions 4-5
        4.4.1.1 Arithmetic operators 4-5
        4.4.1.2 Boolean operators 4-6
        4.4.1.3 Relational operators 4-6
        4.4.1.4 Functions 4-7
    4.4.2 Program 4-8
    4.4.3 External input and output 4-9
    4.4.4 Nodes 4-10
    4.4.5 Constants 4-11
    4.4.6 User-defined functions 4-12
    4.4.7 Z-operator (Delay) 4-13
    4.4.8 Assignment 4-14

Contents

4.4.9 Expressions 4-16
4.4.10 Repetitive and fixed subscripts 4-17
4.4.11 Conditional 4-18

4.5 DFDL examples 4-19
4.5.1 FIR filter 4-19
4.5.2 IIR filter 4-19
4.5.3 2-D convolution 4-20
4.5.4 DFT 4-21
4.5.5 Lattice filter 4-22
4.5.6 Level-crossing detector 4-23
4.5.7 Matrix product 4-23
4.6 Summary 4-24

Chapter 5. DFDL task model

5.1 Data structures 5-1
5.1.1 Executable node data structure 5-1
5.1.2 Non-executable node data structure 5-2
5.1.3 Executable node attributes 5-3
5.1.4 Attribute description 5-4
5.2 Node types 5-5
5.3 Named graph structures 5-7
5.3.1 Input nodes 5-7
5.3.2 Output nodes 5-10
5.3.3 Node nodes 5-13
5.3.4 Constant nodes 5-14
5.4 Primitive graph structures 5-15
5.4.1 Real nodes 5-15
5.4.2 Internal input/output and delay nodes 5-17
5.4.3 Arithmetic nodes 5-20
5.4.4 Branch nodes 5-22
5.4.5 Function nodes 5-23
5.4.6 Communication nodes 5-24
5.4.7 Relational nodes 5-25
5.4.8 Boolean nodes 5-26
5.4.9 Conditional nodes 5-27
5.5 Non-primitive graph structures 5-29
5.5.1 Power and anti-logarithm graph structures 5-29
5.5.2 Sum, product and mean graph structures 5-30
5.5.3 Conditional graph structures 5-32
5.5.4 Maximum and minimum graph structures 5-33
5.5.5 Median graph structure 5-34
5.6 Summary 5-35

Contents

Chapter 6. Parallel processor model

6.1 The Transputer 6-1
6.2 Resources 6-2
6.3 Activity 6-3
6.4 Data structures 6-3
6.4.1 Processor graph data structure 6-4
6.4.2 Activity schedules data structure 6-5
6.5 Processor graph 6-6
6.5.1 Degree 6-6
6.5.2 Arc relationships 6-6
6.5.3 Connectivity 6-7
6.5.4 Connecting the nets in **V** 6-8
6.6 Checking for net isolation 6-9
6.7 Summary 6-9

Chapter 7. Compile-time scheduling

7.1 Scheduling model 7-2
7.1.1 Task graph 7-2
7.1.2 Processor graph 7-3
7.1.3 Sequence constraints within schedules 7-4
7.1.4 Performance measures 7-8
7.1.5 Definition of the scheduling problem 7-9
7.2 Scheduling complexity 7-9
7.2.1 Ordering on **T** is empty 7-10
7.2.2 Ordering on **T** is strictly sequential 7-10
7.2.3 Non-zero communication costs 7-11
7.2.4 Exhaustive enumeration 7-13
7.2.5 P and NP problems 7-14
7.3 Search strategies 7-15
7.3.1 Solution space 7-15
7.3.2 Generating, expanding and exploring solution 7-15
7.3.3 Informed and uninformed search 7-15
7.3.4 Backtracking 7-16
7.3.5 Hill-climbing 7-17
7.3.6 Best-first 7-18
7.3.7 Hybrid search 7-18

7.4 Scheduling algorithm 7-19
7.4.1 Heuristic measures 7-20
7.4.1.1 Minimum length heuristic 7-20
7.4.1.2 Minimum lateness heuristic 7-22
7.4.2 Shortest path, a BF approach 7-24

Contents

7.4.2.1 Dijkstra's shortest path algorithm 7-24
7.4.2.2 Modified Dijkstra's shortest path algorithm 7-25
7.4.2.3 Extension 1 7-26
7.4.2.4 Extension 2 7-26
7.4.2.5 Extension 3 7-26
7.4.2.6 Extension 4 7-26
7.4.3 Comparing task solution nodes, a BT approach 7-27
7.4.4 Scheduling assignment, an HC approach 7-28
7.4.5 Complexity 7-29
7.5 Summary 7-30

Chapter 8. Experimental results

8.1 Scheduling time 8-2
8.2 Factors affecting performance 8-4
8.2.1 Integer effect 8-4
8.2.2 Synchronisation effect 8-6
8.2.3 Latent scope effect 8-6
8.2.4 Finite communication cost effect 8-8
8.2.5 Topology effect 8-13
8.3 Examples 8-15
8.3.1 Finite impulse response filter 8-16
8.3.2 Infinite impulse response filter 8-22
8.3.3 Wave digital filter 8-26
8.3.4 Fast Fourier transform 8-32
8.3.5 Multiple cycle finite impulse response filter 8-37

Chapter 9. Conclusions

9.1 Implementation summary 9-1
9.1.1 Algorithm characteristics 9-1
9.1.2 Algorithm representation 9-2
9.1.3 Language description 9-2
9.1.4 Task graph 9-3
9.1.5 Processor graph 9-3
9.1.6 Compile-time scheduling 9-4
9.2 Concluding remarks 9-5
9.2.1 Performance bounds 9-5
9.2.2 Factors affecting performance 9-5
9.2.2.1 Irrevocable scheduling 9-5
9.2.2.2 Integer effect 9-6
9.2.2.3 Synchronisation and latent scope effects 9-6
9.2.2.4 Communication cost and topology effects 9-6
9.2.3 Conclusions on performance 9-7

Contents

9.2.4 Conclusions on DFDL 9-7
9.3 Future work 9-8
9.3.1 Reducing scheduling time 9-8
9.3.2 Reducing scheduling memory 9-8
9.3.3 Improving the scheduler 9-8
9.3.4 DFDL program linker 9-8
9.3.5 Inputs and outputs 9-9
9.3.6 Language model 9-9
9.3.7 Deadlock avoidance 9-9
9.3.8 Processor graph definition 9-10

Appendix A. Graph concepts and definitions

Appendix B. EBNF description of DFDL

B.1 EBNF description of DFDL lexical analyser B-1
B.2 EBNF description of DFDL syntax B-3
B.2.1 Program B-3
B.2.2 External and Internal declarations B-3
B.2.3 Input, output and node declaration B-3
B.2.4 Constant declaration B-3
B.2.5 User defined function declaration B-4
B.2.6 Assignments B-4
B.2.7 Initialise assignment B-4
B.2.8 Constant expressions B-4
B.2.9 Constant operands B-4
B.2.10 Subscripts B-5
B.2.11 Real arithmetic operators B-5
B.2.12 Functions B-5
B.2.13 Repeat assignment B-5
B.2.14 Objects B-5
B.2.15 Expressions B-6
B.2.16 Operands B-6
B.2.17 Conditionals B-6
B.2.18 Boolean expressions B-6
B.2.19 Relational expressions B-6
B.2.20 Relational operators B-6
B.2.21 Boolean operators B-6

Contents

Appendix C. User's guide

C.1 Installation C-1
C.2 Getting Started C-1
C.3 Running the DFDL compiler C-2
C.4 Exiting C-2
C.5 Making a DFDL source file C-2
C.6 Compiler flow C-3
C.6.1 Processor type C-3
C.6.2 Clock frequency C-3
C.6.3 Link speed C-3
C.6.4 Number of processors C-3
C.6.5 Connect links C-5
C.6.6 View Connections C-5
C.6.7 Schedule C-5
C.6.8 Schedule display C-6
C.6.9 Timing display C-8
C.6.10 Data analysis C-10
C.6.11 Translate C-11

Appendix D. Programmer's guide

D.1 Reading the source file D-1
D.1.1 File name D-1
D.1.2 File contents D-1
D.2 Lexical analysis D-2
D.2.1 Protocol &Lex-&Syntax D-2
D.3 Syntax analysis D-3
D.3.1 Initialise D-4
D.3.1.1 Variables D-4
D.3.1.2 Reserved word initialisation D-4
D.3.1.3 Hash function D-5
D.3.1.4 Appending the symbol table D-5
D.3.1.5 Chaining D-5
D.3.1.6 Symbol table format D-6
D.3.1.7 Symbol table limits D-7
D.3.2 Syntax analysis D-7
D.4 Syntax analyser description D-8
D.4.1 Declaration flow D-8
D.4.1.1 Input, node and output flow D-8
D.4.1.2 Data type flow D-10
D.4.1.3 Subscript size flow D-11
D.4.1.4 Constant flow D-11
D.4.1.5 Real string flow D-13

Contents

D.4.1.6 User defined function flow D-13
D.4.2 Assignment flow D-13
D.4.2.1 Initialise flow D-13
D.4.2.2 Repeat flow D-14

List of tables

Chapter 1. Introduction

Table 1.1 Implementation strategies 1-15

Chapter 4. DFDL definition and syntax

Table 4.1 Valid initialisation section assignment 4-14
Table 4.2 Valid repetitive section assignment 4-14

Chapter 5. DFDL task model

Table 5.1 In/out-degree of task graph nodes 5-1
Table 5.2 Node format 5-3
Table 5.3 Task primitives, in/out-degree and cost 5-5
Table 5.4 Attributes for external input nodes 5-8
Table 5.5 Attributes for input conversion nodes 5-9
Table 5.6 Attributes for external output nodes 5-11
Table 5.7 Attributes for output conversion nodes 5-12
Table 5.8 Attributes for node nodes 5-13
Table 5.9 Attributes for constant nodes 5-15
Table 5.10 Attributes for real nodes 5-16
Table 5.11 Attributes for internal output nodes 5-17
Table 5.12 Attributes for internal input nodes 5-17
Table 5.13 Attributes for delay nodes 5-18
Table 5.14 Attributes for arithmetic nodes 5-20
Table 5.15 Attributes for branch nodes 5-21
Table 5.16 Attributes for function nodes 5-22
Table 5.17 Attributes for relational nodes 5-24
Table 5.18 Attributes for boolean nodes 5-25
Table 5.19 Truth table for GATE node 5-26
Table 5.20 Truth table for PRI.OR node 5-26
Table 5.21 Attributes for GATE node 5-27
Table 5.22 Attributes for PRI.OR node 5-27

Chapter 6. Parallel processor model

Table 6.1 Degree of node types 6-6
Table 6.2 Valid processor graph arcs 6-7

Chapter 7. Compile-time scheduling

Table 7.1 Polynomial and Non-polynomial complexity 7-13

List of tables

Chapter 8. Experimental results

Table 8.1 Scheduling time scale factor 8-3
Table 8.2 Topology factor, Q 8-14

Appendix D. Programmer's guide

Table D.1 Concurrent compiler operation D-3

List of figures

Chapter 1. Introduction

Figure 1.1 Von Neumann architecture 1-2
Figure 1.2 Array architecture 1-3
Figure 1.3 Array architecture 1-4
Figure 1.4 Systolic array architecture 1-4
Figure 1.5 Wavefront array architecture 1-5
Figure 1.6 Loosely coupled architecture 1-5
Figure 1.7 Tightly coupled architecture 1-6
Figure 1.8 Static dataflow architecture 1-7
Figure 1.9 Dynamic dataflow/Reduction architecture 1-8
Figure 1.10 Bus interconnection 1-8
Figure 1.11 Full connectivity 1-9
Figure 1.12 Omega network 1-10
Figure 1.13 3-D Hypercube 1-11
Figure 1.14 Compile-time implementation 1-18
Figure 1.15 Implementation strategy by chapter 1-19

Chapter 2. Discrete algorithms and their graphs

Figure 2.1 Algorithm input-output 2-2
Figure 2.2 Input-process-output synchronisation 2-3
Figure 2.3 Coarse grain structure 2-4
Figure 2.4 Fine grain structure 2-4
Figure 2.5 Task input, output and execution cost 2-5
Figure 2.6 Arc relationships 2-6
Figure 2.7 Acyclic path 2-7
Figure 2.8 Directed acyclic graph (DAG) 2-8
Figure 2.9 DAG, showing the critical path 2-9
Figure 2.10 DAG, showing EST(T), LST(T) and FLT(T) 2-10

Chapter 3. DFDL design aspects

Figure 3.1 Program source and object 3-1
Figure 3.2 DFDL program structure models 3-4

Chapter 5. DFDL task model

Figure 5.1 B and E data structure 5-2
Figure 5.2 External input stream (REAL32) 5-8
Figure 5.3 External input stream (non-REAL32) 5-9
Figure 5.4 External output stream (REAL32) 5-11
Figure 5.5 External output stream (non-REAL32) 5-12

List of figures

Figure 5.6 Node nodes 5-13
Figure 5.7 Constant nodes 5-14
Figure 5.8 Real node 5-16
Figure 5.9 Single delay 5-17
Figure 5.10 Multiple delay 5-19
Figure 5.11 Monadic arithmetic node 5-20
Figure 5.12 Dyadic arithmetic nodes 5-20
Figure 5.13 Branch node 5-22
Figure 5.14 Function nodes 5-23
Figure 5.15 Communication nodes 5-24
Figure 5.16 Relational nodes 5-25
Figure 5.17 Boolean nodes 5-26
Figure 5.18 Conditional nodes 5-27
Figure 5.19 Power graph structure 5-29
Figure 5.20 Power graph structure 5-30
Figure 5.21 (a) SUM (b) PROD (c) MEAN graph structures 5-31
Figure 5.22 Conditional graph structure 5-32
Figure 5.23 Maximum graph structure 5-33
Figure 5.24 Minimum graph structure 5-33
Figure 5.25 Median graph structure 5-34

Chapter 6. Parallel processor model

Figure 6.1 INMOS Transputer 6-1
Figure 6.2 Transputer interconnection network 6-2
Figure 6.3 Gantt chart 6-3
Figure 6.4 Processor graph of Transputer network 6-3
Figure 6.5 Processor graph data structure 6-4
Figure 6.6 Schedule data structure 6-5
Figure 6.7 Partially connected processor graph 6-8

Chapter 7. Compile-time scheduling

Figure 7.1 Task graph, **G** 7-4
Figure 7.2 Processor graph, **V** 7-5
Figure 7.3 Scheduling methods 7-5
Figure 7.4 Task scheduling 7-7
Figure 7.5 Completely connected network 7-11
Figure 7.6 Linear connected network 7-12
Figure 7.7 2-dimensional space of hybrid strategies 7-18
Figure 7.8 Scheduling; minimum length heuristic 7-20
Figure 7.9 Scheduling; minimum lateness heuristic 7-22

List of figures

Chapter 8. Experimental results

Figure 8.1 Scheduling time characteristics (linear) 8-2
Figure 8.2 Scheduling time characteristics (log) 8-3
Figure 8.3 Integer effect on speedup 8-4
Figure 8.4 Speedup degradation 8-5
Figure 8.5 Cyclic processor activity 8-7
Figure 8.6 Cyclic processor and comms activity (proc/comms ratio = 30.0) 8-9
Figure 8.7 Cyclic processor and comms activity (proc/comms ratio = 15.0) 8-9
Figure 8.8 Cyclic processor and comms activity (proc/comms ratio = 6.0) 8-10
Figure 8.9 Cyclic processor and comms activity (proc/comms ratio = 3.0) 8-10
Figure 8.10 Cyclic processor and comms activity (proc/comms ratio = 1.5) ... 8-11
Figure 8.11 Cyclic processor and comms activity (proc/comms ratio = 0.3) ... 8-11
Figure 8.12 Number of communications vs. proc/comms cost ratio 8-12
Figure 8.13 Speedup vs. proc/comms cost ratio 8-13
Figure 8.14 Topology effect on speedup 8-14
Figure 8.15 Pre-schedule activity profile (fir8) 8-16
Figure 8.16 Speedup vs. number of processors (fir8) 8-17
Figure 8.17 Post-schedule activity profile (fir8) 8-21
Figure 8.18 Pre-schedule activity profile (iir2) 8-22
Figure 8.19 Speedup vs. number of processors (iir2) 8-23
Figure 8.20 Post-schedule activity profile (iir2) 8-26
Figure 8.21 Pre- schedule activity profile (wdf4) 8-28
Figure 8.22 Speedup vs. number of processors (wdf4) 8-28
Figure 8.23 Post-schedule activity profile (wdf4) 8-31
Figure 8.24 Pre-schedule activity profile (fft8) 8-33
Figure 8.25 Speedup vs. number of processors (fft8) 8-33
Figure 8.26 Post- schedule activity profile (fft8) 8-35
Figure 8.27 Pre-schedule activity profile (fir8x8) 8-37
Figure 8.28 Speedup vs. number of processors (fir8x8) 8-37
Figure 8.29 Post-schedule activity profile (fir8x8) 8-39

Appendix C. User's guide

Figure C.1 Compiler flow diagram C-4
Figure C.2 Schedule display C-7
Figure C.3 Timing display C-9

Appendix D. Programmer's guide

Figure D.1 Program flow D-8
Figure D.2 Declaration flow D-8
Figure D.3 Input flow D-9
Figure D.4 Node flow D-9

List of figures

Figure D.5 Output flow D-10
Figure D.6 Data type flow D-10
Figure D.7 Subscript size flow D-11
Figure D.8 Constant flow D-12
Figure D.9 Real string flow D-13
Figure D.10 Assignment flow D-13
Figure D.11 Repeat flow D-14
Figure D.12 Parse flow D-19
Figure D.13 Continue flow D-20
Figure D.14 Parse left identifier flow D-21
Figure D.15 Parse left real flow D-22
Figure D.16 Parse right identifier flow D-23
Figure D.17 Parse right real flow D-23
Figure D.18 Function flow D-24
Figure D.19 Monadic flow D-25
Figure D.20 Spatial subscript flow D-26
Figure D.21 Temporal subscript flow D-26
Figure D.22 Parse subscript flow D-27
Figure D.23 Integer expression flow D-28

Acknowledgements

I would like to thank the following people for their support and contributions during my research:

My supervisor, Dr. Stuart Lawson, for the initial inspiration that started me on the project, his helpful comments along the way and for keeping me on track; Prof. Tony Davies, Dr. Dick Comley and Dr. Pat Samwell for their advice on all matters and for making City University a friendly environment in which to work; and Martik Babians, Mike Roberts, Salim Omarouayache, and others for their help, friendship and comments throughout my time at City.

Last but not least, I would like to thank my fiancee, Wanda (S.P.), whose understanding during those moments of uncertainty kept me going and who can brighten up the bleakest of days.

The author was supported by the SERC (Science and Education Research Council) for 3 years.

Declaration

I grant powers of discretion to the University librarian to allow this thesis to be copied in whole or in part without further reference to me . This permission covers single copies made for study purposes, subject to normal conditions of acknowledgement.

Abstract

Recently, there has been an increase in demand for low cost, high throughput parallel processors on which to implement real-time DSP applications. Numerous solutions have been proposed, though often these are application dependent. This is true for SIMD and systolic architectures, which require a high degree of regularity in an application's structure. A more general purpose solution is offered using MIMD architectures, which come in a variety of forms. Here we concentrate on loosely-coupled, homogeneous architectures, because they offer infinite expandability and a low cost/processor ratio.

On the route to successful parallel implementation, there are three fundamental problems to be solved, these are parallelism detection, partitioning and scheduling. An automatic approach to solving these problems is presented in the thesis. The approach is based on a compile-time implementation strategy which extracts parallelism, partitions and schedules during compile-time.

Applications are written in a single-assignment language called DFDL (Digital Filter Description Language); a language designed specifically for deterministic, sampled systems. By using a single-assignment language, programs are easily translated into a graphical form (task graph) which conserves and displays parallelism. The task graph is used as an input to the partitioning and scheduling stages.

Prior to scheduling, the user is prompted for details of the target architecture. This includes the number and type of processors, input, output and inter-processor connections. These parameters are used to form a separate graph, called the processor graph. Execution profile information is added to the task graph, this enables analysis to be performed which aids scheduling.

The compile-time scheduling problem is expressed as an optimisation problem, which is shown to be NP-complete. The thesis presents an efficient approximation algorithm for the scheduling problem, which is based on a lateness heuristic. The resulting schedules are translated into parallel Occam for execution on an array of Transputers. The successes and failures of the implementation strategy are examined and commented upon. Performance results are given for several example applications written in DFDL. These examples are implemented on a range of architectures and the effects of communication, scheduling and topology are discussed.

Notation

O (*f* (n))	"Order of" for *f* (n): a quantity whose magnitude is less than some constant times *f* (n), for all large n
$\mathbf{Z}^+$	Set of positive integers
$\mathbf{Z}^-$	Set of negative integers
G	Acyclic task graph, (**T**, **C**, B, E, **A**)
T	Set of nodes representing processor tasks, $\{T_1, T_2, ..., \text{etc.}\}$
|**T**|	Set cardinality
C	Set of nodes representing communication tasks, $\{C_1, C_2, ..., \text{etc.}\}$
A	Set of arcs, partial order on **T** and **C**
B	Beginning node of graph **G**
E	Ending node of graph **G**
LST( )	Latest start time of a task
EST( )	Earliest start time of a task
FLT( )	Float time of a task
w^*	Critical path length
::=	Production
Z	Delay operator
e_i	Cost (execution duration) of task T_i
c_j	Cost (communication duration) of task C_j
$\lceil \_ \rceil$	Ceiling function
$\lfloor \_ \rfloor$	Floor function

Notation

R	Set relation
V	Processor graph, **(P, I, O, L)**
P	Set of processor nodes, {P1, P2, ..., etc.}
I	Set of input port nodes, {I1, I2, ..., etc.}
O	Set of output port nodes, {O1, O2, ..., etc.}
L	Set of arcs connecting nodes in **V**
D( )	Degree of a node
N	Set of nets, {N_1, N_2, ..., etc.}
N_k	A net: disjoint set of processor nodes, input ports, output ports and connecting arcs
S	Set of schedules, {S_1, S_2, ..., etc.}
S_k	A schedule: disjoint set of sequentially ordered tasks from **T** or **C**
P_j	Processor node representing a processing resource
O_j	Output node representing an output port
I_j	Input node representing an input port
SUC( )	The set of successors to a task
IMSUC( )	The set of immediate successors to a task
PRED( )	The set of predecessors to a task
IMPRED( )	The set of immediate predecessors to a task
COST(T_i, ..., T_k)	Accumulated cost of traversing a path T_i, ..., T_k
MAX{ }	Maximum value of all elements within { }
MIN{ }	Minimum value of all elements within { }

Notation

SUM{ }	Sum of values within { }
PROD{ }	Product of values within { }
n	Number of processor executable tasks
m	Number of processors
$s_i(k)$	Start of execution time for T_i in a schedule S_k
$f_i(k)$	Finish of execution time for T_i in a schedule S_k
$l_i(k)$	Lateness of execution for T_i in a schedule S_k
$w(k)$	Schedule duration of schedule S_k
R	Set of processor executable tasks that are available for scheduling
$w^{\wedge}$	Overall schedule deadline for a scheduling problem
b	Branching factor
p	Number of unique acyclic paths between a processor and all other processors in a network
$\overline{\vert\vert}$	Scheduling problem
$\overline{\vert\vert}^*$	Scheduling problem when partial order **A** on **T** is strictly sequential
h	Scheduling level, $0 < h <= n$
Y_h	Solution node for the $h^{\underline{th}}$ level
L_i	Lower stopping criterion for a possible solution node
U_i	Upper stopping criterion for a possible solution node
TC	Time complexity

Notation

BT	Backtracking
HC	Hill-climbing
BF	Best-first
$\bar{x}$	Mean value of x
x^y	x to the power y
x!	x factorial
$\log_b x$	Logarithm, base b, of x
ln x	Natural logarithm of x
exp x	Exponential of x; e^x
x_{min}	Minimum value of x
x_{max}	Maximum value of x
Q	Topology factor
> >	Much greater than
< <	Much less than
Inf.	Infinity
(x, y)	An arc, from element x to element y
PE	Processing element
IP	Instruction processor
DM	Data memory
IM	Instruction memory
$d_i(k)$	Shortest communication path from P_s to P_k for task T_i

Chapter 1. Introduction

Digital signal processing (DSP) is a branch of signal processing that uses digital systems to operate on signals. This form of processing has many attractions; data can be manipulated in time, a wide range of arithmetic operations and algorithmic complexity is possible and precision is arbitrarily high. One disadvantage, however, is that digital techniques are inherently slower than their analogue counterparts.

The wide scale use of electrical digital systems began in the mid 1940's. Since that time, there have been significant developments. These developments have led to an upward trend in complexity, precision, throughput and miniaturisation. Major technological advances include the replacement of mechanical relays by thermionic valves, the valves subsequent replacement by discrete transistors, and the introduction of high density integration techniques. These developments have meant that switching speeds have fallen drastically and processors that previously occupied a room and required vast amounts of power, will now fit in a pocket and run off a small cell.

Manufactures of integrated circuits, using current fabrication techniques, can produce integrated circuits that consist of millions of transistors. The trend for increased integration, as a way to reduce switching speeds, may continue for some time. However, there are physical limitations related to the molecular structure of semiconductor materials that place a ceiling on such miniaturisation. For the future, there are many directions for advancement, these are mainly motivated by a desire for increased performance; greater data throughput, more accuracy, smaller latency and/or increased complexity. A technological answer to the question, "How do we increase performance?" is to continue to reduce switching speeds. This may be achieved by changing to a different semiconductor material, or by changing the computing medium altogether, e.g. acousto-optics.

An alternative direction, which neither precludes or necessarily requires technological advancement, is to develop new architectures and programming methods. This route aims to exploit the concurrency within processes, by simultaneously processing data independent operations. Allen (1985), in his comprehensive review of computer architectures for digital signal processing (DSP), discusses DSP evolution and highlights many current trends. Both he, and others (e.g. Gaudiot, 1987; Gajski and Pier, 1985), have identified three architecturally related routes to increase processing performance:

- The first route relies on the computer architect to exploit new and existing processing architectures, in order to increase the processing throughput.

- The second, calls for the design of new parallel algorithms, using languages which support the concept of parallelism, like CSP (Hoare, 1978) and Occam (May, 1987).

- Finally, the third route focuses on the design of advanced compilers to extract parallelism automatically.

The underlying aim of these points is to achieve an increased processing throughput by parallel processing. To achieve this aim, Backus (1978) suggests that processing architectures must consist of multiple processing resources and that we should move away from the conventional von Neumann model of computing. With this, and the previous points as motivation, the main objective of this dissertation is to develop an efficient implementation strategy, that maps an algorithm description onto a parallel architecture with the aim of maximising performance.

1.1 Processor architectures

Processor architectures are described according to Skillicorn's taxonomy (Skillicorn, 1988), a taxonomy that extends Flynn's (1972), especially when describing parallel architectures. It is a two level hierarchy in which the upper level classifies architectures by the number of instruction processors, the number of processing elements and the interconnections between them. A lower level, though not discussed here, distinguishes variants even more precisely; this level is based on a state machine view of processors.

1.1.1 Von Neumann machine

The von Neumann abstract machine consists of a single instruction processor (IP), a single processing element (PE) and two memory hierarchies; instruction memory (IM) and data memory (DM). These functional units are arranged as shown in Figure 1.1.

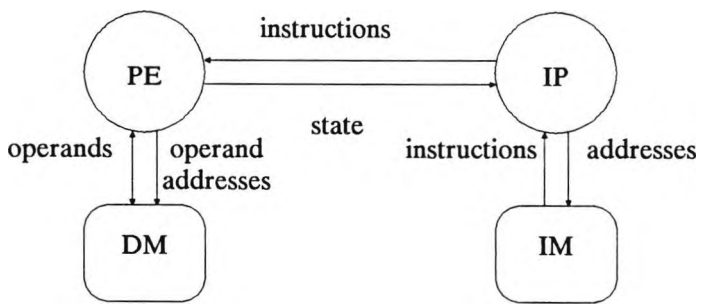


Figure 1.1 Von Neumann architecture

Other computational models are motivated by a desire for increased performance, above that accorded to the abstract von Neumann machine. Enhancements to the performance are made in one, or more, of three ways:

- Rearrange the machine's state diagram to reduce the time taken for an instruction/execute cycle (token) to circulate and complete. This is achieved by executing independent operations simultaneously, either within the instruction processor, processing element, or both, e.g. independent instruction and data busses produce a Harvard architecture. These changes are regarded as state changes and do not alter the architectural class of the machine.

- Have more than one token circulating, and hence, more than one active time step at any one time. This method of performance enhancement is called pipelining. Pipelined behaviour can be described without adding more functional units, because it is regarded as a state level change. Instead, pipelined units are distinguished from simple units by labelling.

- Replicate functional units to permit parallel activity. This form of enhancement changes the architectural class of the machine and is often the most beneficial route to improving performance.

The final way to improve performance is discussed further by showing several of the commonest parallel architectures, beginning with the array machine.

1.1.2 Array machines

The simplest way for replication of functional units is found in the array processor. Typically, array processors consist of a single instruction processor, that broadcasts instructions to a number of processing elements, which have access to data memory. Instructions are included for data to be exchanged between processing elements, either directly, or indirectly.

Array processors use a 1 to N switch (abstract term for connectivity) to broadcast from the single instruction processor to the N processing elements. Two different sub-families are distinguished, based on the relative arrangement of data memory, processing elements and the necessary switching that facilitates interconnection between functional units. The first is shown in Figure 1.2. This kind of array processor has a processing element to data memory connection of N to N and a processing element to processing element connection of N by N; every processing element can communicate with every other processing element. This architecture is similar to that used in the DAP (Reddaway, 1973) and the Connection Machine (Hillis, 1985).

The second kind of array processor, Figure 1.3, has a processing element to data memory connection of N by N. In this case there is no direct connection between processing elements, hence, inter processor communication is achieved via shared memory. An example of this architecture is found in the Burrough's Scientific Processor (Kuck and Stokes, 1982).

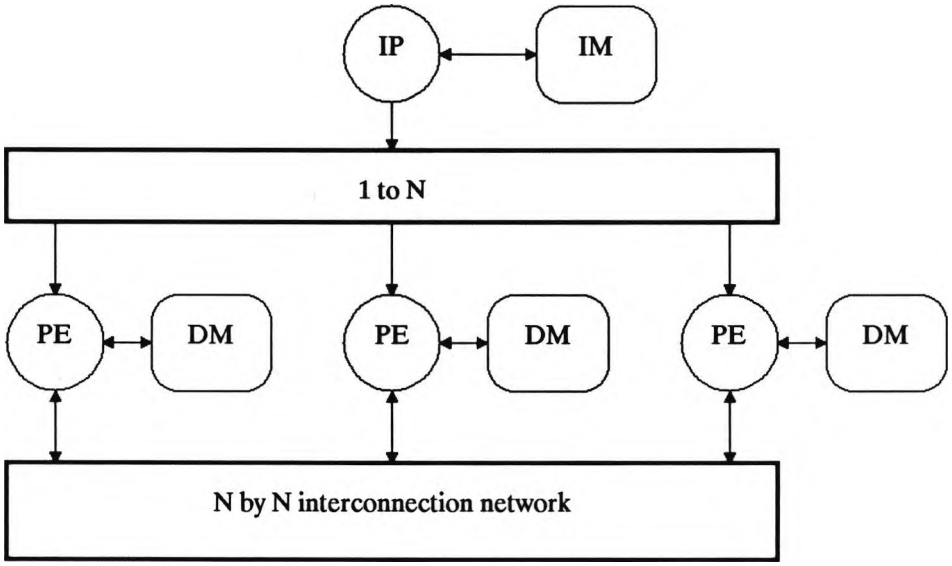


Figure 1.2 Array architecture

Array machines, such as the DAP, Connection Machine and Burrough's Scientific Processor, are used to process general purpose, regularly structured computations. For example, matrix-matrix or vector-matrix operations, as found in finite element analysis and simulation operations.

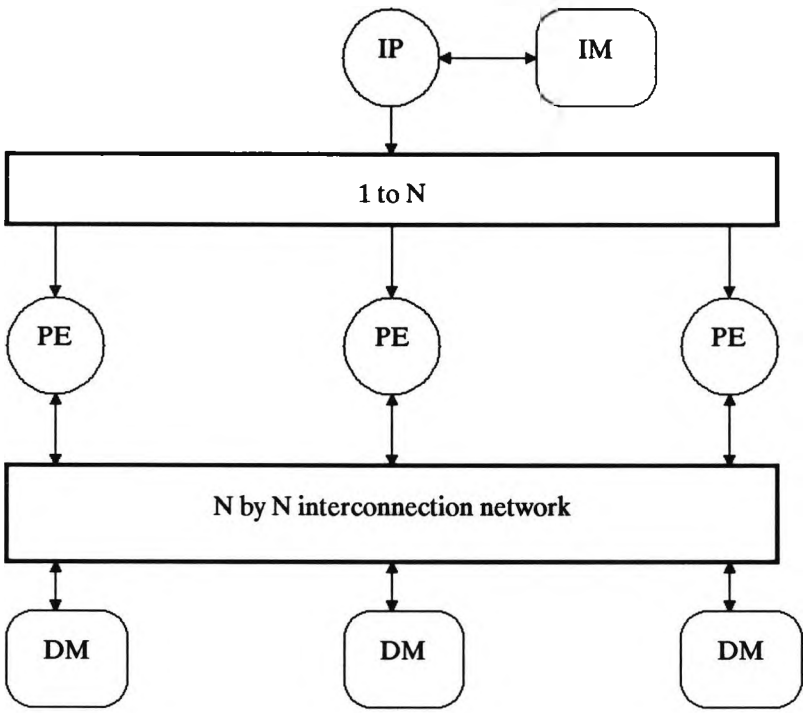


Figure 1.3 Array architecture

An alternative form of array processor is the systolic array (Kung, 1982). A systolic array is designed from regular processing elements (to reduce the design and production costs), each connected to their nearest neighbour. This principle of locality (only nearest neighbour processing elements are connected) aims to reduce communication time between processors and so increase performance. Systolic arrays are often single instruction. Consequently, the instruction processor reduces to a memoryless synchronisation unit, i.e. global clock. The operation of the array is usually determined during the manufacturing process. Systolic arrays rely on suitable algorithms which reflect the high degree of modularity and locality of the architecture. Semi-systolic arrays (More, McCabe and Urquhart, 1987) relax the principle of locality in one or more dimensions, so that a greater number of different algorithms can be implemented. This is often at the expense of performance, because of the introduction of global data distribution.

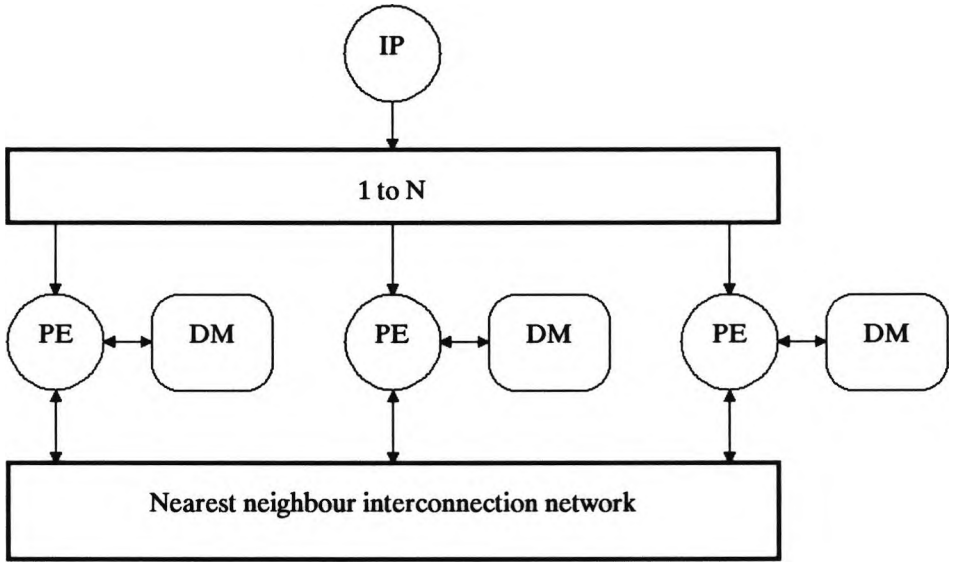


Figure 1.4 Systolic array architecture

Another single instruction array processor worthy of note is the wavefront array processor (Kung, 1984). Like the systolic array, a wavefront array conforms to the principle of locality in order to keep communication time to a minimum, and regularity in order to maintain low design/production costs. However, the wavefront array's processing elements are data synchronous and do not, therefore, require global synchronisation. Its abstract functional architecture, Figure 1.5, reveals the absence of an instruction processor (or global clock). Like the systolic array, the operation of a wavefront array is usually determined during the manufacturing process.

Both systolic and wavefront arrays are, as a rule, dedicated to a single, regularly structured function and would typically form the processing core of a high speed processor. Examples of their uses are correlators and 2D convolvers.

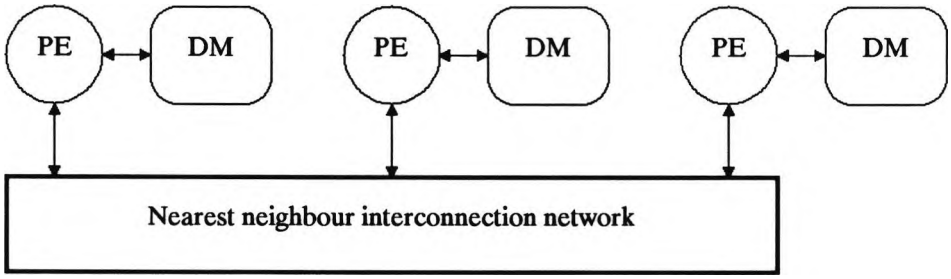


Figure 1.5 Wavefront array architecture

Array processors rely on the existence of regular parallelism; where many different data streams can be manipulated by the same operation simultaneously. Many problems do not fit this paradigm, especially when parallelism is irregularly organised. Consequently, it is natural to consider replicating the instruction processor as well as the processing element. This allows for the simultaneous execution of different instructions on different pieces of data.

1.1.3 Parallel von Neumann machines

One major class of architecture based on replicated instruction processors and replicated processing elements is the parallel von Neumann machine. This structure is aimed at providing parallel, general purpose computing. Essentially, two different architectures result from this approach; loosely coupled machines and tightly coupled machines.

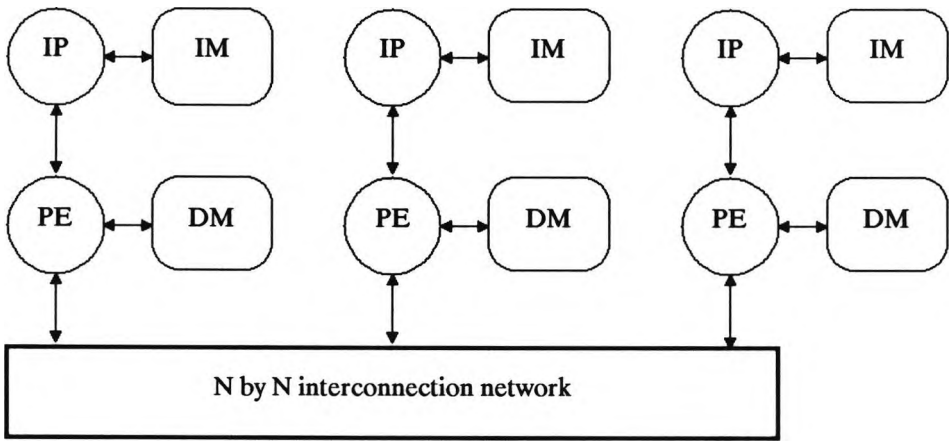


Figure 1.6 Loosely coupled architecture

Loosely coupled machines, Figure 1.6, comprise a set of processing elements, each with their own local memory, a set of instruction processors and an interconnection network. Inter processor communication takes place over the N by N interconnection network, usually by message passing. Typical examples of these machines are the CM*, Intel Hypercube, Meiko MK40, TX16 (Gaudiot, Dubois, Lee and Tohme, 1986) and Supernode (Esprit project 1085), the three latter examples are all Transputer (INMOS, 1986) based.

Tightly coupled machines differ from loosely coupled machines by the way data memory and inter processor communication is organised. Communication between processing elements is achieved via an N by N switch, which connects the data memory to the processing elements. There is no direct interconnection between processing elements, therefore, inter-processor communication is made via shared memory. Examples of tightly coupled (or shared memory) machines are the BBN Butterfly, Denelcor HEP, IBM RP3 and NYU Ultracomputer. The functional architecture of the tightly coupled machine is shown in Figure 1.7.

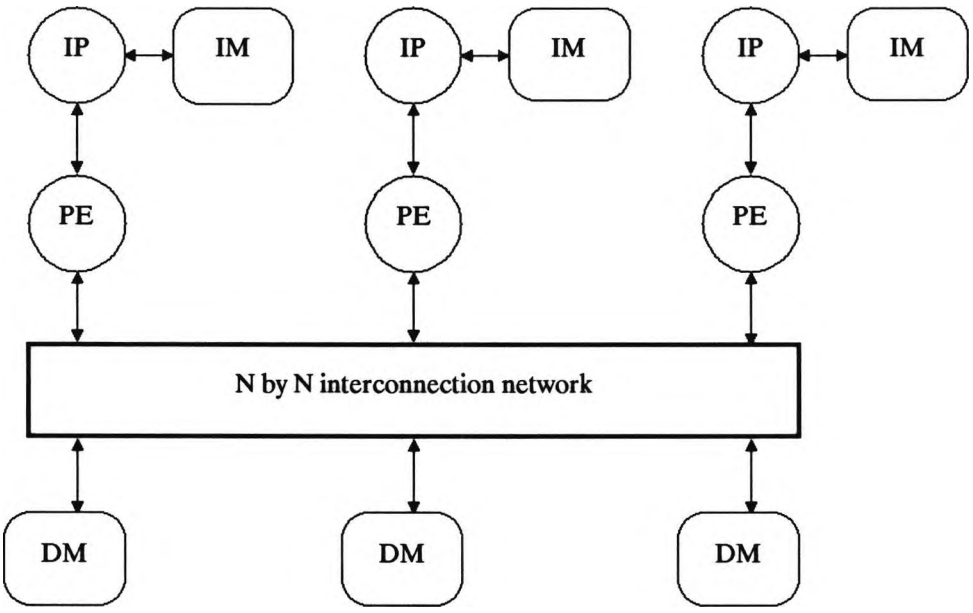


Figure 1.7 Tightly coupled architecture

A number of these example multi-processors are reviewed in (Jesshope,1987).

The von Neumann model of computation is based on a thread of instructions executed sequentially, except where order is explicitly altered. When multiple threads of control are employed, as with a parallel von Neumann machine, programmers must not only consider the ordering of instructions in a single thread, but also the different possible orderings in interacting threads. This problem often makes programming awkward and has led designers of parallel machines to examine alternative models of computation.

1.1.4 Non-von Neumann machines

Alternative, non von-Neumann models of computation are characterised by an absence, in the program description, of an explicit ordering of execution. The only ordering remaining, is that implied by data dependencies. This allows for many different evaluation orders to be considered for execution. Evaluation is made at compile time, or sectioned between compile time and run time, and aims to select the ordering (or schedule) which promises the greatest performance. Models of computation with this property are programmed in non-procedural programming languages.

1.1.5 Dataflow machines

The dataflow model of computation represents a parallel computation as a directed graph (data dependent structure), which removes the requirement for unnecessary sequencing. A task, represented as a node in the directed graph, is only ready for processing once all its preceding dependencies have been executed. Consequently, at any one time, there may be many tasks available for processing.

Most dataflow machines are based on a ring structure, consisting of an unmatched token store (where data values wait until a complete set of operands is present), memory containing the operators and a set of processing elements that execute the operators. Result values, from the processing elements, flow around the ring structure and are matched to tokens within the unmatched token store. The abstract data flow machine can take one of two architectural forms, namely static or dynamic.

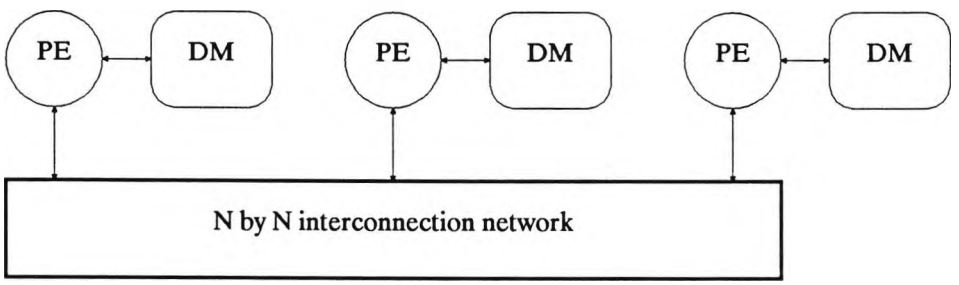


Figure 1.8 Static dataflow architecture

In a static dataflow machine, each processing element has its own memory, and data values needed by other processing elements flow across the inter processing element switch. The diagram of the functional architecture, Figure 1.8, differs from a parallel von Neumann architecture, in that it has neither an instruction processor nor an instruction memory, because the directed graph plays the role of both instruction (in its structure) and data (in its content). The functional diagram is similar to that of a Wavefront Array, which is not surprising since both are data driven machines. An example of a static dataflow machine is the MIT Static Dataflow Machine (Dennis and Misunas, 1975).

In a dynamic dataflow machine, Figure 1.9, all data memory is equally accessible to all processing elements, as in tightly coupled machines. As with a static dataflow machine, a dynamic machine has neither an instruction processor nor an instruction memory, because the directed graph plays the role of both instruction and data. Examples are the MIT Dynamic Dataflow Machine (Arvind and Kathail, 1981) and the Manchester Dynamic Dataflow Machine (Gurd and Watson, 1980). Additional information on dataflow machines and languages has been published in a special issue of Computer (IEEE, 1982).

1.1.6 Graph reduction machines

In reduction machines (Chambers, Duce and Jones, 1984), expressions are evaluated by successively reducing all component sub-expressions until only simple data values remain. Evaluation is achieved by expression substitution; for each expression that is not a simple data value, a set of rules define what is substituted when that expression occurs. The machine works by matching the current expression being processed with its corresponding rule. Once matched, the expression is substituted according to the rule. The process of expression substitution continues until all sub-expressions are processed and only simple data values remain. These represent the value of the expression. All independent sub-expressions can be matched and substituted concurrently, thus there is the potential for a high degree of parallelism.

The diagram of the functional architecture for a reduction machine, Figure 1.9, is functionally identical to that of the dynamic dataflow machine. Graph reduction machines are the focus of current research interest especially in the UK, examples of these are Alice (Darlington and Reeve, 1981) and Flagship (Watson, 1988).

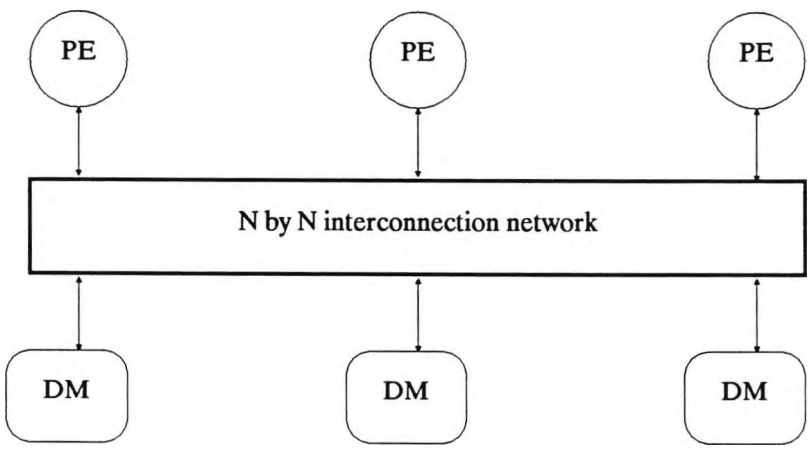


Figure 1.9 Dynamic dataflow/Reduction architecture

1.2 Interconnection Networks

All communicating parallel processing machines employ some form of interconnection network, over which they synchronise or pass data. Such networks are worthy of note, since the interconnection network often limits the performance of a machine.

1.2.1 Bus interconnection

The least complex form of dynamic interconnection network is the shared bus, Figure 1.10, which allows one processor at a time to transmit to one or more devices on the bus. Contention problems arise whenever more than one processor attempts to transmit simultaneously. Consequently, bus arbitration is necessary, this often results in considerable time being spent waiting for the bus to clear before a processing element can transmit.

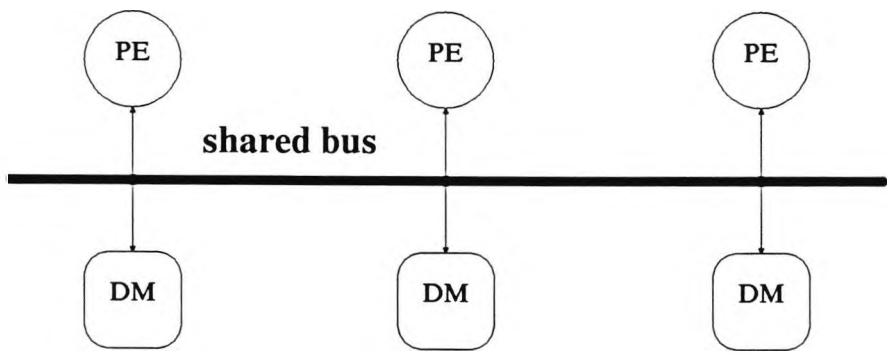


Figure 1.10 Bus interconnection

1.2.2 Crossbar interconnection

At the other extreme, the crossbar switch, Figure 1.11, supports all possible distinct connections between devices. The complexity, however, of an N by N crossbar switch is $\mathbf{O}(N^2)$. For machines comprising large numbers of processors, the complexity, and hence the cost of a crossbar switch may be prohibitive.

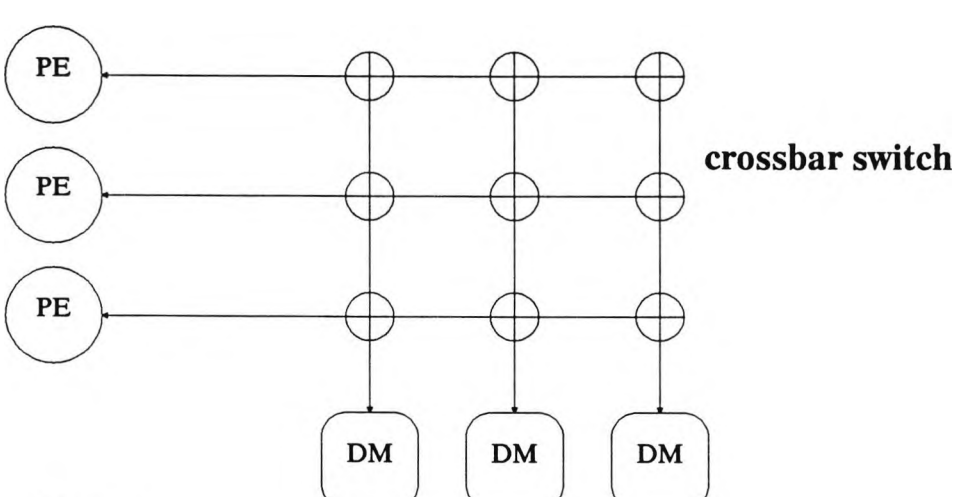


Figure 1.11 Full conectivity

1.2.3 Omega interconnection

A compromise between using a shared bus or using a crossbar interconnection network is to either use multiple busses, or use multistage switches like the omega or delta networks. The omega network, shown in Figure 1.12, is made up from 2 by 2 crossbar switches and has a complexity of $2N\log_2 N$ for an N by N network.

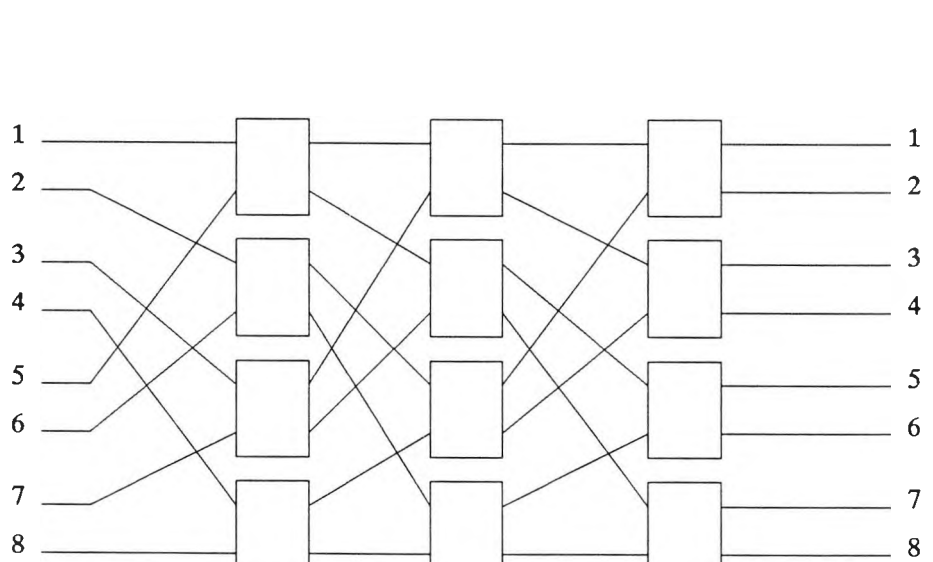
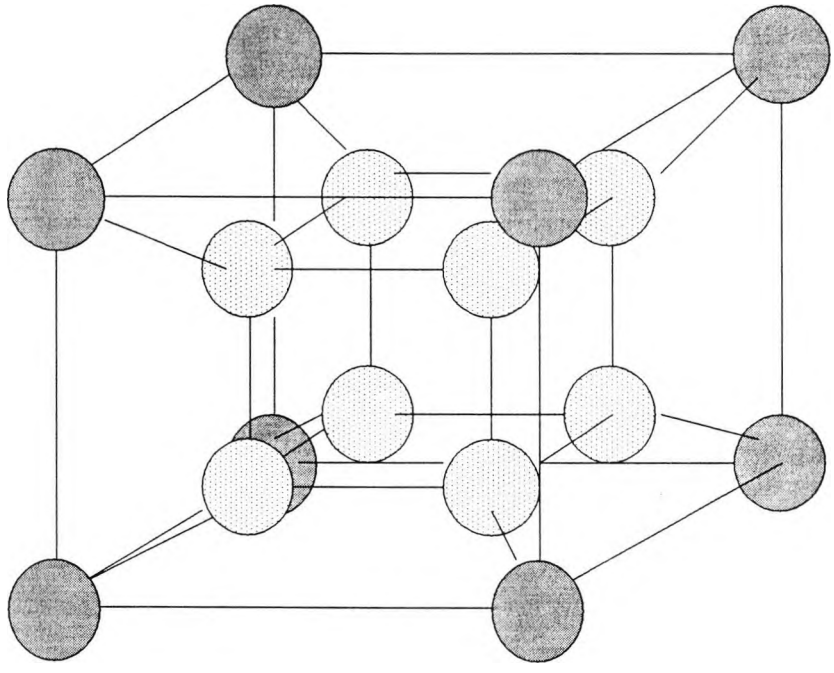


Figure 1.12 Omega network

The penalties for using a multistage network (compared to crossbar interconnection), is an increased latency because of multi-stage switching, and is a possible delay due to routing conflicts. The performance of networks using each type of interconnection method has been studied extensively. Bhuyan, Yang and Agrawal (1989) present a comprehensive review on interconnection networks, including relative figures of merit for different network types.

1.2.4 Static interconnection

Static networks consist of point to point connections called links. These networks are often used where complete connectivity is not essential, as in the case for loosely coupled machines, systolic arrays and wavefront arrays. Networks are classified in terms of their degree and their diameter; the degree is the number of links per processing element, and the diameter is the maximum number of links a message has to travel between any source and destination along the shortest path. Networks that have a lower degree for each processing element give rise to a higher diameter, which means a greater delay in average communication time. Increasing the degree of a processing element reduces the diameter of the network, but increases its cost.



Example 3-D Hypercube has N = 16 processors,
degree and diameter of 4. There are 4 distinct
paths between any source and destination processor.

Figure 1.13 3-D Hypercube

The majority of research on static networks has been carried out on networks that have a regular topology (Reed and Grunwald, 1989), these include linear networks, chordal and simple rings, 3-D torus, 3-D hypercube and tree networks. A linear network and a completely connected network are two examples that represent extremes in their degree and diameter. For example, an N processor linear network has a degree of 2 and diameter of N, where as an N processor completely connected network has a degree of N-1 and a diameter of 1.

1.3 Programming languages

Just as there are different classes of processor architecture, there are also different classes of programming language. Often, there are strong relationships between classes of architecture and classes of language, so much so, that classes of language are sometimes referred to by a hardware analogy, e.g. multi-processor language. This close relationship is not surprising, since both the language and its related architecture class are generally conceived from the same computational model.

It is sometimes useful to distinguish between different types of parallelism. For example, "regular parallelism" and "irregular parallelism", and also "fine grain parallelism" and "coarse grain parallelism". Regular parallelism exists whenever the same task is performed many times over, usually on disjoint data. Whereas irregular parallelism exists whenever different tasks are performed and their data is independent. The size of concurrent sections of code (tasks or processes) relative to the smallest atomic operation defines the granularity of the parallelism. Fine grain suggests there are many small concurrent tasks, whereas coarse grain, suggests there are a few large concurrent tasks.

1.3.1 Array and vector programming

The notion that an existing sequential language can be used to program a parallel machine is appealing and indeed, the proliferation of Fortran programs have motivated many researchers towards investigating this subject. There have been two separate avenues of approach, the first is to detect and extract parallelism in the compiler, while the second relies on parallel extensions to existing sequential languages.

Parallel extraction at the compilation stage is perhaps the most attractive, because existing programs can be used without the cost of re-development (Padua and Wolfe, 1986). In practice, however, to gain any substantial benefit the user must restructure the program to remove ambiguities that the compiler cannot resolve. Languages like Fortran and Pascal are inherently difficult to "parallelise", because they exhibit *side effects* due to the explicit use of storage locations which impedes data flow analysis.

variables and careful control over the scope of variables make this side effect preventable. A less obvious side effect arises from when an array or record is indexed by one or more variables, whose value is not known by the compiler, as is the case when the value is derived from an input. Concurrent execution of these array elements may cause unknown conflicts, hence array elements are executed sequentially in their original order, with the subsequent loss of potential concurrency. However, the worst problem is that of aliasing via the use of unbounded arrays or arithmetic operations on pointers. No amount of compile-time analysis can help unravel devious or undisciplined use of such language "features".

Despite these problems, parallelism may be extracted from repeated regular sections of code, i.e. DO loops. Although, this inevitably means the only parallelism that is generally detectable, is that between regular sections of code (i.e. regular parallelism). For this reason, many paralellising compilers have been written for array processors, where processing is performed in a lock-step or overlapped fashion. For example, compilers have been written to run Fortran on the Cray-1 and on the Illiac IV (Millstein, 1973).

Existing sequential languages, like Fortran, have been extended so the parallelism of a specific machine can be exploited. Often these extensions directly reflect the architecture of the machine which, once modified, renders programs un-portable. However, the implementation problems, which are a major challenge are considerably simplified. Once again though, the only parallelism that is readily exploited is regular parallelism, hence extensions to these languages are suited to array processors. Examples of extended Fortran languages are CFD (Stevens, 1975) for the Illiac IV, DAP Fortran for the ICL DAP and 3L Fortran for the INMOS Transputer. Likewise, several versions of "parallel" C have been written for the INMOS Transputer (INMOS, 1986).

1.3.2 Multi-processor programming

The techniques for programming distributed and multi-processor systems are similar to those used in operating systems for controlling concurrent access to shared resources. There are many specialised languages that have been written for concurrent programming and these differ considerably from one another, however, they all have the following three features in common (Andrews and Schneider, 1983);

(i) the ability to express concurrent execution,

(ii) process synchronisation and

(iii) inter-process communication.

There are four basic mechanisms which have been used for achieving concurrent execution. First and simplest of these is the *co-routine* which has been included in languages such as Modula-2 (Writh, 1978). Secondly, the *fork* and *join* notation which is used in the UNIX operating system and can be found in the Mesa language (Mitchell, Maybury and Sweet, 1979). Thirdly, the *cobegin*, or *parbegin* (Dijkstra, 1968) which is employed in CSP (Hoare, 1978) and more recently in Occam (May, 1983). Finally, explicit *process declarations* are found in Concurrent Pascal (Brinch-Hansen, 1975), Modula (Writh, 1977) and Pascal-M (Abramsky and Bornat, 1982).

Synchronisation and communication between processes can be achieved either by reading and writing to shared data or by sending and receiving messages. In general it is difficult to separate communication from synchronisation, since synchronisation requires a flow of information from one synchronising process to an other. Similarly, communication requires some ordering of events if processes are to communicate with each other sensibly.

When communication is based upon the use of shared data, then there are two types of synchronisation (Andrews and Schneider, 1983); *mutual exclusion* and *condition synchronisation*. Mutual exclusion allows an executing process to be treated as an indivisible sequence of operations, this prevents interference from other processes. The second form of synchronisation is condition synchronisation, which co-ordinates the execution of concurrent processes by controlling when a process waits and when it commences execution. Various methods of achieving synchronisation and communication using shared data have been applied, examples of these are *semaphores* (Dijkstra, 1968), *conditional critical regions* (Brinch-Hansen, 1972) and *monitors* (Hoare, 1974). The use of shared data is a centralised approach to controlling concurrency, and is therefore closely related to tightly coupled processor architectures.

An alternative method to shared data is message passing. Message passing requires that processes are named so messages are passed between identified processes. Inter-process communication takes place within some medium, which is called a *channel* in Occam and a *mailbox* in Pascal-M. Different communication mechanisms vary in the way communication interacts with the activity of a sending process. For example, in a "no-wait send" mechanism the process continues as soon as data is sent, this implies the receiving process has an unbounded buffer in which to hold a queue of messages. Whereas "synchronised send" waits until the message has been received before continuing. The latter method is employed in CSP, Pascal-M and Occam. Message passing is a de-centralised approach to controlling concurrency, and is therefore closely related to loosely coupled processor architectures.

1.3.3 Non-von Neumann programming

The previous two sections have discussed control flow languages, whose common characteristic is to execute the program in the order it is textually composed. Backus (1978) has argued that this style of programming language (influenced by the von Neumann model of computation) can make programming unnecessarily difficult. An alternative execution strategy is to execute operations as and when their input data becomes available, i.e. data driven. Hence, in data driven systems, the order in which programs are written becomes less important, since it does not determine the order of execution.

Dataflow languages are characterised by an absence of concurrent control constructs, as found in multi-processor languages. In place of these explicit constructs are rules that govern assignment; single-assignment or zero-assignment.

Single-assignment languages (SALs) have the appearance of conventional languages, in that they incorporate assignment statements and include typical control flow constructs such as conditional statements and loops. However, they have no concept of sequential execution and no direct control constructs like GOTO. In order to prevent ambiguities that might arise from re-assigning variables, the language only permits a variable to be assigned once throughout the program (Chamberlin, 1971). This limitation significantly alters the nature of the assignment operator, changing it from a dynamic destructive operation to one that statically associates a name to a data value. Special provision is made for variables within iterative expressions, such as SISALs "old" operator. SALs tend to use data structures, such as arrays and streams, that are readily implemented in dataflow graphs. Examples of SALs are SISAL (M^{c}Graw, Skedzielewski, Allan, Grit, Oldehoeft, Glauert, Dobes and Hohensee, 1983) and VAL (Ackerman and Dennis, 1979). Most SALs are designed for generalised programming, however, languages have been written specifically for signal processing applications. SALs are a natural environment for representing signal processing algorithms because of the strong correspondence between signal flow and dataflow. Languages for signal processing include SIGNAL (Guernic, Benveniste, Bournai and Gautier, 1986), SDF (Lee and Messerschmitt, 1987) and PSPL (Thaler, Loeffler and Moschytz, 1987).

Zero-assignment languages are usually known as functional languages, or applicative languages and are based on the mathematics of lambda calculus, or recursion equations. The language has no concept of storage state or assignment (Backus, 1978). Typically, a program consists of an un-ordered set of equations that characterise functions and values; functions are characterised by the use of recursion, other functions and values, while values are characterised by functions of other values. In many ways functional languages are identical to single-assignment languages in that single-assignment and zero-assignment definitions result in non-destructive association. Also, both language types are free from side effects and GOTOs. The functional language SASL (Turner, 1976) has been applied to dataflow machines (Richmond, 1982) with success, however, the efficiency of such implementations are in doubt. Much of the system and application software for the ALICE graph reduction machine is written in a functional language called

HOPE (Burstall, MacQueen and Sandella, 1980). Other examples of functional languages are LISP (M^{c}Carthy, 1960), ML (Gordon, Milner and Wandsworth, 1977) and FP (Backus, 1978).

1.4 Implementation strategies

Unlike sequential systems, parallel systems require the division of a program into separate parts and each part assigned to execute on a processor. The dividing operation is called partitioning, which is defined as an operation that creates a finite number of mutually disjoint tasks, whose union is the program. Assignment of tasks to processors is carried out in both a spatial and a temporal sense, since a parallel architecture's capacity to process is a function of both the number of processors and time. This form of assignment is known as scheduling and generally, the number of tasks far exceeds the number of processors. Consequently, a processor is treated as a shared resource and it is the purpose of the scheduler to co-ordinate task-processor assignment to avoid conflict between tasks. Both partitioning and scheduling are regarded as *implementation operations*.

	Design-time	Compile-time	Run-time
1	parallelism, partition, schedule		
2	parallelism, partition	schedule	
3	parallelism, partition		schedule
4	parallelism	partition, schedule	
5	parallelism	partition	schedule
6	parallelism		partition, schedule
7		parallelism, partition, schedule	
8		parallelism, partition	schedule
9		parallelism	partition, schedule
10			parallelism, partition, schedule

Table 1.1 Implementation strategies

The success of an implementation relies heavily on a program's parallelism, how well it is partitioned and how well it is subsequently scheduled onto a parallel architecture. One important influence on parallelism, partitioning and scheduling, which subsequently affects implementation, is the stage at which such operations are completed. There are three well defined stages between program conception and execution, which are design-time, compile-time and run-time. The range of valid implementation strategies are illustrated by Table 1.1.

The different strategies have a great influence on the methods chosen for partitioning and scheduling and also on the type of programming language used. In the following section different programming language types are discussed and categorised according to their implementation strategy.

Parallel programming languages can be categorised according to their inclusion (or exclusion) of implementation constructs. Categorising languages in this way is useful since it reveals which implementation strategy a language is capable of taking part in.

One convenient way to categorise different parallel programming languages is by the absence, or presence of explicit constructs for parallelism, partitioning and scheduling. For example, parallelism becomes explicit when parallel constructs are defined which distinguish between areas of sequential and parallel execution (e.g., *fork, join, cobegin, parbegin* etc.). When parallelism is unspecified, it is necessary to extract parallelism via a program's dependency graph, usually by automatic means. A programming language which explicitly defines parallelism may also partition explicitly. Explicit partitions group executable code into processes or tasks. Finally, a programming language which partitions explicitly may also schedule explicitly. Hence, the programmer determines which process executes on which processor. The following four categories divide programming languages according to their use of implementation constructs (Sarkar, 1989):

- (1) In the first category, parallelism, partitioning and scheduling are all implicit and therefore unspecified. These language types are suited to implementation strategies 7 through to 10 of Table 1.1. A necessary step to implementation, which is not shared by the other categories, is that automatic dependency analysis is needed to identify parallelism. This is a major obstacle for some types of language.

  Conventional, sequential languages (e.g. Fortran, Pascal), have computations that are based on complex, sequential state transitions (Backus, 1978). Such languages require careful dependence analysis to reveal potential parallelism. This analysis is made difficult, as previously mentioned, because procedural languages exhibit multiple assignments, side-effects and aliasing. These difficulties may restrict the identification of parallelism and produce inefficient results. Nevertheless, there is a growing interest in implementing existing sequential programs on parallel processors, because of the large capital investment many companies have in existing sequential software.

Single-assignment and functional languages (e.g. SISAL, VAL, HOPE, etc.) are free from side-effects, multiple assignment and aliasing, consequently dependency analysis is relatively straight forward. This is a result of neither language type being tied to the von Neumann model of computation. Typically, there is scope for employing a high degree of parallelism within these language types. The absence of explicit parallelism, explicit partitioning and explicit scheduling, makes such languages portable. The attraction of portability being, that as multi-processor designs advance, programs may be implemented without undue modification.

- (2) This category contains those programming languages which exhibit explicit parallelism, while partitioning and scheduling remain unspecified. These language types are suited to implementation strategies 4 through to 6 of Table 1.1. The absence of a process, or task structure mean these languages avoid explicit partitioning. They include parallel programming constructs, such as *doall*, *cobegin* and *coend* and usually synchronise using semaphores or monitors (i.e. suited to tightly coupled architectures). Many of these parallel languages have been developed from existing sequential programming languages, for example, DAP Fortran and IBM Parallel Fortran (IBM, 1988).

- (3) Programming languages in the third category explicitly define parallelism and partitioning, while scheduling remains unspecified. These language types are suited to implementation strategies 2 and 3 of Table 1.1. Partitioning and inter-process synchronisation is defined by the programmer, whose job it becomes to group statements into processes (or tasks). The programmer must ensure that the granularity of a parallel program is fine enough to exploit potential parallelism, while coarse enough to minimise communication overhead. This strategy has the advantage of simplifying implementation, but has the potential disadvantage that implementation is prone to poor partitioning by the programmer. Program portability is retained by automatic scheduling. Many of these languages have been developed solely for parallel processing, for example, CSP (Hoare, 1978) and Ada (Mundie and Fisher, 1986; Ledgard, 1981).

- (4) The final category is for languages which give total control to the programmer, by allowing scheduling to become explicit. These language types are suited to implementation strategy 1 of Table 1.1. The remarks concerning partitioning in (3) apply equally to languages in this category, except that program portability is lost, which may necessitate manual re-scheduling of a program when moved from one machine to another. Languages in this category tend to based on message-passing synchronisation, for example, Occam and the C implementation on the Caltech Cosmic Cube (Su, Faucette and Seitz, 1985).

Though several compilers have been designed to automatically extract parallelism from programs written in imperative languages (1), parallelism extraction is impeded by the difficulties associated with the von Neumann model of computational. Languages from category (2) result in implementations that tend to restrict parallel exploitation to regular repeated regions and so lack the general scope that is necessary for an efficient implementation. The current trend, is for a programmer to make many of the implementation decisions, as is the case in categories (3) and (4). Two of the most probable reasons for category (3) and (4) popularity, is the large capital investment in Ada by the DoD (Department of Defence, USA) and the recent introduction of affordable multi-processors, like the Transputer and Intel Hypercube.

In some cases there are advantages to partitioning and scheduling manually, however, this does tend to burden the programmer with organising *how* things are done, rather than getting right *what* is done. Consequently, programming effort tends to increase when using explicit parallel processing languages. However, the greatest drawback of categories (3) and (4) is the probability that potential parallelism will be lost because a programmer opts for a less than optimal implementation.

This thesis focuses on an automated implementation, category (1), using a non-procedural programming language. The motivation behind this is three-fold; (i) to abstract the programmer from machine oriented influences while programming, (ii) to achieve program portability and (iii) to achieve a "good" implementation. The first two aims, abstraction and portability are language characteristics, while implementation efficiency relies mostly on the compiler or run-time system. The main obstacle to using non-procedural languages is the problem of parallel extraction, partitioning and scheduling, however, recent research (Gaudiot, Dubois, Lee and Thome, 1986; Sarkar, 1989) has shown that compilers can be designed which implement non-procedural languages efficiently. In this thesis, implementation is applied to real-time DSP algorithms. Such algorithms provide a special case, which allows an extremely efficient implementation strategy (Table 1.1, no.7) to be employed. Figure 1.14 shows an outline of this strategy for a Transputer based parallel architecture, using Occam as an intermediate language.

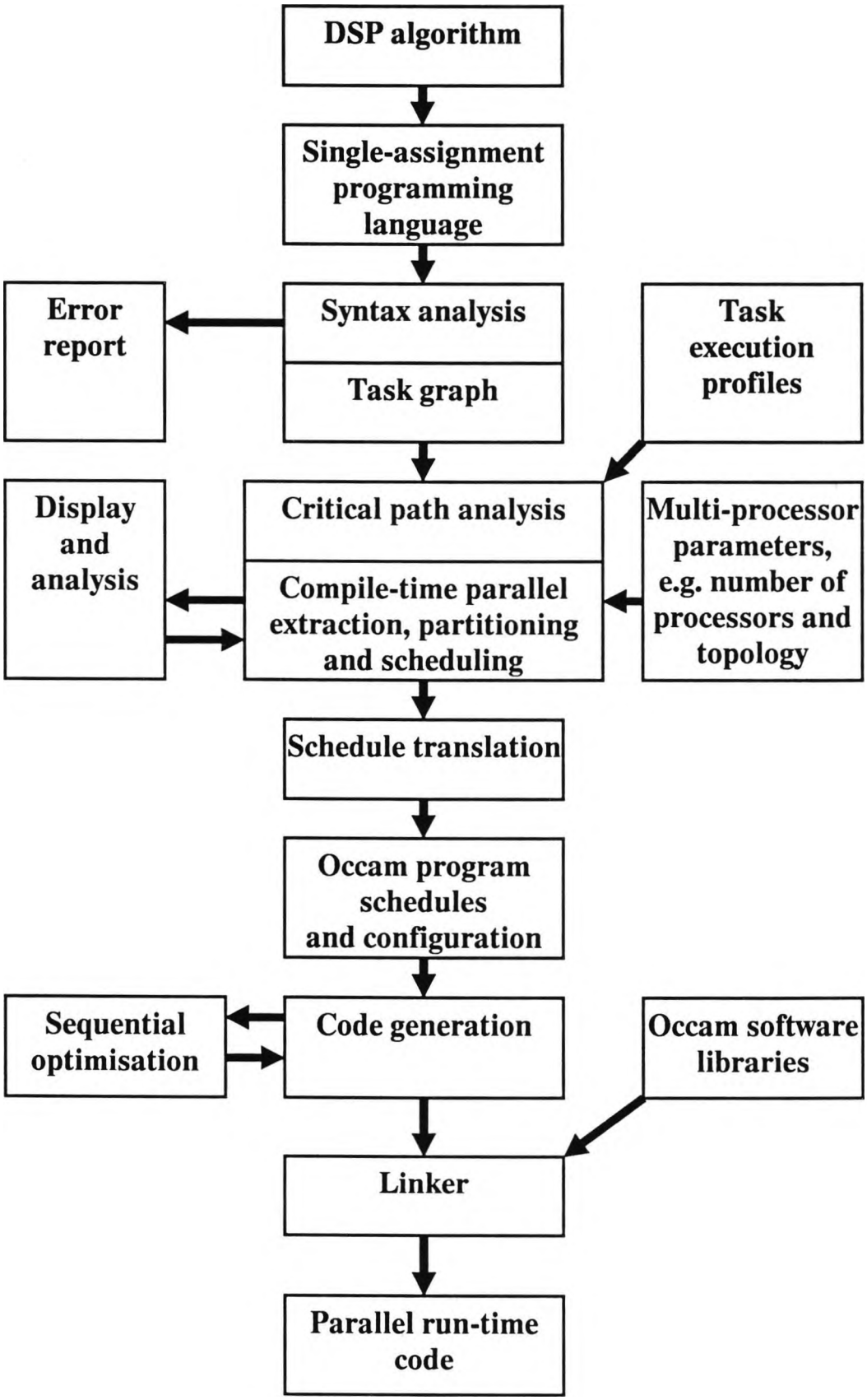


Figure 1.14 Compile-time implementation strategy

1.5 Thesis outline

The thesis is divided into nine chapters and four appendices. This chapter has introduced the subject by discussing different processor architectures, interconnection networks, programming languages and implementation strategies. The second chapter reviews the characteristics of real-time DSP algorithms and represents these algorithms in the form of a graph which preserves parallelism. Chapter 2 is considered a prerequisite to chapter 3, which discusses aspects of language design that are relevant to our application. Chapter 4 develops these ideas and describes the syntax of a single-assignment language called DFDL (Goddard, 1987; Goddard, 1989). Program structure and processor architecture are represented as two separate models, these are described in chapters 5 and 6 respectively. The models are used as inputs to compile-time scheduling, partitioning and parallel extraction. The scheduling process is described by chapter 7 and this is shown to present several difficulties, which are associated with the complexity of the problem. Experimental results are given in chapter 8 which illustrate the efficiency, or not as is the case, of this implementation strategy. The final chapter concludes on the research and offers some suggestions for further work. Figure 1.15 illustrates the relationship between the chapters and the implementation strategy.

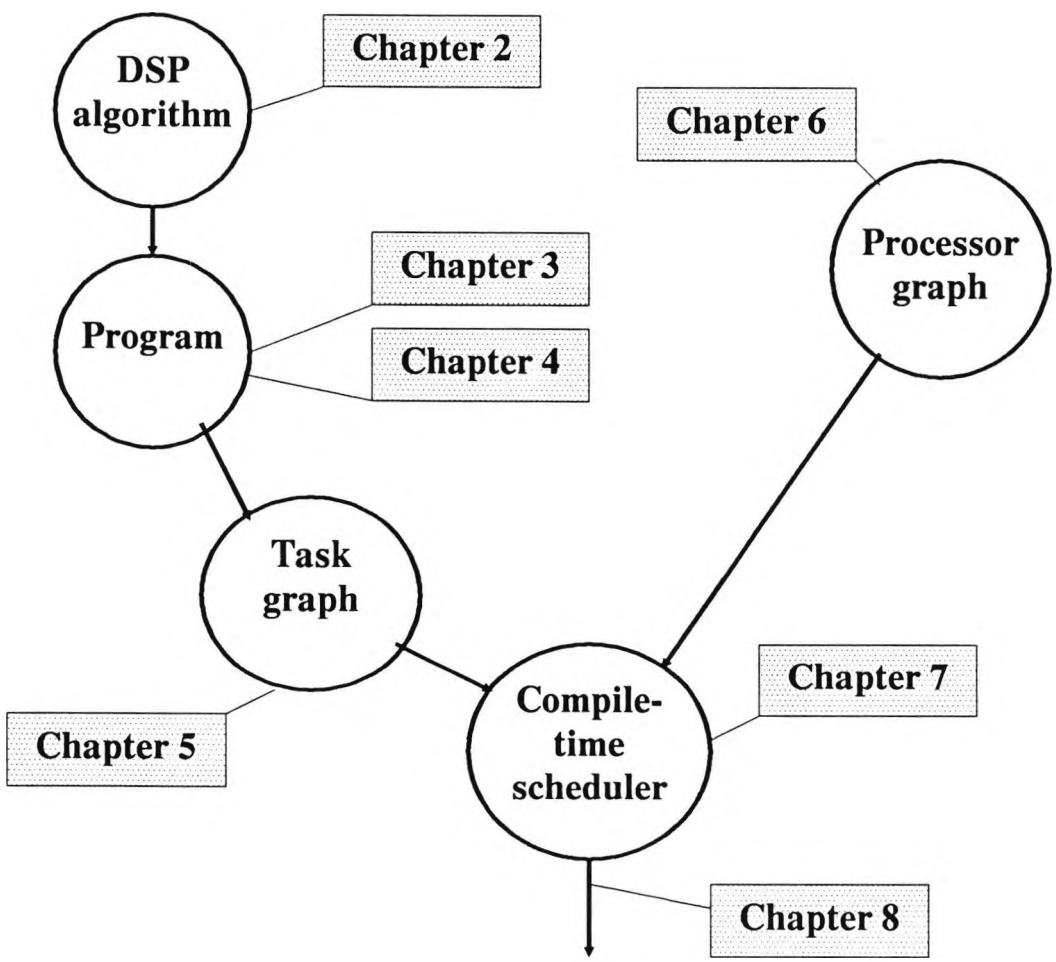


Figure 1.15 Implementation strategy by chapter

The four appendices are identified alphabetically. Appendix A describes some of the graph concepts and definitions used in the text, and Appendix B is an extended BNF description of DFDL. A compile-time user's guide is presented in appendix C, which describes compiler operation. Finally, appendix D is a programmer's guide, which describes some of the major parts of the compiler that are not covered in the main body of the thesis.

Chapter 2. Discrete algorithms and their graphs

In the first part of this chapter the relevant characteristics of discrete algorithms, their composite tasks and the structural relationship between those tasks are examined.

The second part of the chapter presents the algorithm as a data flow graph and introduces some terminology associated with graphs. The graph is seen as a complete diagrammatic representation of a DSP algorithm, where both function and structure are conveyed. Additionally, the graph is viewed as an intermediate stage between the algorithm description (i.e. a program) and the multi-processor schedules. Moreover, the graph represents the algorithm without loss of structure or function.

2.1 The characteristics of discrete algorithms

2.1.1 Discreteness

A discrete algorithm is defined here as a prescribed set of well-defined instructions which act on one or more digital signals (i.e., signals quantised in time and amplitude). The word discrete describes the algorithm as being decomposable into a finite set of individual tasks.

2.1.2 Competence-performance trade-off

The competence of an algorithm is its ability to perform a given function, whereas the performance of an algorithm is a function of execution time. When designing a discrete algorithm for execution on a digital processor, we must be aware of the practical limitations of the hardware. These limitations manifest themselves as finite word length and non-instantaneous task execution times.

Finite word length number representation causes inaccuracy and constrains the range of numbers which can be used. Arithmetic operations on finite word length numbers produce round-up and truncation errors, which are propagated and may cause large accumulated inaccuracies. The limited range of data values, often necessitates schemes which correct for, or flag out-of-range values.

Non-instantaneous execution time is a practical reality of digital systems. The greater the execution time of the tasks in any given iteration, the lower the throughput and bandwidth, and possibly the greater the latency.

Compromises may be made between the competence and performance of an algorithm. Throughput, for example, may be increased at the expense of accuracy by replacing all multiplication with shifting, to give only powers of 2 multiplication. On the other hand, accuracy may be increased by using double word length arithmetic, although this would be at the expense of throughput. These trade-offs between competence and performance heavily influence the design of a digital system and ideally, should be made in the context of the particular implementation.

2.1.3 Inputs and outputs

Hetch (1977) describes an algorithm as having zero or more inputs and one or more outputs. In the case where there are no inputs, the algorithm can only generate signals. The alternative case is where inputs are present, the algorithm can process as well as generate signals. These two models are illustrated in Figure 2.1. For a single iteration of the algorithm, the outputs are a function of both the algorithm's internal state (i.e. the value of internal data immediately prior to the commencement of the current iteration) and the current input values, where applicable.

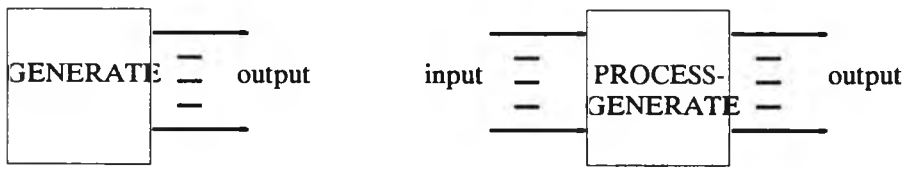


Figure 2.1 Algorithm input-output

2.1.4 Memory

An algorithm is said to have memory if it can retain data from one iteration to the next. The outputs of an algorithm without memory, do not depend on the algorithm's internal state from previous iterations, or on the inputs from previous iterations. Alternatively, the outputs of an algorithm with memory, may depend on the algorithm's internal state from a number of previous iterations, or on the inputs from a number of previous iterations; if that number is finite it is said to be of finite order (e.g. FIR filter), otherwise it is said to be of infinite order (e.g. IIR filter). Outputs which depend on an infinite number of previous iterations imply the use of feedback within the algorithm.

Realisable, stable algorithms exclude non-causal or oscillatory behaviour. Hence, outputs do not depend on the algorithm's internal state from future iterations, or on the inputs from future iterations. Also, outputs and internal values do not depend on themselves during the present iteration.

2.1.5 Complexity

The time complexity of an algorithm is the aggregate amount of time, usually expressed as a function of the number of inputs u, to process the algorithm. The limiting behaviour of the time complexity is called the asymptotic time complexity. Definitions for space complexity and computational complexity are analogous.

To express worst-case complexity we use the "big-**O** " notation (i.e. order of magnitude). Rather than present complexity as an absolute function of the number of inputs, it is presented as a function of some order. For example, the order of complexity of an algorithm that processes u inputs and has a worst-case time complexity of cu^2 (for some constant c) is expressed **O** (u^2).

2.1.6 Real-time, deterministic, synchronised systems

Young (1982) defines a real-time information processing system as one which has to respond to externally generated input stimuli within a finite and specifiable delay. In the case of sampled signals which arrive regularly, the delay is the time interval from one instance of the signal arriving at the input to the next. Ideally, this time interval is consistent throughout.

Interaction between an external input signal and an algorithm, as depicted by Figure 2.2, occurs when the algorithm reads the input. In order that no signals are lost, the interaction between input and algorithm must be synchronised.

- *Observation 2.1:* Synchronisation is maintained iff (if and only if) the maximum algorithm iteration interval is less than or equal to the minimum external input interval.

Similarly, there has to be synchronisation between the algorithm iteration interval and the external output interval (Figure 2.2). Hence from observation 2.1, we can say that there must be synchronisation between the input interval and the output interval.

- *Observation 2.2:* Synchronisation is maintained iff the maximum output interval is less than or equal to the minimum input interval.

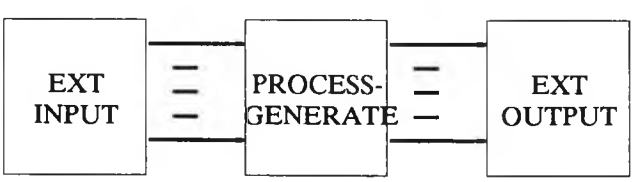


Figure 2.2 Input-process-output synchronisation

From observation 2.1, we can say an algorithm's maximum execution time (for a single iteration) is bounded by the minimum input interval. To guarantee this bound is not exceeded, an algorithm's worst-case execution time has to be specifiable prior to run- time, hence the algorithm has to be deterministic. This *a priori* condition precludes all algorithms whose worst-case execution time cannot be determined at compile-time, i.e., algorithms that have a time complexity which relies on data values, or on some random operation; these algorithms are regarded as non-deterministic and as such do not lend themselves to efficient static modelling or analysis.

2.1.7 Granularity

Algorithms can be decomposed (or partitioned) into separate, distinct tasks, by dividing the algorithm into atomic operations, that have the capacity to manipulate data in some deterministic manner. Deciding how large, or how small the tasks should be in terms of their time complexity is not straightforward. Consider, for a moment, the implementation of different sized tasks on a multi-processor, whose inter- processor communication costs are finite:

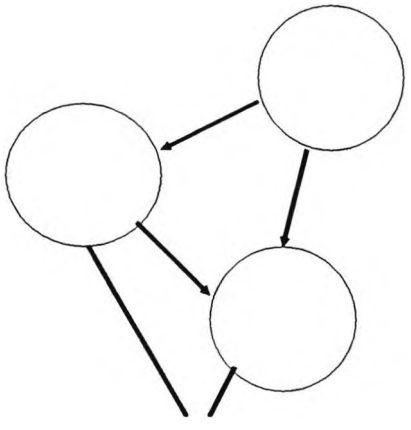


Figure 2.3 Coarse grain structure

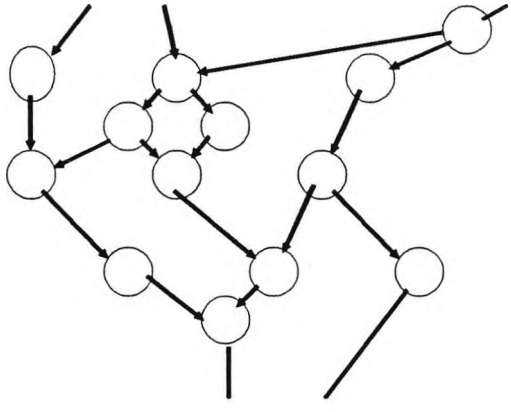


Figure 2.4 Fine grain structure

An algorithm that is divided into a few coarse grain tasks (i.e. each comprising many instructions) of low I/O bandwidth, when distributed across the processors will only need a few communications, since there are only a few tasks. This may result in a low overall communication cost. However, the opportunity to exploit any parallelism which exists within the large grain tasks is lost.

Alternatively, an algorithm that is divided into many fine grain tasks (i.e. comprising few instructions) of low I/O bandwidth, when distributed across the processors will require many communications, since there are many tasks. This may result in a high overall communication cost. However, the opportunity to exploit any parallelism within the algorithm will be high.

The partitioning dilemma is not easily resolved at this stage without the benefit of post-analysis information, which will indicate where useful parallelism exists and where it does not. One approach to partitioning, which is adopted here, is to initially opt for a medium/fine grain tasks structure. This approach does not obscure parallelism and does not preclude tasks from being "bundled" together to form composite, coarser grain tasks at a later stage, i.e. during scheduling.

2.1.8 Task primitives

Each task T_i represents a sequentially ordered list of one or more instructions, which has an in-degree (i.e. number of inputs) of zero or more operands and an out-degree (i.e. number of outputs) of one or more objects. The limit placed on the in-degree and out-degree of tasks is $|Z^+|$ (highest positive integer). Generally though, tasks have a low in/out-degree. This is important since they represent fine grain, low bandwidth operations such as addition, subtraction, etc.

The outputs of a task T_i are denoted v_{ki}, $\{0 < k <= |Z^+|\}$, and are defined in terms of the task's transfer function and operands. The operands of T_k are u_{jk} $\{0 <= j <= |Z^+|\}$, and the input and output transfer functions are $f_{ji}( )$ and $g_{ki}( )$ respectively. Each output is given as $v_{ki} = g_{ki}(f_{ji}(u_{ji}))$.

Assigned to each task T_i is a set of execution times, $[e_{kj}]_i$. This matrix of costs represents the worst-case execution times of $g_{ki}(f_{ji}(u_{ji}))$. Worst-case execution times are adopted throughout to allow for fluctuations in execution time from one instance to the next.

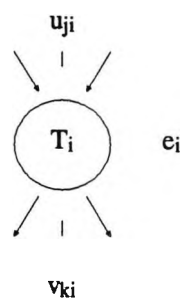


Figure 2.5 Task input, output and execution cost

Tasks are selected so wherever possible the transfer functions between all inputs and all outputs have equal time complexities. Where this is not possible, the overall worst-case value is adopted. This simplification allows us to replace the matrix of execution times by a single execution cost e_i which is assigned to the task T_i.

2.2 Data flow graphs

2.2.1 Nodes and arcs

The graph **G** = (**T**, **C**, B, E, **A**) consists of a non-empty but finite set of processor executable nodes **T** and a finite set of communication nodes **C**. The set **C** is initially empty and remains empty until scheduling, consequently it will not be considered until that time. All nodes correspond directly to a task and visa-versa, hence the terms node and task are often interchanged. The cardinality of **T**, **C** and of **A** is denoted $|\mathbf{T}|$, $|\mathbf{C}|$ and $|\mathbf{A}|$ respectively.

The set of arcs **A** joins pairs of distinct nodes. Each arc symbolises the flow of data from one task to another task. An arc (T_i, T_j) leaves the tail node T_i and enters the head node T_j. We say that T_i is an immediate predecessor of T_j, and T_j is an immediate successor of T_i. The set **IMPRED**(T_i) comprises all nodes that are immediate predecessors of T_i, and the set **IMSUC**(T_i) comprises all nodes that are immediate successors of T_i. The in-degree of node T_i is $|\mathbf{IMPRED}(T_i)|$, and the out-degree of node T_i is $|\mathbf{IMSUC}(T_i)|$.

Where there are two consecutive arcs (T_i, T_j) and (T_j, T_k), we can say T_i is an immediate predecessor of T_j, and T_j is an immediate predecessor of T_k, therefore T_i is a predecessor of T_k. By similar inductive analysis we can say that T_k is a successor of T_i. The set **PRED**(T_i) comprises all nodes that are predecessors of T_i, and the set **SUC**(T_i) comprises all nodes that are successors of T_i. The sets **IMPRED**(T_i) and **IMSUC**(T_i) are subsets of the sets **PRED**(T_i) and **SUC**(T_i) respectively.

The arcs belonging to **A** are governed by *R*, where *R* denotes a relation on **A**. This relation *R* is *transitive*, *irreflexive* and *asymmetric* (Figure 2.6). The definitions for these are as follows:

- (a) *R* is transitive iff for all nodes T_i, T_j, T_k in **T**, (T_i *R* T_j) AND (T_j *R* T_k) implies (T_i *R* T_k);
- (b) *R* is irreflexive iff for all nodes T_i, T_j in **T**, (T_i *R* T_j) AND (T_i is not equal to T_j);
- (c) *R* is asymmetric iff for all nodes T_i, T_j in **T**, (T_i *R* T_j) implies NOT(T_j *R* T_i).

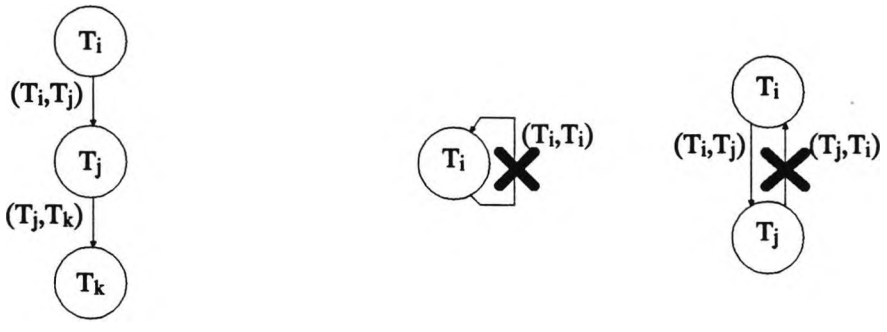


Figure 2.6 Arc relationships

From the relation *R* on **A**, three observations are made:

- *Observation 2.3:* If *R* is both transitive and irreflexive, then *R* is also *asymmetric*.

- *Observation 2.4:* If *R* is transitive, irreflexive and asymmetric, for all T_i in **T**, then T_i *can not* belong to either of the sets **PRED**(T_i) or **SUC**(T_i).

- *Observation 2.5:* If *R* is transitive, irreflexive and asymmetric, for all T_i in **T**, then the *intersection* of **PRED**(T_i) and **SUC**(T_i) is the empty set.

2.2.2 Paths and acyclic paths

Hetch (1977) defines a path as a finite sequence of one or more arcs, i.e.

$((T_1, T_2), ..., (T_{k-1}, T_k))$.

This can be written more simply (Figure 2.7) as:

$(T_1, T_2, ..., T_k)$.

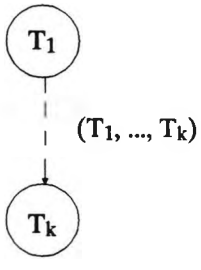


Figure 2.7 Acyclic path

A cycle is a path (T_1, ..., T_k) where $T_1 = T_k$. A path that is free from cycles is called *acyclic*. Where the relationship on the set of arcs **A** is transitive, irreflexive and asymmetric then all paths are acyclic. This statement is supported by observations 2.4 and 2.5.

The graph is characterised as being directed and acyclic, this form of graph is given the acronym DAG (directed acyclic graph).

A node T_k is said to be *reachable* from a node T_1, iff there is a path from node T_1 to node T_k, consequently the set **SUC**(T_1) contains all nodes that are reachable from T_1.

2.2.3 Initiating and terminating DAGs

As a convention all graphs are drawn with their arcs pointing downwards, so data flows from the top of the graph to the bottom as shown in Figure 2.8. For practical convenience two "dummy" nodes are added to the graph, one at the top and one at the bottom, their purpose is to initiate and terminate the graph respectively. Both exhibit zero execution cost.

The top dummy node is denoted B (i.e. begin). This node has the properties that **PRED**(B) is the empty set, therefore B has an in-degree of zero, and **SUC**(B) includes all nodes in **T**, hence all nodes in **T** are reachable from B.

The bottom dummy node is denoted E (i.e. end). This node has the properties that **SUC**(E) is the empty set, therefore E has an out-degree of zero, and **PRED**(B) includes all nodes in **T**, hence E is said to be reachable from all nodes in **T**.

2.2.4 Dependence and independence

The set of arcs **A** places a partial order on **T**. The reality of the partial ordering is that execution order of tasks (corresponding to their nodes) is restricted. The restriction on execution order is due to the data dependency which one task has on another. For example, in Figure 2.8, the predecessors of T_5 are nodes T_2, T_3, and B, these nodes have to complete before node T_5 can begin, whereas node E is the successor of T_5 and cannot commence until T_5 has finished.

Tasks that are independent of one another may be executed simultaneously. The test for independence of two different nodes T_i and T_j is that they must not be successors or predecessors of one another, i.e., node T_j is not in set **PRED**(T_i) nor is it in set **SUC**(T_i). For example, in Figure 2.8, T_5 can be executed in parallel with T_1 or T_4, since T_1 and T_4 are not in either of the sets **PRED**(T_5) and **SUC**(T_5).

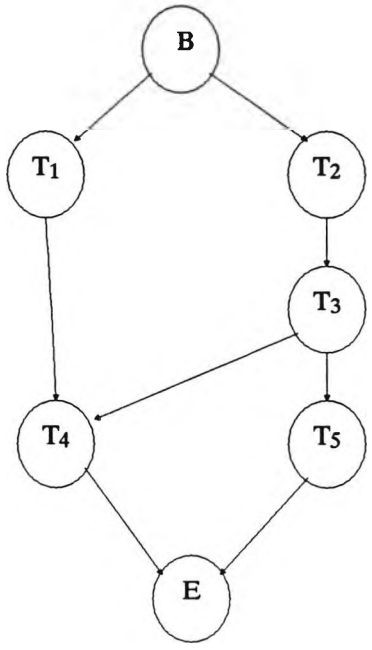


Figure 2.8 Directed acyclic graph (DAG)

2.2.5 Path costs and the critical path

The cost of traversing a path, in terms of execution time, is the accumulated execution time of each task on that path. For a path (T_1, ..., T_i, ..., T_k) the cost of traversing that path is (e_1 + ... + e_i + ... + e_k), where e_i is the worst-case execution time of the task corresponding to node T_i. The cost function is denoted COST(T_i) and is equal to e_i, similarly COST(T_1, ..., T_i, ..., T_k) is equal to the cost of traversing the path (T_1, ..., T_i, ..., T_k).

The longest path, in terms of execution time, from node B to node E is called the *critical path* (Figure 2.9). The cost of traversing the critical path, denoted w^*, is defined as:

$$w^* = \text{MAX}\{\text{COST}(B, ..., E)\}, \tag{2.1}$$

where the cost associated with nodes B and E is zero. Critical path length (time) is important, since it defines a lower bound on the overall execution time of the algorithm, ignoring resource limitation and communication overhead.

Minieka (1978) describes efficient methods for detecting the critical path in graphs, known as the critical path method (CPM). The uncertainty in task execution costs is discussed in Moder and Phillips (1970); where a weighted average cost is used, combining optimistic, realistic and pessimistic costs at a ratio of $1/6^{th}$, $4/6^{th}$ and $1/6^{th}$ respectively. The justification, in our case, for only using pessimistic (i.e. worst-case) costs is given in section 2.1.6.

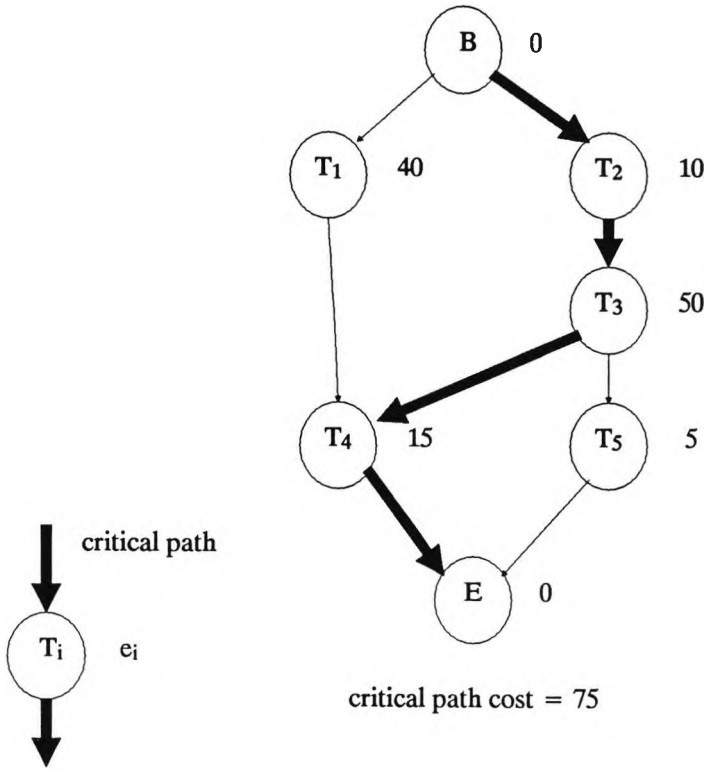


Figure 2.9 DAG, showing the critical path

2.2.6 Earliest and latest start times and float

Once all the nodes T_i in **T** have been allocated their execution time e_i, the earliest and latest start times can be evaluated. The earliest start time of a node T_i is denoted EST(T_i), and the latest start time is denoted LST(T_i). Evaluation of EST(T_i) and LST(T_i) is a necessary step in finding the critical path and is part of the critical path method (Minieka, 1978).

EST(T_i) is the *earliest* possible time node T_i can begin executing, assuming node B starts at time zero. EST(T_i) is defined as the cost of the longest path from node B to the *immediate* predecessor of node T_i, i.e.

$$\text{EST}(T_i) = \text{MAX}\{\text{COST}(B, ..., \textbf{IMPRED}(T_i))\} \qquad (2.2)$$

A special case of EST(T_i) is where $T_i = E$. This is the cost of the critical path, which is defined:

$$\text{EST}(E) = \text{MAX}\{\text{COST}(B, ..., \textbf{IMPRED}(E))\} = w^* \qquad (2.3)$$

and equates to our previous definition (section 2.2.5) since the execution cost of node E is zero.

LST(T_i) is the *latest* possible time node T_i can begin executing without extending the length of the critical path. LST(T_i) is defined as the critical path cost minus the cost of the shortest path from node T_i to node E, i.e.

$$LST(T_i) = w^* - MIN\{COST(T_i, ..., E)\} \quad (2.4)$$

The difference between the earliest start time and the latest start time of a task is called the float. Float is the *maximum* time a task T_i can be delayed beyond EST(T_i) without extending the length of the critical path. The float of task T_i, FLT(T_i), is defined as:

$$FLT(T_i) = LST(T_i) - EST(T_i) \quad (2.5)$$

The value of float is normally non-negative, since LST(T_i) > = EST(T_i).

Float is a minimum, in this case zero, for all tasks that lie on the critical path. Figure 2.10 illustrates the single critical path, and shows the cost, earliest start time, latest start time and float for each task in the DAG. The DAG does not preclude the existence of more than one critical path. Indeed, there may be many such paths which branch or join within the DAG. However, for every DAG there exists at least one path that is critical, beginning at B and ending at E.

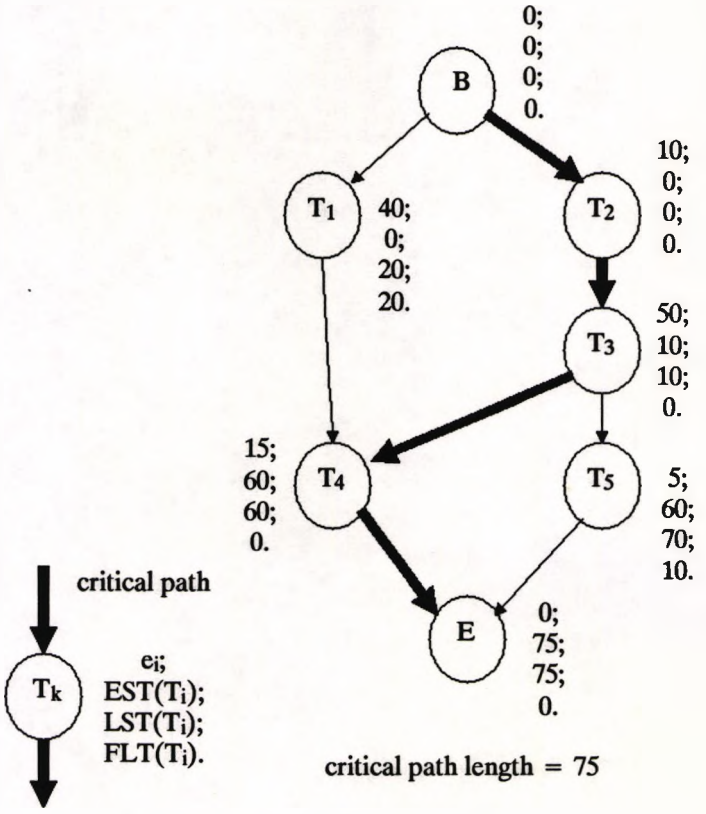


Figure 2.10 DAG, showing EST(T), LST(T) and FLT(T)

2.3 Summary

The first part of this chapter reviewed the characteristics of discrete algorithms and discussed discreteness, complexity, memory, structure and composition. The rules for maintaining synchronisation in real-time systems between algorithm and input device, and algorithm and output device have been established. Moreover, the synchronisation requirement has shown that for real-time systems, the algorithm must be deterministic.

The algorithm has been expressed as a set of disjoint tasks whose complexity influences the granularity of the structure. The cost (or execution time) of different task types is chosen to be small (i.e. low complexity tasks), hence an algorithm consists of many medium/fine grain tasks. Choosing medium/fine grain tasks, rather than coarse grain tasks (i.e., process level tasks), enables a potentially high degree of parallelism to be represented.

The latter part of this chapter introduced the graph **G** = (**T**, **C**, B, E, **A**) (nodes and arcs) which has been shown to be transitive, irreflexive and asymmetric. These characteristics ensure the graph is both directed and acyclic (i.e. a DAG). The DAG is suited to representing deterministic algorithms and is completely equivalent to the algorithm in terms of function and structure, hence any parallelism is preserved.

Once execution times have been assigned to tasks (or nodes), the CPM (critical path method) can be applied to **G**. CPM produces earliest and latest start times for all the tasks in **T** and gives the earliest overall cost for completion of the algorithm irrespective of the resource constraints. Results from the CPM are to be used for analysing the DAG, with a view to scheduling the tasks onto processors.

Chapter 3. DFDL design aspects

In chapter 2 the characteristics of discrete algorithms and their directed acyclic graphs (DAG) were discussed. The discrete algorithm and DAG are regarded as a program's source and object respectively (Figure 3.1). The constraints governing **G** are reflected back into the programming language such that the graph's transitive, asymmetric, irreflexive relation on data dependency becomes the language's single-assignment rule.

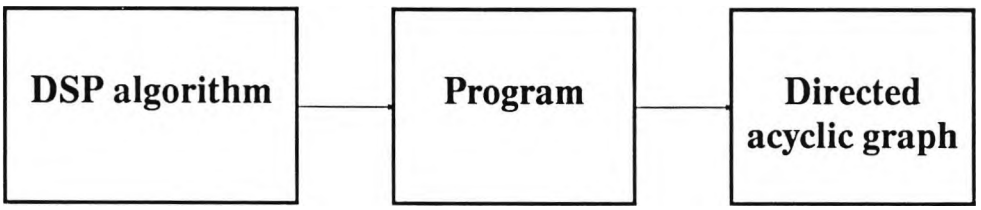


Figure 3.1 Program source and object

The purpose of a language is to present a medium in which an algorithm can be described without loss of functional integrity or structure. However, the language should prevent invalid items, such as incorrect syntax, inadmissible structures and items out of context. In order to ensure program portability a language should be closely related to the problem domain and detached, as far as possible, from influences arising from the processor architecture. With these aims in mind, the design aspects of Digital Filter Description Language (DFDL) are presented.

3.1 Language grammar

The description of a language is the grammar of the language. Assuming a language is made up from sentences (which in turn comprise words, which in turn comprise letters) then a grammar shows how sentences can be built up using successive expansions of strings of symbols. There are two types of symbols; terminal and non-terminal. Here, terminal symbols and non-terminal symbols are distinguished from one another by expressing terminal symbols in a plain font and non-terminal symbols in an *italic font*. BNF (Backus Naur Form) productions are used to state the rules of grammar.

Chomsky concluded that there are four types of grammar; type 0 in which the form of productions is unlimited, while types 1, 2, and 3 are categorised by increasing restrictions on the form of productions available (Bornat, 1979). The two types of grammar that interest us are types 2 and 3.

- A type 2 (or context free) grammar contains only productions of the form:

  A ::= **alpha**

  where *A* is a single non-terminal symbol and **alpha** is a string of terminal and/or non-terminal symbols.

- A type 3 (or regular expression) grammar contains only productions of the form:

  A ::= a
  A ::= a *B*

  in which *A* and *B* are single non-terminal symbols, a is a single terminal symbol and the second production is right recursive. An alternative definition exists for the second production, where *A* and *C* are single non-terminal symbols, a is a single terminal symbol and the second production is left recursive.

  A ::= a
  A ::= *C* a

  Type 3 grammar must either be left or right recursive, but not both.

In the case of type 2 and 3 grammars it is possible to define three important properties of a symbol which appears on the left hand side of a production (::= + means a production in one or more steps):

- If *A* ::= + **alpha** *A*, then symbol *A* is right recursive.
- If *A* ::= + *A* **beta**, then symbol *A* is left recursive.
- If *A* ::= + **alpha** *A* **beta**, then symbol *A* is self embedding.

The final case (self embedding) cannot occur in type 3 grammars and its presence is often used to distinguish between type 2 and 3 grammars. Similarly, left and right recursive symbols cannot exist together in type 3 grammars, but can in type 2.

Ambiguity arises in a grammar when it is possible to produce two or more distinct derivations for the same sentence. Ambiguity is a problem because of the confusion it introduces about the interpretation of a sentence. Any grammar that contains a symbol that is both left and right recursive will be ambiguous.

DFDL's lexical grammar is entirely type 3 and right recursive. A language defined by type 3 grammar can be recognised by a finite state machine, in which there are states that correspond to non-terminal symbols and in which the state transitions are determined by the terminal symbols in the productions of the grammar. Hence, every decision the lexical analyser makes is based on the last terminal symbol read. Finite state machines are easy to implement and are highly efficient.

DFDL's syntax grammar is type 2 and consequently somewhat more complex than its lexical grammar. All left recursive productions have been removed, leaving right recursive and self embedding productions. All the grammar is classified as one-track grammar, which is unambiguous. The one-track grammar is parsed by a top-down one-track parser which separates parenthesised expressions (i.e. self embedding) into individual regular expressions (i.e. right recursive) before parsing. This allows regular expressions to be treated individually as type 3 grammar. There is no priority between operators in DFDL, hence, extensive use is made of parentheses in expressions.

Error detection and reporting is very effective when using a top-down one-track parser. At each stage all expected terminal and non-terminal symbols are known to the parser and when an unexpected symbol occurs an error of the form "expected ..., found ..." can be given. One-track error detection compares favourably with some early compilers which suffered from backtracking (tend to pass the error before they detect it) and as a consequence could only produce error messages like "syntax error".

An extended BNF description of DFDL lexical grammar and syntax grammar is given in Appendix B and a detailed description of DFDL is given in chapter 4.

3.2 Single-assignment

Single-assignment languages (SALs) have the appearance of traditional imperative languages, in that they incorporate the assignment statement and typical control flow statements such as conditional constructs and loops. However, SALs impose a rule that a variable is only assigned once in a program (Chamberlin, 1971). This rule significantly alters the nature of the assignment operator, changing it from a dynamic destructive operation to a static operation that associates a name to a data value.

Single-assignment has implications on how programming constructs are used. For example, repetitive statements in SISAL employ an "old" operator to distinguish between the new state of a variable and its old state while in a loop:

```
for initial
    R := X / 20.0
    P := 0.0
    while X > R
        repeat
            R := old R + 3.14
            P := old P + 1.0
    returns value of P
end for
```

Single-assignment languages like Id (Arvind, Gostelow and Plouffe, 1978) and Lucid (Ashcroft and Wadge, 1977) are similar to SISAL, but use a "new" operator:

```
(initial R <- X / 20.0; P <- 0.0
    while X > R do
        new R <- R + 3.14;
        new P <- P + 1.0;
    return P)
```

Single-assignment also bars the type of conditional statement found in most imperative languages, where assignment is both multiple and conditional. SALs restrict conditional statements to single-assignment and unconditional choice. For example, the following SISAL expression selects the greater value P or R and makes it equal to S:

```
S := if P > R then P else R
```

The static nature of single-assignment allows the normal ordering restrictions found in imperative languages (e.g. Fortran, Pascal, etc.) to be relaxed. Once data on the right hand side of the assignment statement is available, the expression can be executed. This property lends itself to data-driven execution which can be applied to dataflow architectures or data-driven multi-processor architectures.

Perhaps the most important property of SALs, is that they inherently preserve an algorithm's structure (i.e. dependency relations between tasks), due to the non-destructive nature of assignment. Structural preservation also protects any parallelism that resides within the algorithm from being destroyed. Hence, SALs have no need for explicit parallel constructs, since they inherently express parallelism.

3.3 Program flow

In chapter 2 a discrete algorithm is defined as a repetitive process, with zero or more external inputs and one or more external outputs. In order to satisfy this definition, DFDL conforms to a strict flow (Figure 3.2). Before entering the repetitive input-process/generate-output (or just generate-output) cycle, DFDL includes an optional initialise section, this is in keeping with algorithms which often require a known, pre-set initial state. Initialise is executed once only, if executed at all. Program termination occurs after either a specified number of iterations, an arithmetic error (section 3.5), or when stopped by the user. DFDL program flow is illustrated in Figure 3.2.

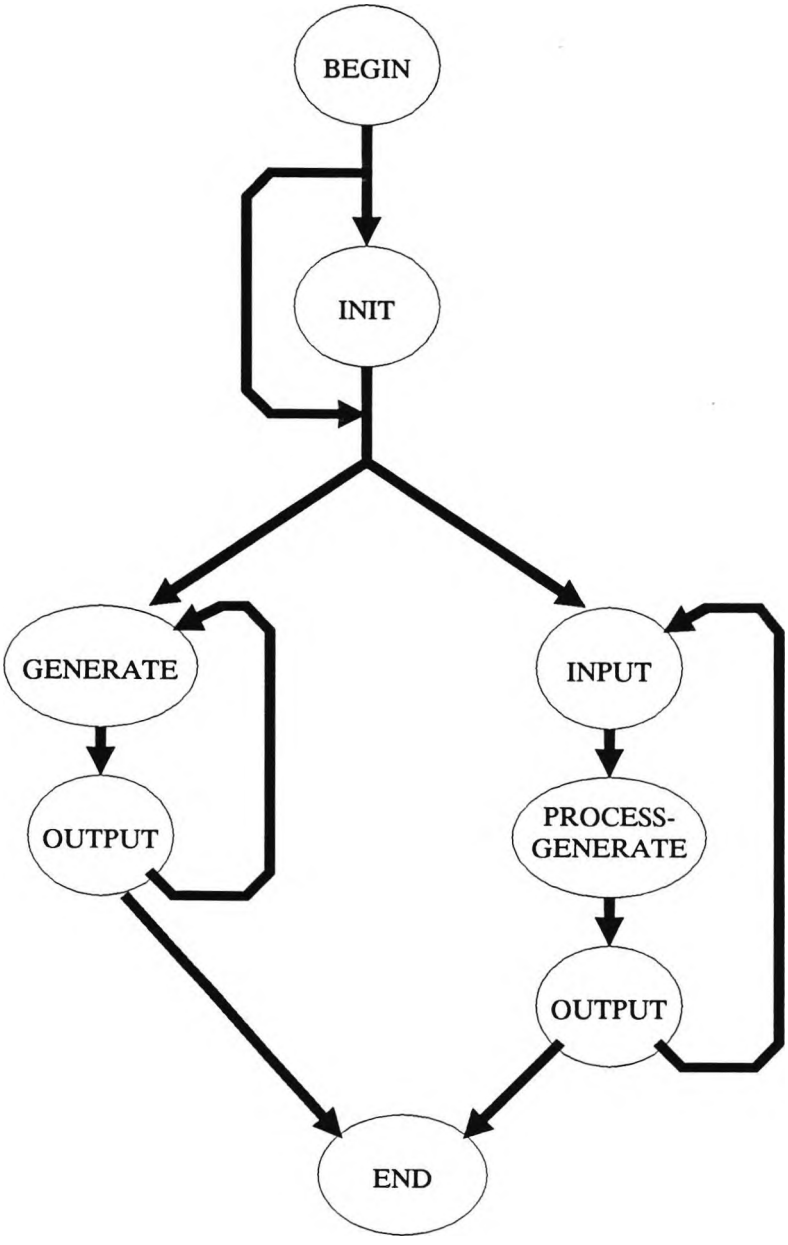


Figure 3.2 DFDL program structure models

3.4 Past values

The majority of DSP algorithms not only use present values in their result, but also past values from previous iterations. The single-assignment rule causes difficulties in expressing assignment from historical values, so DFDL includes an operator that gives a programmer access to any past value from previous iterations. This operator is called Z, named after its discrete equivalent. Z is more extensive than SISAL's "old" operator, because it can refer to past values from any previous iteration. This is achieved by post scripting Z with an integer value, e.g. Z[n]. Z is causal, so only past values are admissible and these values cannot be re-assigned since this would violate single-assignment. Generally, Z is intended to simplify working under single-assignment and facilitates an ordered mechanism for passing values from past iterations to the present iteration.

3.5 Error handling

Single-assignment languages tend to exhibit locality of effect, that is their operators do not have unnecessary far reaching data dependencies. Locality of effect requires that arithmetic errors are handled by error values rather than some global error flags or program interruption.

If an error occurs the system should react in a deterministic manner and not in an uncontrolled or unpredictable way. This is achieved by propagating error values along the flow of data; if an argument to an arithmetic operator is an error value, then the result is also an error value. Upon reaching the end of an iteration a propagated error value prevents output. In this way, the entire computation will come to a stop, yielding an error value as its result. If the processor keeps a record of every error generated and propagated, then the point where the error occurred may be traced.

The DFDL run-time system should detect and convey arithmetic errors such as overflow, underflow, divide by zero, negative square root, unstable result, inaccurate result, remainder from infinity, remainder by zero and undefined result. This is possible when using an arithmetic data type that is floating point format ANSI/IEEE std.745-1985.

3.6 Data types

To provide the language with a degree of flexibility a range of different arithmetic data types are necessary, e.g. byte, integer, floating point. Those data types different from floating point ANSI/IEEE std.745-1985 are restricted to input and output, and are converted to floating point ANSI/IEEE std.754-1985 on entry and exit respectively. This strategy is necessary in DFDL because of the error handling system (section 3.5). Additionally, a wide dynamic range and time efficient processing make floating point format an attractive proposition for digital signal processing applications.

Boolean variables, although present in DFDL are not explicit. Their use is restricted to the conditional statement where they are created and they cannot be transported to other statements. Other data types, such as strings of characters, are excluded from DFDL.

3.7 Scope and use of variables

In order to prevent side-effects and indeterminate program structures the scope and use of variables in DFDL is restricted. Scope is limited to the program unit in which the variable is declared. Consequently, all values passed between program units are passed by formal parameter only and the existence of global variables is disallowed. This locality rule does not extend to constant values, which have global scope. However, this does not mean that a variable may assume constant status and become global, since all variables in DFDL are assigned once every iteration and cannot therefore act as constant values.

DFDL bans the use of data values in indexing arrays or in determining the number of iterations of repetitive constructs. This limitation may appear to be severe, however, it is a necessary condition for determinacy and hence, ensures a DAG can be produced from a program. Consider, if the following were allowed:

```
input(R)
P := 4.231
while R > 19.24
    R := R / P
    P := P + 3.8
```

R is assigned an external value at run-time and is therefore unknown at compile-time. Consequently, the number of iterations required to satisfy the post-condition (R < = 19.24) cannot be determined at compile-time. The program fragment could make one, none, or many tens of thousand of iterations before it terminated. A similar result would occur if the input operation were replaced by a random or unstable function, or anything which gave R an ill defined or unknown value at compile-time.

3.8 Program units

DFDL does not allow the use of procedures because of their susceptibility to side-effects, however, it does make heavy use of functions, which are always free from side-effects. DFDL uses functions as arguments, since they evaluate to a single result. The result of a function is not passed as a formal parameter, but is represented by the instantiation of the function. In order to maintain a deterministic structure, functions cannot call themselves directly, or indirectly. Scope is restricted to the program unit that declares the function and all other functions in the program unit declared after the function.

Individual DFDL programs may be connected together, input to output, by the use of common formal parameters. Where necessary, this allows periods of iteration to differ within different programs. Common passed data synchronises the separate program units according to the declared protocol of the input or output. Program nesting is prohibited, since programs would then assume the role of procedures.

3.9 Summary

The type of lexical grammar and syntax grammar for DFDL has been established and its effects on parsing and error detection/reporting have been discussed.

At a higher level, we have seen the need to impose a single-assignment rule on the language because of the deterministic nature of the DAG. Single-assignment has been shown to have many effects, one of which is to preserve parallelism. Other effects have shown an influence on programming constructs such as conditionals, and repetitive constructs. As a result of single-assignment the Z operator has been incorporated, which is shown to solve the problem of moving data from past iterations.

Program flow has been modelled on the repetitive input-process-output (or generate-output) cycle of a discrete algorithm. This has made DFDL suitable for describing deterministic discrete processes (i.e. sampled systems) that have zero or more inputs and one or more outputs.

The restrictions on the type, scope and use of variables have been discussed and the ability for error handling described. Finally, the different types of program unit have been presented and the rules governing their use informally stated. A more formal description of DFDL is given in Appendix B and the design aspects outlined here are developed in the following chapter.

Chapter 4. DFDL definition and syntax

In this chapter the operators, functions and program constructs of DFDL are described in detail. The language description is divided into three parts. The first part is a short section which describes the program facilities. The second illustrates the composition of lexical units, e.g. names, reserved words, numbers, operators. The third part describes DFDL's syntax with the aid of extended BNF productions. An explanation of extended BNF is given at the beginning of the chapter. The final section of the chapter gives several examples of DFDL programs.

4.1 BNF notation

A variant of Backus-Naur form (BNF) notation, known as extended BNF (or EBNF) is employed to describe the syntactic and lexicographic relationships in DFDL. A grammar consists of a number of production rules, which define lexicographic and syntactic categories in terms of other lexicographic and syntactic categories, and which define the terminal symbols belonging to DFDL. Terminal and non-terminal symbols are distinguished from one another by expressing all non-terminal symbols in *italics*. Terminal symbols can only appear on the right hand side of a production, unlike non-terminal symbols which may appear either side. Note, some lexical non-terminal symbols are treated as terminal symbols for syntax descriptions, e.g. real, ident, integer, etc.

The examples below are used to illustrate the meaning of the EBNF operators; |, { n }, *[ ]*, and ::=.

4.1.1 Production

The ::= operator is read "is defined to be", hence the meaning of

monadic.boolean.op ::= NOT

is "*monadic.boolean.op* is defined to be NOT".

4.1.2 Alternative

The operator | is read as "OR", hence the meaning of

data.type :: = BYTE | INT16 | INT32 | REAL32

is "*data.type* is defined to be BYTE or INT16 or INT32 or REAL32". This production may also be written

data.type :: = BYTE
data.type :: = INT16
data.type :: = INT32
data.type :: = REAL32

The operator *[ ]* also provides a way of expressing alternatives where there is commonalty between the different productions. The meaning of

sub.size :: = *col.size [row.size]*

is "*sub.size* is defined to be *col.size* or *col.size* followed by *row.size* ". This production could be re-written

sub.size :: = *col.size* | *col.size row.size*

4.1.3 Repetition

The repetition operator { n *symbol*} means that the enclosed *symbol* may be produced n or more times, hence the meaning of

real.string :: = real { 0 , real }

is "*real.string* is defined to be real or real, real or real, real, real etc.".
The { n *symbol*} operator yields a recursive production, this is evident when re-written

real.string :: = real | real, *real.string*

4.2 Program facilities

4.2.1 Continuation

Expressions normally occupy a single line. Where it is necessary to break a line and spread an expression over several lines the continuation symbol "..." must be used. The continuation symbol is placed at the end of each broken line, immediately before or immediately after an operator. For example:

```
y := (x1 + x2) + ...
  (x3 + x4)
```

4.2.2 Comments

Comments are introduced by the percentage character "%". A comment may follow a statement, occupy an entire line or reside within a statement. The percentage character may be used to toggle comments on and off. All comments are toggled off when the end of line is reached, e.g.,

```
y := x      % y becomes x.
```

4.3 Lexical types

The DFDL character set comprises alphabetic, numeric and special characters.

4.3.1 Alphabetic characters (letters)

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
```

4.3.2 Numeric characters (digits)

```
0123456789
```

4.3.3 Special characters

```
% * ( ) - + = : [ ] ; \ < > . / space
```

Characters may be combined to produce different lexicographic types (e.g. identifiers, reserved words, integers, numbers), all of which are separated from one another by the space character.

4.3.4 Delimiters

Delimiters are constructed from special characters and comprise characters which symbolise arithmetic operators, relational operators, brackets, etc. The valid set of delimiters are:

```
% | * | ** | ( | ) | - | + | := | = | [ | ] | ;
 | \ | < = | < | < > | > | > = | . | ... | /
```

4.3.5 Identifiers

Identifiers are strings of characters used to identify (or name) some element in the program. The types of element that are named by identifiers are certain operands, objects, repetition identifiers, user-defined functions and the program name.

Identifiers consist of a sequence of lower-case letters, digits and dots, of which the first character must be a lower-case letter.

ident :: = *lc.letter* { 0 *lc.letter* | *digit* | . }

4.3.6 Reserved words

Reserved words consist of a sequence of upper-case letters used to identify DFDL functions and key words.

reserved.word :: = { 1 *uc.letter* }

The legal set of reserved words are:

```
ABS | ACOS | ALOG | AND | ASIN | ATAN | BEGIN | BYTE | COS
 | ELSE | ELSEIF | END | EVERY | EXP | EXPRESSION | FOR
 | FOREVER | FROM | FUNCTION | IF | INIT | INPUT |  INT | IS
 | LN | LOG | MAX | MIN | MEAN | MED | MOD | NODE | NOT
 | OR | OUTPUT |  PROD | PROG | REAL | REPEAT | RESULT
 | SGN | SIN | SQRT | SUM | TABLE | TAN | THEN | VALUE | Z
```

4.3.7 Integer numbers

Integers are used to specify the size of arrays and may be used to subscript single array elements and repetitive arrays. Integers consist of a sequence of one or more digits, which may be preceded by a sign.

integer :: = [+ | -] { 1 *digit* }

Integers are held in a 32-bit 2's complement form.

External data may be of integer type BYTE, INT16, INT32 as well as of type REAL32. The valid range of positive integers p is, $0 <= p <= +(N/2) - 1$, while the valid range of negative integers n is, $0 > n >= -(N/2)$, where N is 2^8 for BYTE integers, 2^{16} for INT16 integers and 2^{32} for INT32 integers.

4.3.8 Real numbers

Floating point numbers (real numbers) are used to represent all data values within DFDL programs. A *real* consists of a two sequences of digits, separated by a decimal point, which may be preceded by a sign and may be succeeded by an exponent.

real :: = [*monadic.op*] { 1 *digit* } . { 1 *digit* } [*exponent*]
exponent :: = E *monadic.op* { 1 *digit* }
monadic.op :: = + | -

A *real* is of type REAL32 which has a format according to ANSI/IEEE standard 745-1985. REAL32 has 1 sign bit s, an 8-bit exponent e and a 23-bit fraction f. The value is positive if $s = 0$ and negative if $s = 1$, its magnitude is

$2^{(e - 127)} * 1.f$ if $0 < e < 255$
$2^{-126} * 0.f$ if $e = 0$ and $f <> 0$
0 if $e = 0$ and $f = 0$.

The range of a REAL32 value r, is approximately, $-3.4 * 10^{38} <= r <= +3.4 * 10^{38}$, these values are referred to as -x.max and +x.max respectively.

4.3.9 Real errors and overflow

When an argument to an operation is outside the domain of that operation or the argument is not-a-number (NaN), the operation produces an exceptional result. This result may be NaN, + Inf or -Inf. NaNs are used to designated different causes of an error, while Infs mean the result is too large to be represented as REAL32 format. NaNs and Infs are propagated from one operation to the next, this facilitates error and overflow detection. Different error conditions are shown below, the 32-bit error codes are in hexadecimal format.

7FC00000	Divide zero by zero.
7FA00000	Divide infinity by infinity.
7F900000	Multiply zero by infinity.
7F880000	Addition of opposite signed infinities.
7F880000	Subtraction of same signed infinities.
7F840000	Negative square root.
7F804000	Remainder from infinity.
7F802000	Remainder by zero.
7F800010	Result not defined mathematically.
7F800008	Result unstable.
7F800004	Result inaccurate.

4.4 Syntax

The rules for defining legal sequences of lexically correct elements are given in this section. The lexical non-terminal symbols *real, ident* and *integer* become terminal symbols in the syntax definition and are written in a non-italicised script, e.g. real, ident, integer. Reserved words and combinations of special characters also become terminal symbols, these are expressed in their lexical form.

4.4.1 Operators and functions

4.4.1.1 Arithmetic operators

DFDL operators are all of type REAL32. The arithmetic operators +, -, *, /, \ and ** yield the arithmetic sum, difference, product, quotient, remainder and power respectively. Remainder x \ y produces the result x - (y * n), where n is the result of x / y rounded to the nearest integer value. Power X ** Y is only defined for X > = 0, since values of X that are less than zero may produce complex (i.e. real + imaginary) results. All results are of type REAL32 which are rounded to the nearest floating point value.

Arithmetic operators are classified as either dyadic or monadic operators:

dyadic.op ::= + | - | * | / | ** | \
monadic.op ::= + | -

4.4.1.2 Boolean operators

Within conditional expressions DFDL employs boolean and relational operators, these operators are not valid outside conditional expressions. The boolean operators NOT, AND, OR yield the boolean result b, true or false:

NOT false = true
NOT true = false
false AND b = false
true AND b = b
false OR b = b
true OR b = true

where b is a boolean variable of value true or false.

4.4.1.3 Relational operators

The operators =, < >, >, > =, <, < = are all dyadic relational operators which compare the value of two real numbers and yield a boolean result, true or false. The result of x = y is true if the value of x is exactly equal to the value of y. Other relational operators obey the rules:

(x < > y) = NOT(x = y)
(x > y) = (y < x)
(x > y) = NOT(x < = y)
(x < y) = NOT(x > = y)

where x and y are real values.

Production rules for boolean and relational operators

boolean.op ::= *monadic.boolean.op* | *dyadic.boolean.op*
monadic.boolean.op ::= NOT
dyadic.boolean.op ::= AND | OR
relational.op ::= = | < > | < | < = | > | > =

4.4.1.4 Functions

DFDL functions (non user-defined) all act on real data (with the exception of REAL) and produce a single real result. Some functions have just one operand, while others have a multiple number of operands. Details concerning the range and accuracy of the arithmetic and trigonometrical functions can be found in the Occam standard library documentation (INMOS, 1987).

Each of the functions below are expressed y := function(x), where x and y are the operand and object respectively.

y := REAL(x), y becomes the real equivalent of integer x,
domain $[-2^{31}, +2^{31} - 1]$.
y := ABS(x), y becomes the modulus of x,
domain [-Inf, +Inf].
y := SGN(x), y becomes +1.0 if x >= 0 and -1.0 otherwise,
domain [-Inf, +Inf].
y := SQRT(x), y becomes the square root of x,
domain [0, x.max].
y := LOG(x), y becomes $\log_{10}(x)$,
domain [0, x.max].
y := ALOG(x), y becomes 10^x,
domain [-Inf, +38.53].
y := LN(x), y becomes $\log_e(x)$,
domain [0, x.max].
y := EXP(x), y becomes e^x,
domain [-Inf, 88.72].
y := SIN(x), y becomes sin(x), where x is in radians,
domain [-12868.0, +12868.0]
y := COS(x), y becomes cos(x), where x is in radians,
domain [-12868.0, +12868.0]
y := TAN(x), y becomes tan(x), where x is in radians,
domain [-6434.0, +6434.0]
y := ASIN(x), y becomes $\sin^{-1}(x)$, where y is in radians,
domain [-1.0, +1.0]
y := ACOS(x), y becomes $\cos^{-1}(x)$, where y is in radians,
domain [-1.0, +1.0]
y := ATAN(x), y becomes $\tan^{-1}(x)$, where y is in radians,
domain [-Inf.0, +Inf.0]

where the value +x.max corresponds to the largest valid REAL32 value (approximately $3.4 * 10^{38}$) and -x.max = -(+x.max).

Each of the functions below are expressed y := function(x_0, ..., $x_{(k-1)}$), where x_0, ..., $x_{(k-1)}$ and y are the operands and object respectively and k > = 2.

y := SUM(x_0, ..., $x_{(k-1)}$), y becomes the sum of all operands x_0, ..., $x_{(k-1)}$.
y := PROD(x_0, ..., $x_{(k-1)}$), y becomes the product of all operands x_0, ..., $x_{(k-1)}$.
y := MEAN(x_0, ..., $x_{(k-1)}$), y becomes the mean of all operands x_0, ..., $x_{(k-1)}$.
y := MED(x_0, ..., $x_{(k-1)}$), y becomes the median of all operands x_0, ..., $x_{(k-1)}$.
y := MAX(x_0, ..., $x_{(k-1)}$), y becomes the maximum of all operands x_0, ..., $x_{(k-1)}$.
y := MIN(x_0, ..., $x_{(k-1)}$), y becomes the minimum of all operands x_0, ..., $x_{(k-1)}$.

Production rules for functions

function :: = *monadic.function* | *multi.function* | *conv.function*
monadic.function :: = ABS | SGN | SQRT | LOG | ALOG | LN | EXP
| SIN | COS | TAN | ASIN | ACOS | ATAN
multi.function :: = SUM | PROD | MEAN | MED | MAX | MIN
conv.function :: = REAL

4.4.2 Program

A DFDL program is structured as shown below:

```
PROG prog.name ( {1 { 0 input.declaration } { 1 output.declaration } } )
 {0 { 0 node.declaration }
 { 0 constant.declaration }
 { 0 function.declaration } }
BEGIN
 [ INIT initialise.section ]
REPEAT ( FOREVER | FOR non.neg.integer ) repeat.section
END
```

The first line of a DFDL program begins with the key word PROG followed by the program's name. This is succeeded by an optional external input declaration and a mandatory external output declaration, both of which may be repeated any number of times. Output declaration is followed by node, constant and function declarations, all of which may be declared zero or more times.

The BEGIN and END keywords embrace the assignment part of the program, which begins with an optional initialisation section. This section allows the initial state of the program to be set and, if used, is executed only once. The main program (repeat section) comes after the REPEAT statement. The directive immediately following REPEAT can either be FOREVER or FOR n, where n is a non-negative integer. If n is set to zero, the program will terminate without executing the repetitive section. When the repeat directive is declared as FOREVER, the contents of the loop are executed until the loop is stopped by external means, e.g., power down or reset.

Production rules for the program structure

program :: = PROG *program.name* (*ext.declaration*)
internal.declaration BEGIN *assignment.section* END
program.name :: = ident
ext.declaration :: = {1 { 0 *input.declaration* } { 1 *output.declaration* } }
internal.declaration :: = {0 { 0 *node.declaration* }
{ 0 *constant.declaration* } { 0 *function.declaration* } }
assignment.section :: = [INIT *initialise.section*]
REPEAT (FOREVER | FOR non.neg.integer) *repeat.section*

4.4.3 External input and output

DFDL employs two types of external data interface, denoted input and output. These are used to interface to other DFDL programs and the outside world. Each DFDL program has zero or more inputs and one or more outputs.

Input and output are declared after the program declaration and begin with the key words INPUT and OUTPUT respectively. Attached to each input and output keyword is the data type of the external data. External data types are either 8-bit integer (BYTE), 16-bit integer (INT16), 32-bit integer (INT32) or 32-bit floating point (REAL32). When more than one data type is used, say for different inputs, then the key word INPUT has to be re-written:

INPUT(BYTE) $u_1, u_2, \ldots, u_k$ INPUT(REAL32) $u_{k+1}, u_{k+2}, \ldots, u_p$
OUTPUT(INT32) $v_1, v_2, \ldots, v_l$

Within a DFDL program all data manipulation is performed in type REAL32. Hence, all non-REAL32 type data from external inputs and all non-REAL32 type data to external outputs has to be converted. At the boundaries of the program (i.e. input and output), DFDL implicitly inserts conversion operators which transform non-REAL32 input data to REAL32 data, and transform REAL32 data to non-REAL32 output data. Data type declaration is only necessary for inputs and outputs, since all other data structures are of type REAL32 by default.

Input and output variables are declared as either scalars or arrays and their scope is restricted to the program unit where they are declared. Arrays are distinguished from scalars by the square braces [pos.integer] which succeed the variable name and the non-singular subscript size.

The size of an input or output array does not define the number of distinct inputs or outputs, but the number of elements streamed through the declared input or output per iteration. For example, the following single input streams 1024 sequentially ordered bytes:

INPUT(BYTE) u[1024]

DFDL supports arrays of up to two spatial dimensions. The number of elements in a two dimensional array is the product of the two declared subscript sizes. For example, the following output array has 20 elements, (10 columns of 2 rows):

OUTPUT(REAL32) v[10][2]

Data is streamed to the output (above) in a "column before row" fashion, e.g., v[0][0], v[1][0], v[2][0], ..., v[9][0], v[0][1], ..., v[8][1], v[9][1].

Production rules for input and output declarations

ext.declaration ::= { 1 { 0 *input.declaration* } { 1 *output.declaration* } }
input.declaration ::= INPUT(*data.type*) *input [sub.size]* { 0 , *input [sub.size]* }
output.declaration ::= OUTPUT(*data.type*) *output [sub.size]* { 0 , *output [sub.size]* }
data.type ::= BYTE | INT16 | INT32 | REAL32
input ::= ident
output ::= ident
sub.size ::= *column.size [row.size]*
column.size ::= [pos.integer]
row.size ::= [pos.integer]

4.4.4 Nodes

Nodes are used as intermediate variables between inputs and outputs. Each DFDL program has zero or more nodes.

Nodes are declared after the external declaration and begin with the key word NODE. All nodes are of data type REAL32, hence their data type is not explicitly declared:

NODE n_1, n_2, ..., n_k

Node variables are declared as either scalars or arrays and their scope is restricted to the program unit where they are declared. Arrays are distinguished from scalars by the square braces [pos.integer] which follow the variable name and the non-singular subscript size. As in the case for inputs and outputs, nodes can have up to two spatial dimensions. For example, the following node is declared as a 50 element vector;

```
NODE n[50]
```

whereas the example below shows a node array of 80 elements, (10 columns of 8 rows):

```
NODE n[10][8]
```

Production rules for node declaration

node.declaration ::= NODE *node [sub.size]* { 0 , *node [sub.size]* }
node ::= ident
sub.size ::= *column.size [row.size]*
column.size ::= [pos.integer]
row.size ::= [pos.integer]

4.4.5 Constants

Constant values are either named, in which case they are called constants, or used directly in expressions as reals. Each DFDL program has zero or more constants.

Constant declarations begin with the key word VALUE. All constants are of data type REAL32, hence their data type is not explicitly declared:

```
VALUE c IS real
```

Constants are declared as either scalars or arrays. Arrays of different values may be declared using a table of real values. For example, a 5 element constant array is declared as:

```
VALUE TABLE c[5] IS [3.45, 2.00, -34.0, -0.91, 34.9]
```

2-dimensional constant arrays are created in a similar fashion. The example below shows a constant array of 6 elements, (3 columns of 2 rows):

```
VALUE TABLE c[3][2] IS   [0.3042, 12.346, 6.6511;
                          2.2220, 332.23, 3.8449]
```

Constants may also be formed by the assignment of constant expressions, in place of a single real value, as shown previously. Constant expressions, consist of operators, functions and previously declared constants as well as real values:

```
VALUE c IS 1.3334
VALUE d IS (c * ASIN(-0.86475))
```

The repetition operator (section 4.4.10) enables arrays of different values to be constructed by the assignment of a constant expression. In DFDL this is a two stage process; first the size of the array is declared and placed equal to EXPRESSION;

```
VALUE pi IS 3.141592654
VALUE c[10] IS EXPRESSION
```

then during the initialisation section each element of the un-defined constant is equated a value:

```
INIT
   c[index FROM 0 FOR 10] IS COS((REAL([index FROM 0 FOR 10]) * pi) ...
   - (pi / 2.0)) + 1.0
```

The subscript identifier (name of subscript index) has a value which varies from 0 to 9, this is converted from an integer to a real by the REAL function.

Production rules for constant declaration

constant.declaration :: = VALUE *(constant [sub.size]* IS
constant.expression | *undefined.constant [sub.size]* IS EXPRESSION
| TABLE *constant [sub.size]* IS *table)*
constant :: = ident
undefined.constant :: = ident
sub.size :: = *column.size [row.size]*
column.size :: = [pos.integer]
row.size :: = [pos.integer]
table :: = *real.string* { 0 ; *real.string* }
real.string :: = [real { 0 , real }]

4.4.6 User-defined functions

User-defined functions provide a degree of abstraction and can be used in a repeated fashion to reduce programming effort and program length. All operands to a function have to be passed as formal parameters or declared within the function. Those operands declared inside a function are only valid within the scope of that function. These limitations ensure that functions are free from side-effects. Functions have an in-degree of zero or more operands and an out-degree of one (i.e. a single result). Within the function declaration, a single object corresponding to the function output is mandatory, this single object is denoted RESULT.

DFDL is a static language, consequently functions cannot be called recursively, i.e., a function cannot call itself directly or indirectly. An ordering is imposed on function declarations to enforce the static restriction. This ordering only permits functions to be called from other functions, which have been previously declared. Within a function, formal parameters cannot be assigned values, otherwise the single assignment rule would be broken. In addition, operands cannot be subscripted with the delay operator from within a function, therefore any function that requires old values for its operands must have them passed as formal parameters.

Production rules for user-function declaration

function.declaration :: = FUNCTION *function.name*
([*formal.parameters*]) *function.body*
function.name :: = ident
formal.parameters :: = *formal* [*sub.size*] { 0 , *formal* [*sub.size*] }
formal :: = ident
sub.size :: = *col.size* [*row.size*]
col.size :: = [pos.integer]
row.size :: = [pos.integer]
function.body :: = {0 { 0 *node.declaration* } { 0 *constant.declaration* }
{ 0 *function.declaration* } } BEGIN { 0 *assignment* }
RESULT : = (*conditional.expression* | *expression*)

A function is called by instancing its name, followed by the passed parameters enclosed in parentheses. Passed parameters can either be singular, multiple or mixed. A comma is used to separate consecutive pairs of singular parameters, or to separate singular parameters from multiple parameters, or to separate consecutive multiple parameters. The total number of passed parameters must correspond to that declared for the function.

Production rules for user-function instanitation

function.instance :: = *function.name* ([*passed.parameters*])
function.name :: = ident
passed.parameters :: = *operand* { 0 , *operand* }

A typical function declaration and instanciation is presented below, note that the declaration of a function must precede its instanciation.

```
FUNCTION weighted.average(f.1, f.2, f.3)
   NODE n.1, n.2
   VALUE a.1 IS (1.0 / 6.0)
   VALUE a.2 IS (4.0 / 6.0)
   BEGIN
      n.1 : = (f.1 + f.3) * a.1
      n.2 : = f.2 * a.2
   RESULT : = n.1 + n.2

y : = weighted.average(x[i FROM 0 FOR 3] )
```

4.4.7 Z-operator (Delay)

DFDL incorporates a Z operator, which is equivalent to the discrete mathematical delay operator used in difference equations, where xZ[n] is equivalent to xZ^{-n}. The value xZ[n] is the value of x, n steps in the past, where a step is equivalent to a single cycle of the repetitive section of the program, i.e., a single pass of the algorithm. The number of steps n is a non-negative integer. When n is equal to zero, xZ[n] is the same as x. The value of n cannot be less than zero.

The Z operator alleviates some of the difficulties associated with looping within a single assignment language, and is in effect a developed form of the "**old**" operator, as used in SISAL, where "**old** x" is equivalent to xZ[1]. DFDL only allows inputs, outputs and nodes to be subscripted with Z[n]. Subscripting constant values with Z[n] would be meaningless, since c = cZ[n] for all n.

All Z[n] subscripted terms are implicitly updated between the time they are last used in the current cycle and the time when they are first used in the next cycle. Consequently, all past values (i.e. subscripted Z[n], n > 0) cannot be re-assigned during the repetitive section of the program, otherwise the single assignment rule would be broken.

The production rules for delay are presented later, in section 4.4.9 and 4.4.10

4.4.8 Assignment

The assignment operator := assigns a new value to an object, or more precisely, since DFDL is a single-assignment language, the assignment operator statically associates a value with a name (i.e. name of the object).

object := value

In the initialisation section, past inputs, past nodes and past outputs (i.e. subscripted with Z[n], n > 0) may be pre-assigned values to initialise the program. The initialisation section also allows un-valued constants to be given values.

The rules for assignment in the initialisation section are summarised below:

In the repetitive section of the program the rules of single assignment preclude multiple or non-assignment of objects, hence only un-assigned nodes and un-assigned outputs can appear to the left of the assignment operator. Inputs cannot be assigned, since they are assigned implicitly.

Objects	Operands
input Z[n] *output* Z[n] *node* Z[n] *undefined.constant*	*constant* *real* (*constant.expression*)

Table 4.1 Valid initialisation section assignment

The rules for assignment in the repetitive section are summarised below.

DFDL places an additional rule on the assignment operation, which states that unassigned nodes and unassigned outputs cannot be used as operands until they are assigned. This rule places a partial ordering on assignment statements, but prevents the formation of oscillatory loops within the program.

Objects	Operands
output *node*	*constant* *real* (*expression*) *input* [Z[n]] *output* [Z[n]] *node* [Z[n]] *function.instance*

Table 4.2 Valid repetitive section assignment

Production rules for assignment

assignment.section ::= [INIT *init.section*]
REPEAT (FOREVER | FOR non.neg.integer) *repeat.section*
init.section ::= { 1 *value.constant* | *init.assignment* }
value.constant ::= *undefined.constant* IS *constant.expression*
undefined.constant ::= ident
init.assignment ::= *init.object* := *constant.expression*
init.object ::= *input* { 0 *spatial.sub* } Z *temporal.sub*
| *output* { 0 *spatial.sub* } Z *temporal.sub*
| *node* { 0 *spatial.sub* } Z *temporal.sub*
repeat.section ::= { 1 *assignment* }
assignment ::= *object* := (*expression* | *conditional.expression*)
object ::= *node* { 0 *spatial.sub* } | *output* { 0 *spatial.sub* }

4.4.9 Expressions

An expression evaluates to a single real value. Expressions are constructed from operands, operators, functions and parentheses.

A regular expression consists of a single operator or function and the operands required for that operator or function. Equal priority is given to all operators and functions, hence parentheses have to be used to define priority between operators or functions in expressions that are not regular.

For example, consider the sum of four inputs, u.1, u.2, u.3, u.4, each weighted by the values 0.1, 0.3, 0.4, 0.2 respectively. Some languages would allow;

```
y := u.1 * 0.1 + u.2 * 0.3 + u.3 * 0.4 + u.4 * 0.2
```

whereas DFDL requires the use of parentheses:

```
y := ((u.1 * 0.1) + (u.2 * 0.3)) + ((u.3 * 0.4) + (u.4 * 0.2))
```

DFDL distinguishes between two forms of expression, namely expressions which evaluate to a constant value and those whose value may change from one cycle to the next. The former is called a constant expression and is the only type of expression that may be used in the initialisation section. The latter type of expression (simply referred to as an expression) is used exclusively in the repetitive section.

Production rules for constant expression and expression

constant.expression :: = *constant.operand*
| *monadic.function constant.operand*
| *multi.function constant.operand* { 0 , *constant.operand*}
| *conv.function spatial.sub* | *monadic.op constant.operand*
| *constant.operand dyadic.op constant.operand*
constant.operand :: = real | *constant* { 0 *spatial.sub* }
| (*constant.expression*)
expression :: = *operand* | *monadic.function operand*
| *multi.function operand* { 0 , *operand*} | *conv.function spatial.sub*
| *monadic.op operand* | *operand dyadic.op operand*
operand :: = real | *input* { 0 *spatial.sub* } [Z *temporal.sub*]
| *output* { 0 *spatial.sub* } [Z *temporal.sub*]
| *node* { 0 *spatial.sub* } [Z *temporal.sub*]
| *constant* { 0 *spatial.sub* } | *function.instance*
| (*expression*)
monadic.function :: = ABS | SGN | SQRT | LOG | ALOG
| LN | EXP| SIN | COS | TAN | ASIN | ACOS | ATAN
multi.function :: = SUM | PROD | MEAN | MED | MAX | MIN
conv.function :: = REAL
monadic.op :: = + | -
dyadic.op :: = + | - | * | / | ** | \

4.4.10 Repetitive and fixed subscripts

Subscripts belonging to inputs, outputs, nodes and constants can either be fixed or varied over a defined range. The repetitive subscript attempts to simplify the program by allowing an operation to be performed on every element within the same array. For example the set of DFDL statements below uses fixed subscripts;

```
y[0] := 2.0 * x[5]
y[1] := 2.0 * x[6]
y[2] := 2.0 * x[7]
```

these statements can be condensed into a single statement:

```
y[var FROM 0 FOR 3] := 2.0 * x[var FROM 5 FOR 3]
```

The repetition identifier, named "var" (any unused identifier will do), associates subscripts belonging to different variables. The repetition identifier can also be used as an argument to the expression, once it is converted from type INT32 to type REAL32, e.g.

```
y[var FROM 0 FOR 3] :=  REAL([var FROM 10 FOR 3]) * ...
                        x[var FROM 5 FOR 3]
```

The syntax of the repetition subscript is of the form [repetition.ident FROM start FOR range [EVERY step]]. The range must be equal for all repetitive subscripts which are in the same assignment statement and have the same repetition identifier. The step (step size), when not declared as in the previous examples, defaults to 1. Step size may be (i) positive, (ii) zero or (iii) negative:

(i) When the step size is positive then the subscript evaluates to an integer number which increases with r, 0 < = r < range:

```
subscript = start + (step * r)
```

(ii) When the step size is zero the subscript evaluates to a single integer value:

```
subscript = start
```

(iii) When the step size is negative the subscript evaluates to an integer number which decreases with r, 0 < = r < range:

```
subscript = start - ( | step | * r)
```

In all cases the value of the subscript must not exceed the bounds of the variable it is subscripting.

Production rules for fixed and repetitive subscripts

spatial.sub :: = *fixed.sub* | *repetitive.sub*
temporal.sub :: = *fixed.sub* | *repetitive.sub*
fixed.sub :: = [*non.neg.integer.expression*]
repetition.sub :: = [*repetition.ident* FROM *non.neg.integer.expression*
FOR *pos.integer.expression* [EVERY *integer.expression*]]

where *non.neg.integer.expression* = *integer.expression* > = 0
and *pos.integer.expression* = *integer.expression* > 0

integer.expression :: = *integer.operand*
| *monadic.integer.op integer.operand*
| *integer.operand dyadic.integer.op integer.operand*
integer.operand :: = integer | (*integer.expression*)
monadic.integer.op :: = + | -
dyadic.integer.op :: = + | - | * | MOD

4.4.11 Conditional

Conditional expressions are included in DFDL to support non-linear operations like thresholding and median filtering. A conditional expression is made up from one or more boolean expressions b_1 ... b_{k-1}, which evaluate to true or false:

IF b_1 THEN x_1 ELSEIF b_2 THEN x_2 ... ELSE x_k

After each boolean expression b_i there is a corresponding expression x_i. The conditional expression terminates with a single default expression x_k, preceded by the key word ELSE. Boolean expressions and their corresponding expressions are separated by the key word THEN. The first boolean expression is preceded by the keyword IF, while subsequent boolean expressions are preceded by the keyword ELSEIF. Conditional expressions in DFDL always evaluate to a single real value, which is assigned to the single object on the left hand side of the assignment operator.

y := IF b_1 THEN x_1 ELSEIF b_2 THEN x_2 ... ELSE x_k

Multiple assignment is prevented by assigning only one of the expressions x_i to the single object, where an expression x_i is assigned iff the boolean expression b_i is true and all preceding boolean expressions are false. Hence, the preceding boolean expressions, within the same conditional expression, have priority over their successors. Priority is established by the rule:

b_i = b_i AND NOT(b_{i-1} OR b_{i-2} OR ... OR b_1)

In its simplest form a conditional expression consists of a single boolean expression b_1:

y := IF b_1 THEN x_1 ELSE x_2

If b_1 is true then x_1 is assigned to y, otherwise x_2 is assigned to y. More complex conditional expressions can be written by including ELSEIF or by nesting conditional expressions:

y := IF b_1 THEN x_1 ELSEIF b_2 THEN x_2 ELSEIF ... ELSE x_k or

y := IF b_1 THEN (IF b_2 THEN x_2 ELSE x_m) ELSE x_k

Conditional expressions cause branching within the program which will inevitably lead to different time complexities for different branches. This uncertainty appears to contradict the rule of determinacy. However, we find that the time complexity of all branches within a conditional expression are determinate and they all converge, meeting at the assignment of the single object. Therefore, the worst-case time taken to evaluate a conditional expression is the most costly path through the conditional expression.

Production rules for conditional expression

assignment :: = *object* := (*expression* | *conditional.expression*)
conditional.expression :: = IF *boolean.expression*
THEN (*expression* | (*conditional.expression*))
{ 0 ELSEIF *boolean.expression*
THEN (*expression* | (*conditional.expression*)) }
ELSE (*expression* | (*conditional.expression*))
boolean.expression :: =
relational.expression | *monadic.boolean.op relational.expression*
| *relational.expression dyadic.boolean.op relational.expression*
relational.expression :: =
operand relational.op operand | (*boolean.expression*)
relational.op :: = < | < = | = | > = | > | < >
monadic.boolean.op :: = NOT
dyadic.boolean.op :: = AND | OR

4.5 DFDL examples

Please note that the examples given here are for a fully implemented version of DFDL, also note that coefficient values are for illustration only.

4.5.1 FIR filter

```
%
% The example is a 5 stage, finite impulse response filter. The filter has one
% input named x and one output named y and calculates the product of the
% constant array c and xZ[d], where the delay d, ranges from 0 to 4, in
% integer steps of 1. Output y becomes the sum of the products.
%
PROG fir.filter( INPUT(REAL32) x OUTPUT(REAL32) y )

  NODE n[5]
  VALUE TABLE c[5] IS [0.434, 0.782, 0.975, 0.782, 0.434]

  BEGIN
    INIT
      xZ[col FROM 1 FOR 4] := 1.0

    REPEAT FOREVER
      n[col FROM 0 FOR 5] := xZ[col FROM 0 FOR 5] * c[col FROM 0 FOR 5]
      y := SUM(n[col FROM 0 FOR 5])
  END
```

4.5.2 IIR filter

```
%
% The example is a 2nd order infinite impulse response filter. The filter has
% one input named x and one output named y and calculates the product
% of two feedback and two feedforward paths.
%
PROG iir.filter( INPUT(INT32) x OUTPUT(INT32) y )

  NODE n
  VALUE TABLE c[4] IS [0.563, 0.782, -0.345, -0.714]

  BEGIN
    INIT
      nZ[k FROM 1 FOR 2] := 1.0

    REPEAT FOREVER
      n := SUM(x, nZ[k FROM 1 FOR 2] * c[k FROM 2 FOR 2])
      y := SUM(nZ[2], nZ[k FROM 0 FOR 2] * c[k FROM 0 FOR 2])
  END
```

4.5.3 2-D convolution

```
%
% The first program "frame" places a single column, single row
% frame around the 2-D image
%
PROG frame( INPUT(BYTE) picture[256][256] OUTPUT(BYTE) framed[258][258] )

  BEGIN
    REPEAT FOREVER
      framed[0][row FROM 0 FOR 258] := 0.0
      framed[257][row FROM 0 FOR 258] := 0.0
      framed[col FROM 1 FOR 256][0] := 0.0
      framed[col FROM 1 FOR 256][257] := 0.0
      framed[col FROM 1 FOR 256][row FROM 1 FOR 256] := ...
        picture[col FROM 0 FOR 256][row FROM 0 FOR 256]
  END

%
% This program performs a 2-D convolution using a 3 by 3 window. Frame
% pre-processing prevents interference between adjacent images and successive
% lines.
%
PROG convolution( INPUT(BYTE) framed OUTPUT(BYTE) result )

  NODE column.sum[3]
  VALUE TABLE weight[3][3] IS
    [-0.982, -0.707, -0.982;
    -0.707, +6.756, -0.707;
    -0.982, -0.707, -0.982]

  BEGIN
    INIT
      framedZ[col FROM 1 FOR 515] := 0.0

    REPEAT FOREVER
      column.sum[0] := SUM(framedZ[col FROM 0 FOR 3] * weight[col FROM 0 FOR 3][0])
      column.sum[1] := SUM(framedZ[col FROM 256 FOR 3] * weight[col FROM 0 FOR 3][1])
      column.sum[2] := SUM(framedZ[col FROM 514 FOR 3] * weight[col FROM 0 FOR 3][2])
      result := SUM(column.sum[col FROM 0 FOR 3])
  END
```

4.5.4 DFT

```
%
% This example inputs a stream of 64 data samples and outputs two streams of
% 64 real values. One output stream is the magnitude spectrum, while the
% other output stream is the phase spectrum. The program terminates after 80
% streams have been inputted and outputted.
%
PROG dft( INPUT(INT32) data[64] OUTPUT(REAL32) magnitude[64], phase[64] )

  NODE smoothed.data[64], real[64], image[64]
  VALUE pi IS 3.141592654
  VALUE pi.by.2 IS pi * 2.0
  VALUE window[64] IS EXPRESSION
  VALUE w.cos[64][64] IS EXPRESSION
  VALUE w.sin[64][64] IS EXPRESSION

  BEGIN
    INIT
      window[col FROM 0 FOR 64] IS 0.5 + (COS(((REAL(col) / 64.0) - 0.5) * pi) / 2.0)
      w.cos[col FROM 0 FOR 64][row FROM 0 FOR 64] IS (1.0 / 64.0) * ...
        COS(((REAL(row) + 1.0) * (REAL(col) / 64.0)) * pi.by.2)
      w.sin[col FROM 0 FOR 64][row FROM 0 FOR 64] IS (1.0 / 64.0) * ...
        SIN(((REAL(row) + 1.0) * (REAL(col) / 64.0)) * pi.by.2)

    REPEAT FOR 80
      smoothed.data[col FROM 0 FOR 64] := data[col FROM 0 FOR 64] * ...
        window[col FROM 0 FOR 64]
      real[row FROM 0 FOR 64] := SUM (data[col FROM 0 FOR 64] * ...
        w.cos[row FROM 0 FOR 64][col FROM 0 FOR 64])
      image[row FROM 0 FOR 64] := SUM (data[col FROM 0 FOR 6 4] * ...
        w.sin[row FROM 0 FOR 64][col FROM 0 FOR 64])
      magnitude[col FROM 0 FOR 64] := SQRT((real[col FROM 0 FOR 64] ** 2) + ...
        (image[col FROM 0 FOR 64] ** 2))
      phase[col FROM 0 FOR 64] := ATAN(image[col FROM 0 FOR 64 ] / ...
        real[col FROM 0 FOR 64])
  END
```

4.5.5 Lattice filter

```
%
% This example is of a 7 stage lattice filter. The upper and lower data value
% at each stage are calculated by the functions top.section and bottom.section
% respectively. The sequential nature of the upper data path is reflected by
% the dependency successive top.sections have on their predecessors.
%
PROG lattice( INPUT(REAL32) signal OUTPUT(REAL32) upper, lower )

  NODE top[5], bottom[5]
  VALUE TABLE p[7] IS [0.4563, 0.4562, 0.8935, 0.2345, 0.9374, 0.3533, 0.7745]
  VALUE TABLE q[7] IS [0.3329, 0.7846, 0.7844, 0.9938, 0.6366, 0.3443, 0.2323]

  FUNCTION top.section(in.top, in.bottom, c)
    NODE n
    BEGIN
      n := in.bottom * c
    RESULT := n + in.top

  FUNCTION bottom.section(in.top, in.bottom, c)
    NODE n
    BEGIN
      n := in.top * c
    RESULT := n + in.bottom

  BEGIN
    REPEAT FOREVER
      top[0] := top.section(signal, signalZ[1], p[0])
      top[1] := top.section(top[0], bottom[0]Z[1], p[1])
      top[2] := top.section(top[1], bottom[1]Z[1], p[2])
      top[3] := top.section(top[2], bottom[2]Z[1], p[3])
      top[4] := top.section(top[3], bottom[3]Z[1], p[4])
      top[5] := top.section(top[4], bottom[4]Z[1], p[5])
      upper := top.section(top[5], bottom[5]Z[1], p[6])
      bottom[0] := bottom.section(signal, signalZ[1], q[0])
      bottom[1] := bottom.section(top[0], bottom[0]Z[1], q[1])
      bottom[2] := bottom.section(top[1], bottom[1]Z[1], q[2])
      bottom[3] := bottom.section(top[2], bottom[2]Z[1], q[3])
      bottom[4] := bottom.section(top[3], bottom[3]Z[1], q[4])
      bottom[5] := bottom.section(top[4], bottom[4]Z[1], q[5])
      lower := bottom.section(top[5], bottom[5]Z[1], q[6])
  END
```

4.5.6 Level-crossing detector

```
%
% This example transfers the input to the output whenever the absolute
% value of the input crosses the value set by level, for all other conditions
% the value of output is zero.
%
PROG level.crossing.detector( INPUT(REAL32) input, level OUTPUT(REAL32) output )

  NODE value

  BEGIN
    REPEAT FOREVER
      value : = ABS(in)
      output : = IF (value > = level)
        THEN (IF (valueZ[1] > = levelZ[1]) THEN 0.0 ELSE input)
        ELSE (IF (valueZ[1] > = levelZ[1]) THEN input ELSE 0.0)
  END
```

4.5.7 Matrix product

```
%
% This example performs a cross product of an [m] by [n] matrix with an [n] by [m] matrix,
% producing an [n] by [n] matrix.
%
PROG matrix.product( INPUT(INT32) p[10][20], q[20][10]
                    OUTPUT(INT32) r[10][10] )

  BEGIN
    REPEAT FOREVER
      r[i FROM 0 FOR 10][j FROM 0 FOR 10]  : = ...
        SUM(p[i FROM 0 FOR 10][k FROM 0 FOR 20] * ...
        q[k FROM 0 FOR 20][j FROM 0 FOR 10])
  END
```

4.6 Summary

This chapter has described the language DFDL and in doing so has detailed types, operators, functions, program structure, external input/output, nodes, constants, user-defined functions, Z-operator, assignment, expressions, subscripts and conditional constructs. These descriptions have been supplemented by some examples at the end of the chapter. A summary of DFDL syntax is given in Appendix B.

Chapter 5. DFDL task model

This chapter describes the translation of a discrete algorithm from a program to a graph and it describes the graph's data structure. The graph of the discrete algorithm is a directed acyclic graph (DAG), which represents the discrete algorithm without loss of function or structure.

The DAG is called the task graph, denoted **G**, which is given as **G** = (**T**, **C**, B, E, **A**). The set of nodes **T**, corresponds to the program's task primitives that are executable on processors and the set of directed arcs **A**, expresses sequential dependency between the tasks. Branching in the graph represents parallelism, rather than alternative paths of computation.

The chapter begins by describing the data structure of task primitives and how these data structures are connected together to form **G**. Following this, the different types of nodes are detailed and finally, the transformations are described, these fall into three categories; (i) named graph structures, (ii) primitive graph structures and (iii) non-primitive graph structures. All the data structures are written in Occam.

5.1 Data structures

5.1.1 Executable node data structure

The in-degree and out-degree of nodes in **T** and **C** are shown in Table 5.1. With the exception of the initiating and termination nodes, (B and E respectively), all other nodes have an in/out-degree less than or equal to two. This low in/out degree tends to reflect the low I/O complexity of the tasks, which is regarded as a prerequisite for a medium/fine grain graph structure.

Type	In-degree	Out-degree	Comments
B	0	multiple	initiates G
E	multiple	0	terminates G
Others	1	1	monadic input and output
	1	2	monadic input dyadic output
	2	1	dyadic input monadic output

Table 5.1 In/out-degree of task graph nodes

G is created in the form of a doubly linked list, where each node contains links to both its immediate successors and its immediate predecessors. This makes it possible to traverse the graph in either direction, such that any node may be reached from any other node in **G**. The flexibility given by doubly linking must be offset against the overheads of storage and time to traverse linked nodes. However, the majority of nodes in **G** have a low I/O degree which makes **G** sparse, hence nodes are compact and therefore storage overheads are low. Generally a doubly linked list produces a memory efficient data structure, whose size is proportional to $|\mathbf{T}| + |\mathbf{C}|$.

5.1.2 Non-executable node data structure

At the top and bottom of the graph **G** are the nodes B and E respectively. Entries for these two nodes have a different format to other nodes in **G**, because of their variable in/out-degree. B and E data structures each take the form of a vector and pointer (Figure 5.1). The entries in B hold all the addresses of nodes in **T** and **C** that are at the top of **G** and the entries in E hold all node addresses in **T** and **C** that connect to the bottom of **G**. The pointer (Figure 5.1) keeps track of the last entry in the vector.

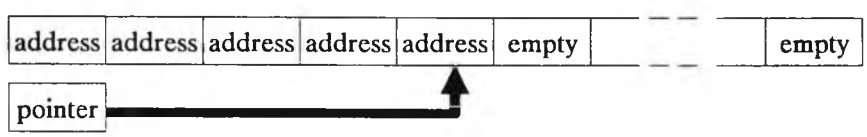


Figure 5.1 B and E data structure

5.1.3 Executable node attributes

Nodes other than B and E employ blocks of 5 contiguous words (20 bytes) each. A node comprises several attributes which are all stored at specific locations within the node. Most of these attributes are common to all nodes, however, where this is not the case, alternative, mutually exclusive attributes occupy the same location in different node types. Locations belonging to attributes which become redundant may be used by successive attributes. The format of these nodes is shown in Table 5.2. Note: the execution cost of a task type (node) is represented by the node's name, i.e. its TYPE.

word 4	word 3	word 2	word 1	word 0	Attributes
					TYPE
					COLOUR A/B
					NUMBER
					TO.FIRST
					EST
					LST
					FR.FIRST
					TO.SECOND
					FR.SECOND
					DELAY.VAL
					REAL.VAL
					LABEL
					INDEX
					TO.PROC
					LIST.PROC
					FR.PROC

Table 5.2 Node format

5.1.4 Attribute description

TYPE:	Defines the type of node and therefore its execution cost.
COLOUR A/B:	Control information.
NUMBER:	Node number.
TO.FIRST:	Address of first immediate successor.
TO.SECOND:	Address of second immediate successor (dyadic output only).
FR.FIRST:	Address of first immediate predecessor (not real or constant nodes).
FR.SECOND:	Address of second immediate predecessor (dyadic input only).
EST:	Earliest start time or scheduled start time (used during scheduling).
LST:	Latest start time.
DELAY.VAL:	Integer delay value (delay nodes only).
REAL.VAL:	Real value (real or constant nodes only).
LABEL:	Symbol table address of identifier (named nodes only).
INDEX:	Array index (named nodes only).
TO.PROC:	Processor connected to (used during scheduling).
FROM.PROC:	Processor connected from (used during scheduling).
LIST.PROC:	Resource type (used during scheduling).

5.2 Node types

Table 5.3 summarises the different types of task primitives T_i and C_q that may be used in **G**. The relative task execution cost category gives the worst-case costs for T_i, denoted e_i, for a T414-20 Transputer. Where different, T800-20 costs are shown in parentheses. Communication costs c_q for a task C_q are given for link speeds of 20Mbit s^{-1}, these are superscripted with an asterisk to distinguish them from processor costs. Transputers which operate at a different clock frequency, or communicate using a different link speed have their costs scaled accordingly. Arithmetic and trigonometrical operations are costed for 32 bit floating point operations. All costs are in units of microseconds.

Task primitive	Type attribute	In-degree	Out-degree	Relative execution /communication cost
EXT.IN	1	1	2	0.0
EXT.OUT	2	2	1	0.0
BYTE:REAL32	3	1	1	3.4 (0.65)
INT16:REAL32	4	1	1	3.8 (0.65)
INT32:REAL32	5	1	1	4.6 (0.60)
REAL32:BYTE	6	1	1	2.15 (0.90)
REAL32:INT16	7	1	1	2.15 (0.90)
REAL32:INT32	8	1	1	2.10 (0.85)
NODE	9	1	1	0.0
CONST	10	1	1	0.0
REAL	11	1	1	0.0
INT.IN	12	1	2	0.0
INT.OUT	13	2	1	0.0
COMM.IN	14	1	1	0.55
NEG	15	1	1	5.0 (0.7)

continued over ...

continuation ...

Task primitives	Type attributes	In-degree	Out-degree	Relative execution /communication cost
ADD	16	2	1	15.0 (0.35)
SUB	17	2	1	15.0 (0.35)
MULT	18	2	1	12.0 (0.65)
DIV	19	2	1	14.0 (0.95)
REM	20	2	1	16.0 (1.7)
DELAY	21	1	1	0.5
BRANCH	22	1	2	0.0
SGN	23	1	1	0.35
ABS	24	1	1	0.25
SQRT	25	1	1	24.0 (7.2)
LN	26	1	1	125.0 (28.0)
LOG	27	1	1	138.0 (31.4)
EXP	28	1	1	112.0 (36.7)
COMM.OUT	29	1	1	0.55
SIN	30	1	1	150.0 (33.8)
COS	31	1	1	160.0 (25.1)
TAN	32	1	1	143.0 (34.7)
ASIN	33	1	1	127.0 (26.0)
ACOS	34	1	1	117.0 (24.8)
ATAN	35	1	1	130.0 (25.2)
COMM.BYTE	36	1	1	0.55*
COMM.INT16	37	1	1	1.1*
COMM.INT32	38	1	1	2.2*
COMM.REAL32	39	1	1	2.2*
AND	40	2	1	0.10
OR	41	2	1	0.40
NOT	42	1	1	0.10
EQ	43	2	1	3.0 (0.45)
NEQ	44	2	1	3.0 (0.45)
LT	45	2	1	3.0 (0.50)
LT.EQ	46	2	1	3.0 (0.50)
GT	47	2	1	3.0 (0.50)
GT.EQ	48	2	1	3.0 (0.50)
GATE	49	2	1	0.65
PRI.OR	50	2	1	0.40

Table 5.3 Task primitives, in/out-degree and cost

5.3 Named graph structures

Named graph structures are generated in response to input, output, node and constant declarations. These structures are produced prior to the repetitive section of a program and begin with the DFDL keywords INPUT, OUTPUT, NODE and VALUE, respectively. Named graph structures, comprising named nodes, form the skeleton of **G**, from which all successive nodes are connected, either directly or indirectly. Each different type of named node has its connections configured according to its type, position in an array and, in the case of inputs and outputs, its data type. To simplify the description, all named nodes are taken to be 2 dimensional arrays, e.g., x[*col.size*][*row.size*]. For example, a scalar has a *col.size* and *row.size* equal to 1.

All named nodes have a label attribute, which is a pointer into the symbol table, directed at the first character of the node's identifier. These nodes also possess an index attribute, which identifies the node's subscript (i.e. array subscript). This index is held as a single integer, which is equal to i + (j * *col.size*); i is the column index and j the row index, 0 < = i < *col.size* , 0 < = j < *row.size* .

5.3.1 Input nodes

An external input graph structure is constructed in response to the declaration:

INPUT(*data.type*) *input [sub.size]* { 0 , *input [sub.size]* }

The simplest graph structure for an external input is for a data type of REAL32, since conversion from type non-REAL32 to REAL32 is unnecessary (all internal numeric operations are type REAL32 in the DFDL environment). Input arrays are configured as streams (Figure 5.2), such that any input u_i cannot precede any other input u_{i-1}, for all i, 0 < i < k. k is the total number of elements in an input stream, given as *col.size* * *row.size*. When a column, or row, size is not declared, a default value of size 1 is substituted.

An EXT.IN (external input) node has a monadic input and a dyadic output. The first element in any external input stream is u_0, whose single input is connected from B, the initiating node (Figure 5.2). This arc defines precedence between B and u_0, however, it does not imply the transfer of data. Precedence between successive EXT.IN nodes in the same stream is established by "daisychaining"; precedence is transferred through the node, from the single input out to the second output, to the succeeding EXT.IN node's single input. The final node in an input stream u_{k-1} terminates the stream and thereby leaves its second output unconnected.

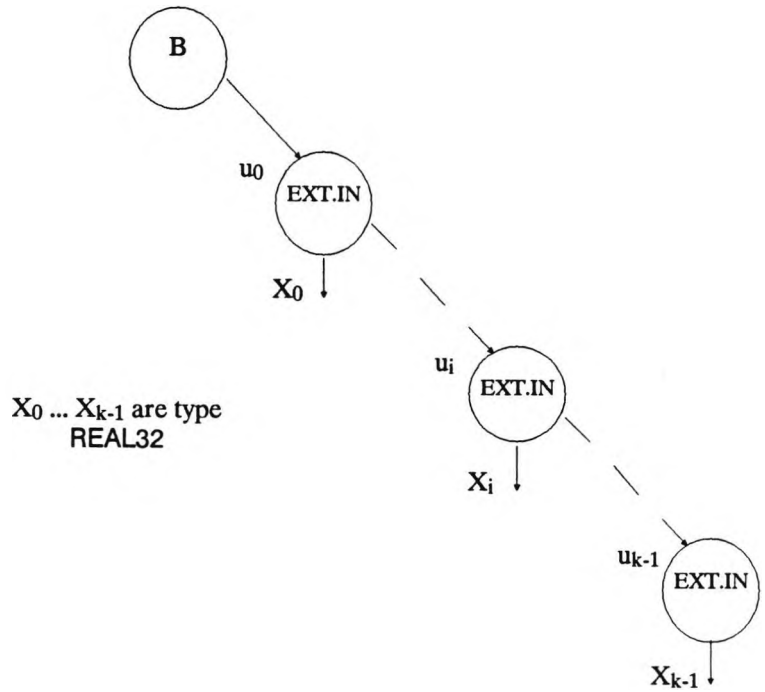


Figure 5.2 External input stream (REAL32)

A summary of the relevant attributes given to an EXT.IN node during the construction of **G** is shown Table 5.4.

External inputs which have a non-REAL32 data type, include conversion nodes in their graph structure. The type of conversion node depends on the data type of the external input. The three different types of input conversion node are BYTE:REAL32, INT16:REAL32 and INT32:REAL32. A conversion node has its input connected from the first output of an EXT.IN and has its output connected into **G**. Non-REAL32 external inputs have a graph structure as shown by Figure 5.3. For XXX, read BYTE, INT16 or INT32.

A summary of the relevant attributes assigned to input conversion nodes, BYTE:REAL32, INT16:REAL32 and INT32:REAL32, is shown in Table 5.5.

Attributes	
TYPE	EXT.IN
TO.FIRST	Data: if u_i is type non-REAL32, connects to convertion node
TO.SECOND	Sync: u_i connects to u_{i+1} for all i, $0 <= i < k-1$.
FR.FIRST	Sync: u_i connects from (i) B if $i = 0$, (ii) u_{i-1} for all i, $0 < i < k$.
FR.SECOND	
DELAY.VAL	
REAL.VAL	
LABEL	Address of identifier in symbol table
INDEX	Node u_i has index i, where $0 <= i < k$.

Table 5.4

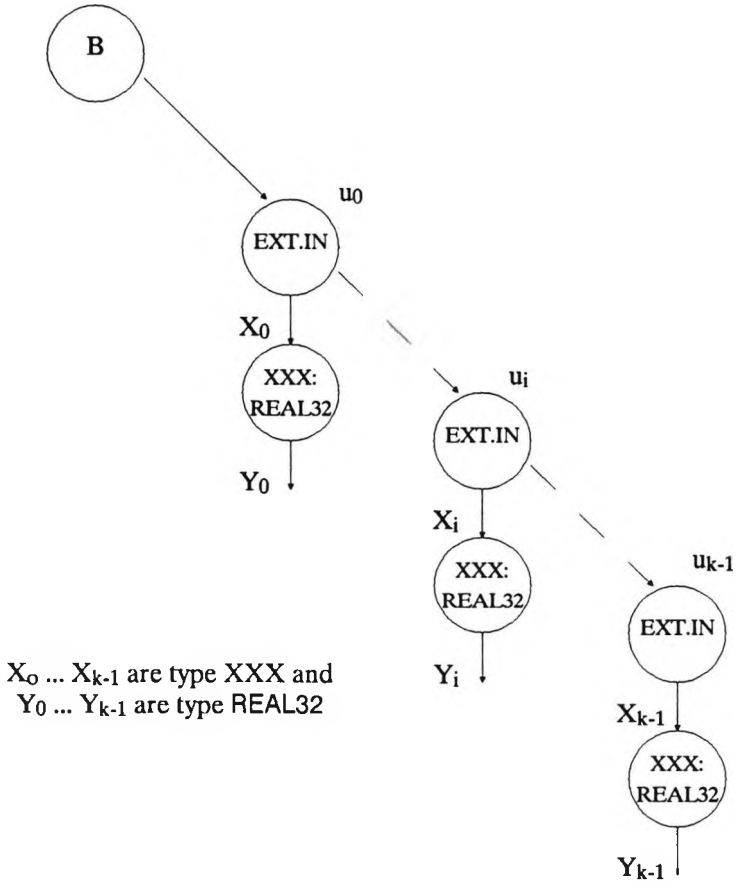


Figure 5.3 External input stream (non-REAL32)

Attributes	
TYPE	BYTE:REAL32, INT16:REAL32, INT32:REAL32
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Data: connects from external input
FR.SECOND	
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.5

5.3.2 Output nodes

An external output graph structure is constructed in response to the declaration:

OUTPUT(*data.type*) *output [sub.size]* { 0 , *output [sub.size]* }

Outputs are similar to inputs, in that the simplest output case is for a data type of REAL32, since conversion to REAL32 is unnecessary. Output arrays are configured as streams, such that any output v_i cannot precede any other output v_{i-1}, for all i, $0 < i < k$. k is the total number of elements in an output array, given as *col.size* * *row.size*. When the column or row size is not declared, a default value of size 1 is substituted.

An EXT.OUT (external output) node has a dyadic input and a monadic output. The last element in any external output stream is denoted v_{k-1}, whose single output is connected to E, the terminating node (Figure 5.4). Precedence between successive EXT.OUT nodes in the same stream, beginning with v_0, is established by "daisychaining"; precedence is transferred through the node, from the second input and out to the single output, to the succeeding EXT.OUT node's second input. The first node in an output stream v_0 initiates the stream and as such, leaves its second input unconnected.

A summary of the relevant attributes assigned to an EXT.OUT node during the construction of **G** is given by Table 5.6.

External outputs that have a non-REAL32 data type, include conversion nodes in their graph structure. The type of conversion node depends on the data type of the external output. The three different types of output conversion node are REAL32:BYTE, REAL32:INT16 and REAL32:INT32. A conversion node's output is connected to the first input of an EXT.OUT and its input is connected from **G**. Non-REAL32 external outputs have a graph structure as shown by Figure 5.5.

A summary of the relevant attributes assigned to output conversion nodes, REAL32:BYTE, REAL32:INT16 and REAL32:INT32, is given by Table 5.7.. For XXX, read BYTE, INT16 or INT32.

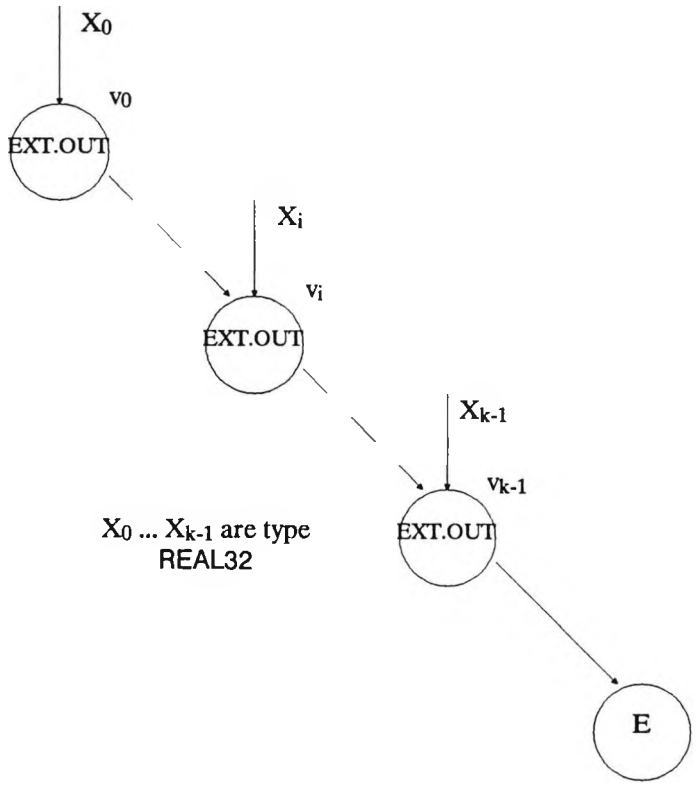


Figure 5.4 External output stream (REAL32)

Attributes	
TYPE	EXT.OUT
TO.FIRST	Sync: v_i connects to (i)E if $i = k-1$, (ii)v_{i-1}, for all i, $0 <= i < k-1$.
TO.SECOND	
FR.FIRST	Data: if v_i is type non-REAL32, connects from convertion node
FR.SECOND	Sync: v_i connects from v_{i-1} for all i, $0 < i <= k-1$.
DELAY.VAL	
REAL.VAL	
LABEL	Address of identifier in symbol table
INDEX	Node v_i has index i, where $0 <= i < k$.

Table 5.6

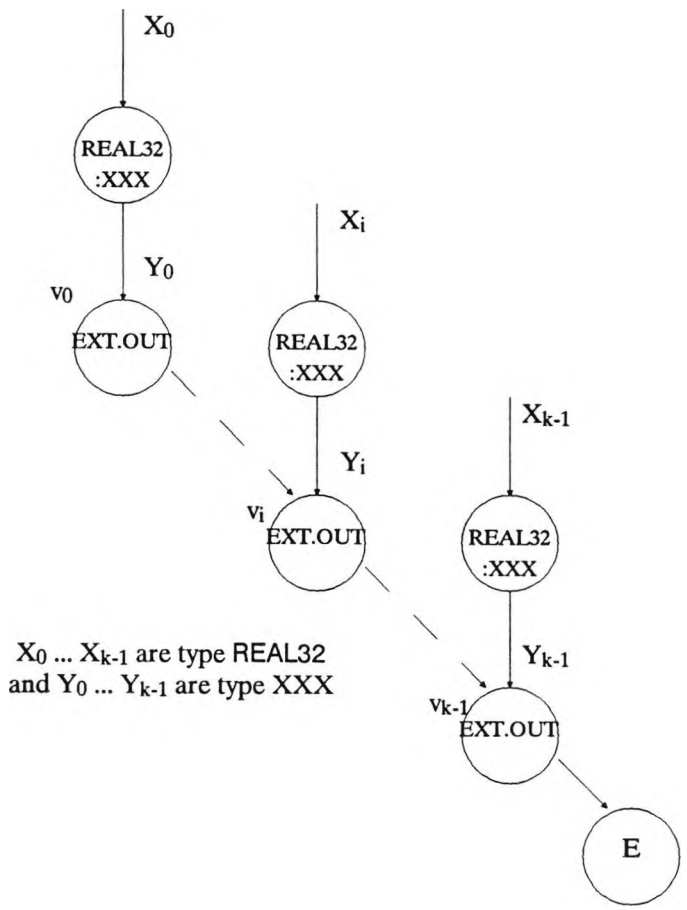


Figure 5.5 External output stream (non-REAL32)

Attributes	
TYPE	REAL32:BYTE, REAL32:INT16, REAL32:INT32
TO.FIRST	Data; connects to external output
TO.SECOND	
FR.FIRST	Data
FR.SECOND	
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.7

5.3.3 Node nodes

A node graph structure is constructed in response to the declaration:

NODE *node [sub.size]* { 0 , *node [sub.size]* }

Unlike input and output graph structures, node arrays are not configured as streams, but as independent elements (Figure 5.6). The number of elements in an array is k, where k is equal to *col.size* * *row.size*. The first element in an array is denoted n_0 and the last is denoted n_{k-1}. When a column size or row size is not declared, a default size of 1 is substituted.

A NODE (node) node has a monadic input and a monadic output. Figure 5.6 illustrates a node array, note that each element in the array is configured in an identical manner. Each node's single input transfers data from **G**, through the node, to the single output back to **G**. Data is unaffected by passing through a node.

A summary of the relevant attributes assigned to a NODE node during the construction of **G** is given by Table 5.8.

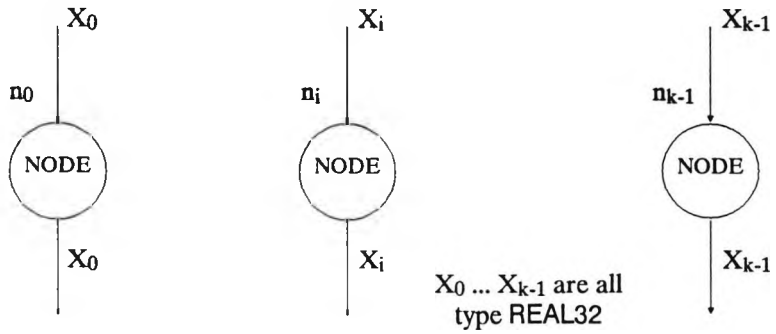


Figure 5.6 Node nodes

Attributes	
TYPE	NODE
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Data
FR.SECOND	
DELAY.VAL	
REAL.VAL	
LABEL	Address of identifier in symbol table
INDEX	Node n_i has index i, where $0 <= i < k$.

Table 5.8

5.3.4 Constant nodes

A constant graph structure is constructed in response to the declaration:

VALUE *constant [sub.size]* IS real or
VALUE TABLE *constant [sub.size]* IS *table* or
VALUE *constant [sub.size]* IS EXPRESSION

Constant nodes are independently configured, such that any constant c_i precedes or succeeds any other constant c_j, for all i not equal to j, $0 <= i < k$, $0 <= j < k$. k is the total number of elements in a constant array, which is given as *col.size* * *row.size*. When the column or row size is not declared a default value of size 1 is substituted.

A CONST (constant) node has a monadic input and a monadic output. The first element in a constant array is denoted c_0 and the last element is denoted c_{k-1}. For every constant node, the single input is connected from B[note 1], the initiating node (Figure 5.7). This defines precedence between B and c_i, for all i, $0 <= i < k$. However, this arc does not imply a transfer of data. The single output of each constant node derives from its real value, which connects into **G**.

A summary of the relevant attributes assigned to a CONST node during the construction of **G** is shown by Table 5.9.

note 1: The use of attribute REAL.VAL precludes the use of attribute FR.FIRST (see Table 5.2). Since all CONST nodes connect from B, the connection is implied.

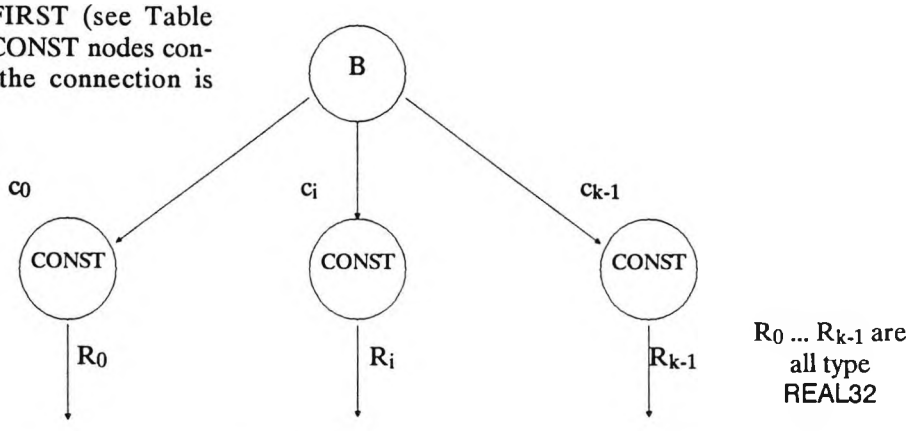


Figure 5.7 Constant nodes

Attributes	
TYPE	CONST
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Sync: c_i connects from B[note 1] for all i, $0 <= i < k$.
FR.SECOND	
DELAY.VAL	
REAL.VAL	Real value
LABEL	Address of identifier in symbol table
INDEX	Node c_i has index i, where $0 <= i < k$.

Table 5.9

5.4 Primitive graph structures

Primitive graph structures comprise the primitive nodes which equate to single or part DFDL program operations. The majority of primitive nodes, with the exception of named nodes and conversion nodes, are created during the repetitive section of a program. Non of the nodes described in this section are labelled or indexed, since they are not declared as "named structures".

5.4.1 Real nodes

Real laterals in expressions are translated to REAL nodes. A REAL node is identical to a CONST node in all its attributes, except that REAL nodes are not labelled or indexed.

A REAL has a monadic input and monadic output. The input always connects from B[note 2], the initiating node, and the output connects to **G**. The attribute REAL.VAL holds the real value of the node. A summary of the relevant attributes given to REAL is given by Table 5.10 and Figure 5.8 illustrates the connections for a REAL node in **G**.

note 2: The use of attribute REAL.VAL precludes the use of attribute FR.FIRST (see Table 5.2). Since all REAL nodes are connected from B, the connection is implied.

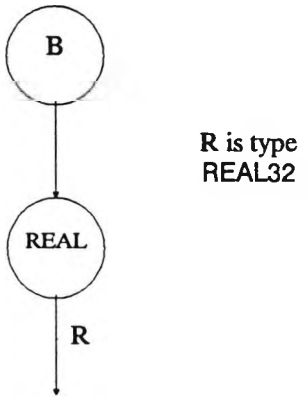


Figure 5.8 Real node

Attributes	
TYPE	REAL
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Sync: connects from B[note 2].
FR.SECOND	
DELAY.VAL	
REAL.VAL	Real value
LABEL	
INDEX	

Table 5.10

5.4.2 Internal input/output and delay nodes

Internal input nodes and internal output nodes are used in pairs (one of each type) to signify the transfer of data from one cycle to the next (a cycle is the period corresponding to the sampling interval). An internal input node is denoted INT.IN, it has a monadic input and dyadic output (Figure 5.9). The single input always connects from B, the initiating node. The first output of an INT.IN node is used to deliver a data value, that is passed to the node during the previous cycle. The internal output node, INT.OUT, has a dyadic input and monadic output (Figure 5.9). The single output always connects to E, the terminating node. The first input takes in a data value and passes it to the node's paired INT.IN node, ready for the next cycle.

A synchronising arc connects from the second output of INT.IN to the second input of INT.OUT, this prevents the INT.OUT node from passing data before the INT.IN node has completed, it also references INT.IN to INT.OUT and vice-versa. Figure 5.9 illustrates the connections between B, INT.IN, INT.OUT and E.

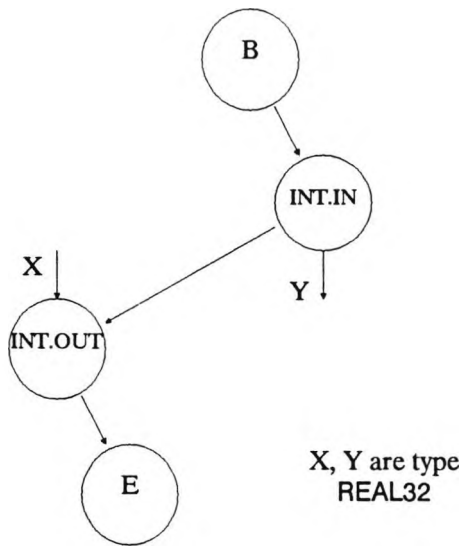


Figure 5.9 Single delay

The relationship between a value X which enters INT.OUT and appears as value Y in the following cycle at the INT.IN node is:

$$Y := XZ^{-1}$$

The single internal input/output pair form a delay of 1. Multiple cycle delays could be constructed by reproducing this graph structure d times, where d is the required delay. However, this would be somewhat expensive in terms of numbers of nodes, especially where large delays are concerned. A more compact method is to include a multiple delay node with each pair of INT.IN, INT.OUT nodes, whenever the delay exceeds 1. The delay node, DELAY, holds the attribute DELAY.VAL, which is given a positive integer value (d - 1), where d is the required delay.

Attributes	
TYPE	INT.OUT
TO.FIRST	Sync: connects to E
TO.SECOND	
FR.FIRST	Data
FR.SECOND	Sync: connects from INT.IN
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.11

Attributes	
TYPE	INT.IN
TO.FIRST	Data
TO.SECOND	Sync: connects to INT.OUT
FR.FIRST	Sync: connects from B.
FR.SECOND	
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.12

Figure 5.10 shows the DELAY node added to the graph structure. The relationship between values X, W and Y is as follows:

$W := XZ^{-(d-1)}$

$Y := WZ^{-1}$, hence,

$Y := XZ^{-d}$

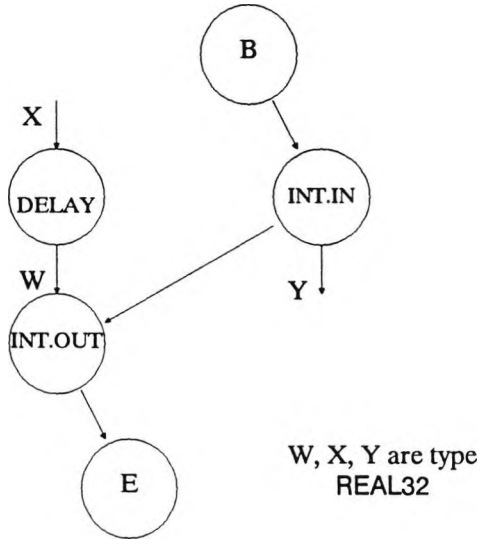


Figure 5.10 Multiple delay

Attributes	
TYPE	DELAY
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Data
FR.SECOND	
DELAY.VAL	Integer value
REAL.VAL	
LABEL	
INDEX	

Table 5.13

A summary of the relevant attributes for INT.IN and INT.OUT are given by Tables 5.11 and 5.12. The attributes for a DELAY node are given by Table 5.13. When the attribute DELAY.VAL evaluates to zero (i.e., d = 1), the DELAY node is omitted, since a delay of 1 is achieved using INT.IN and INT.OUT nodes only.

5.4.3 Arithmetic nodes

The monadic arithmetic operator - and the dyadic arithmetic operators +, -, *, /, \ are represented by a simple transformation to a single node per operator. The monadic operator transforms to node NEG, this has a monadic input and a monadic output. The dyadic operators transform to nodes ADD, SUB, MULT, DIV and REM respectively, where each node has a dyadic input and a monadic output.

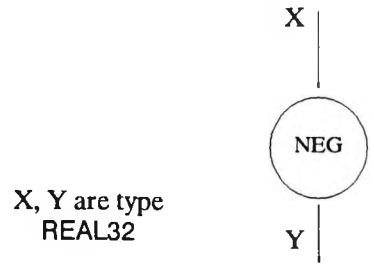


Figure 5.11 Monadic arithmetic node

Figure 5.11 shows the NEG node. For a NEG node, the output Y is equal to -X.

The dyadic nodes are illustrated by Figure 5.12. The relationships between their input and output values are:

ADD: Y := X + W
SUB: Y := X - W
MULT: Y := X * W
DIV: Y := X / W
REM: Y := X \ W

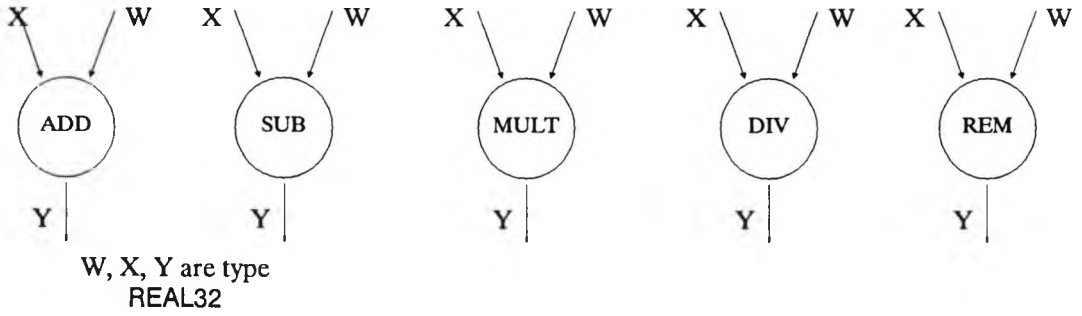


Figure 5.12 Dyadic arithmetic nodes

A summary of the relevant attributes assigned to arithmetic nodes during the construction of **G** is given by Table 5.14.

Attributes	
TYPE	POS, NEG, ADD, SUB, MULT, DIV, REM
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Data
FR.SECOND	Data: Dyadic nodes only.
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.14

5.4.4 Branch node

The branch node is used within **G** to distribute values, it achieves this by splitting a path into two, where both paths convey the same value. BRANCH has a monadic input and a dyadic output (Figure 5.13). The relationship between the input and output values can be expressed as:

$$Y_1 := X \text{ and } Y_2 := X.$$

A summary of the relevant attributes assigned to BRANCH nodes during the construction of **G** is given by Table 5.15.

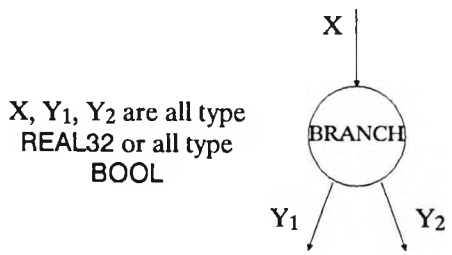


Figure 5.13 Branch node

Attributes	
TYPE	BRANCH
TO.FIRST	Data / boolean
TO.SECOND	Data / boolean
FR.FIRST	Data / boolean
FR.SECOND	
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.15

5.4.5 Functions

The majority of DFDL's monadic functions transform directly to single nodes. The name chosen for a particular node type is the same as its function name. The nodes are SGN, ABS, SQRT, LOG, LN, EXP, SIN, COS, TAN, ASIN, ACOS and ATAN. All nodes have a monadic input and a monadic output (Figure 5.14).

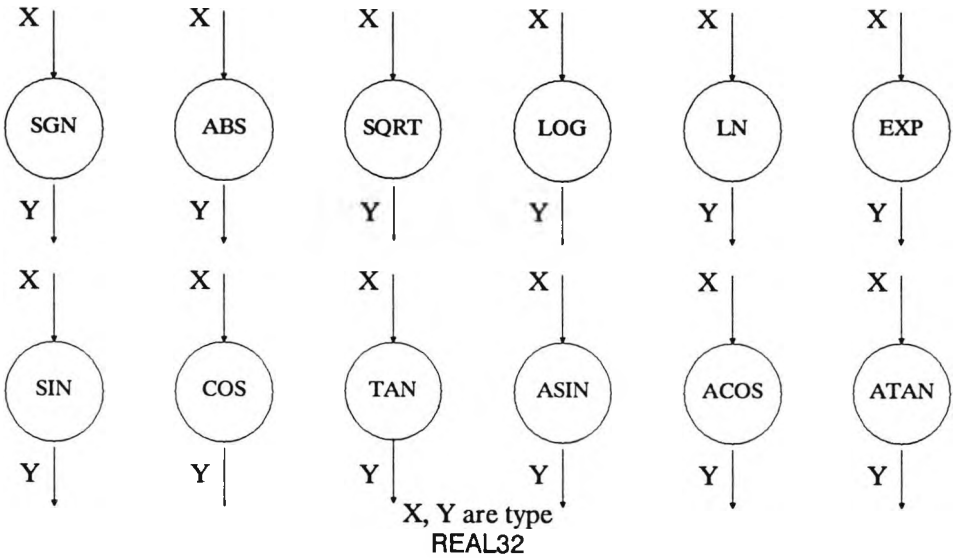


Figure 5.14 Function nodes

Attributes	
TYPE	SGN, ABS, SQRT, LOG, LN, EXP, SIN, COS, TAN, ASIN, ACOS, ATAN
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Data
FR.SECOND	
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.16

The relationship between the input X and the output Y is:

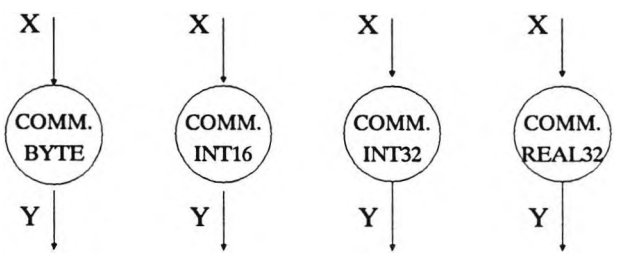
$$Y := \text{function } X$$

A summary of the relevant attributes for the functions SGN, ABS, SQRT, LOG, LN, EXP, SIN, COS, TAN, ASIN, ACOS and ATAN that are assigned during the construction of **G** are given in Table 5.16.

5.4.6 Communication nodes

Communication nodes represent communication effort, either between a processor and an external device or between different processors. Communication between adjoining processors is, in the case of DFDL, at word level (multiples of 4 bytes), because DFDL has an internal data type of REAL32. Communication to/from external devices, however, may be of 1, 2 or 4 bytes in length, depending on the declared data type of the external input or external output, i.e., BYTE, INT16 or INT32/REAL32. The cost attributed to a communications node includes the cost expended by the processor in setting up the communication and the cost expended by the link in transferring the data.

The four types of communication node are called COMM.BYTE, COMM.INT16, COMM.INT32 and COMM.REAL32. Each type has a monadic input and a monadic output (Figure 5.15).



X, Y are type XXX
where node is of type
COMM.XXX

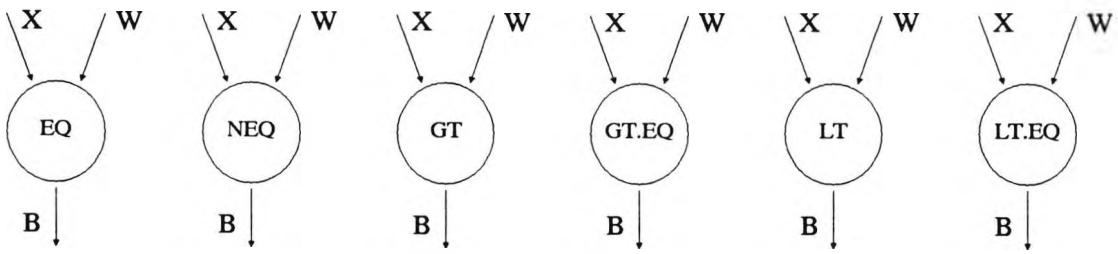
Figure 5.15 Communication nodes

There are no communication nodes in **G** during its initial construction, because communication cannot be established until the number of processors and the topology of the processor network are known. Consequently, communication nodes are added to set **C**, which is a member of **G**, during scheduling.

5.4.7 Relational nodes

The relational operators =, < >, >, > =, <, < = are exclusive to conditional expressions in DFDL programs. These six operators transform directly to single nodes, denoted EQ, NEQ, GT, GT.EQ, LT and LT.EQ, these stand for "equal to", "not equal to", "greater than", "greater than or equal to", "less than" and "less than or equal to" respectively.

All relational nodes have a dyadic input and a monadic output (Figure 5.16). The inputs are of type REAL32, while the output is type BOOL. The boolean output value (true or false) is a result of the relational function operating on the two data values. Relational nodes can only connect from nodes of type REAL32 and can only connect to nodes of type BOOL. The nodes are illustrated by Figure 5.16 and their attributes are given in Table 5.17.



W, X are type REAL32 and B is type BOOL

Figure 5.16 Relational nodes

Attributes	
TYPE	EQ, NEQ, GT, GT.EQ, LT, LT.EQ
TO.FIRST	Boolean
TO.SECOND	
FR.FIRST	Data
FR.SECOND	Data
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.17

5.4.8 Boolean nodes

Like relational operators, the boolean operators AND, OR, NOT are confined exclusively to conditional expressions in DFDL programs. These operators transform directly to single nodes and the nodes have identical names to their corresponding operators.

AND and OR nodes have a dyadic input while the NOT node has a monadic input, all boolean nodes have a monadic output (Figure 5.17). The inputs and outputs of all boolean nodes are of type BOOL.

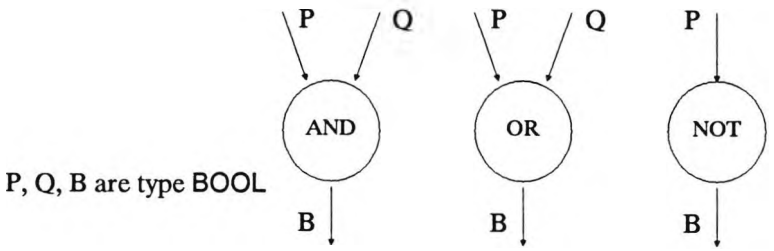


Figure 5.17 Boolean nodes

Attributes	
TYPE	AND, OR, NOT
TO.FIRST	Boolean
TO.SECOND	
FR.FIRST	Boolean
FR.SECOND	Boolean: Dyadic boolean nodes only
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.18

A summary of the attributes assigned to boolean nodes during the construction of G is given in Table 5.18.

5.4.9 Conditional nodes

At the heart of DFDL conditional expressions (see section 5.5.3) are two conditional nodes, these are called GATE and PRI.OR. These nodes both have a dyadic input and a monadic output (Figure 5.18).

A GATE node has a single data input and a single data output, of which both are of type REAL32. Its second input is of type BOOL and is used to control the passage of data through the node. When this input (second input) is false the output of the node is an o/c (open circuit) value, otherwise, when the boolean input is true the output is equal to the node's data input. The operation of a GATE node is described in Table 5.19.

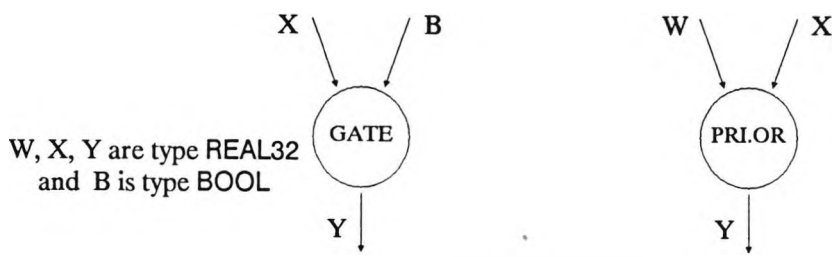


Figure 5.18 Conditional nodes

GATE		
input 1	input 2	output
X	false	o/c
X	true	X

Table 5.19

PRI.OR		
input 1	input 2	output
o/c	o/c	o/c
o/c	X	X
W	o/c	W
W	X	W

Table 5.20

A PRI.OR node's inputs and output are all of type REAL32. The node will always output the value at its first input, except when the data at that input is an o/c value and the data at the second input is not an o/c value, whereupon it will output the value at its second input. The operation of this node is described in Table 5.20.

An o/c value is a value which cannot be interpreted as a valid REAL32 value and is used to indicated that the value is "disconnected". The attributes for the two types of node are given in Tables 5.21 and 5.22.

Attributes	
TYPE	GATE
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Data
FR.SECOND	Boolean
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.21

Attributes	
TYPE	PRI.OR
TO.FIRST	Data
TO.SECOND	
FR.FIRST	Data
FR.SECOND	Data
DELAY.VAL	
REAL.VAL	
LABEL	
INDEX	

Table 5.22

5.5 Non-primitive graph structures

In DFDL the symbols **, ALOG, SUM, PROD, MEAN, IF...THEN...ELSE, MAX, MIN and MED transform to non-primitive graph structures. All these graph structures comprise primitive nodes which have been described in the previous section.

5.5.1 Power and anti-logarithm graph structures

Power graph structures are created in response to the operator **, where:

$$Y := X ** P$$

The operands X and P are the argument and power respectively and Y is the result of X^P. DFDL restricts X to non-negative values, since negative arguments of non-integer value produce complex (i.e. real + imaginary) results which are not catered for in DFDL.

To give a reasonably constant execution cost, the graph structure for power is fixed for all valid arguments and powers. The graph structure consists of a LN, a MULT and an EXP node (Figure 5.19). The graph structure evaluates X^P in three steps:

(i) evaluate ln (i.e. $\log_e$) of X
(ii) multiply result of (i) by P
(iii) evaluate e raised to the power of result of (ii), i.e. $e^{(P * \ln X)}$

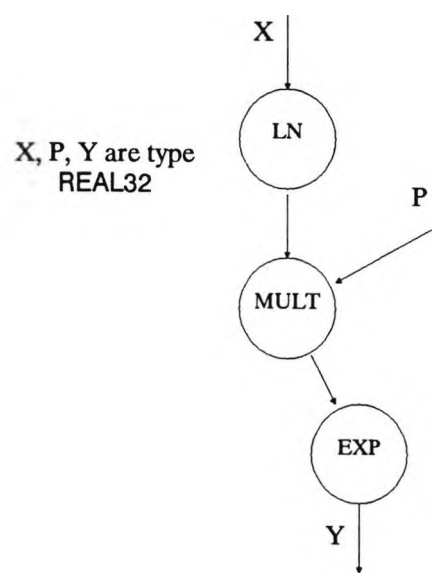


Figure 5.19 Power graph structure

Anti-log graph structures are created in response to the function ALOG, where:

$$Y := \text{ALOG } X$$

The expression ALOG X is equivalent to 10^X.

The graph structure for ALOG is similar to that of power. It consists of a REAL (value equal to ln10), a MULT and an EXP node (Figure 5.20). The graph structure evaluates 10^X in two steps:

(i) multiply X by ln10 (2.30585093)
(ii) evaluate e raised to the power of result of (i), i.e. $e^{(\ln 10 * X)}$

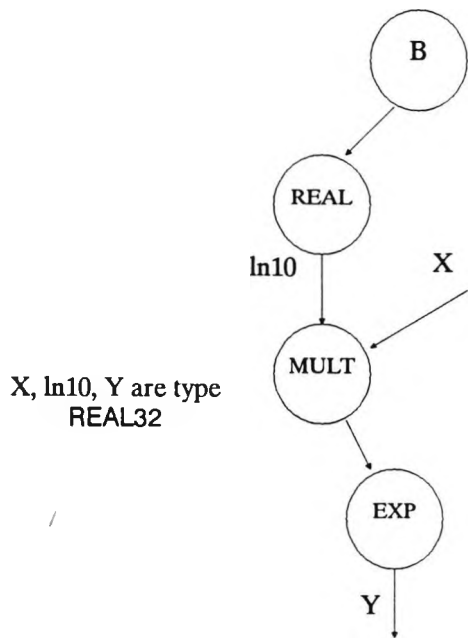


Figure 5.20 Anti-log graph structure

5.5.2 Sum, product and mean graph structures

The functions SUM, PROD and MEAN have k arguments (k > = 2) and a single result. Each of the functions consists of nodes arranged in a binary tree structure. Such a structure is maximally parallel and is characterised as having a longest path proportional to L, where L is defined as:

$$L = \lceil \log_2 k \rceil$$

Figure 5.21 illustrates the three graph structures; (a) SUM, (b) PROD and (c) MEAN. SUM consists solely of ADD nodes, PROD consists solely of MULT nodes and MEAN is a SUM graph, whose output is divided by k.

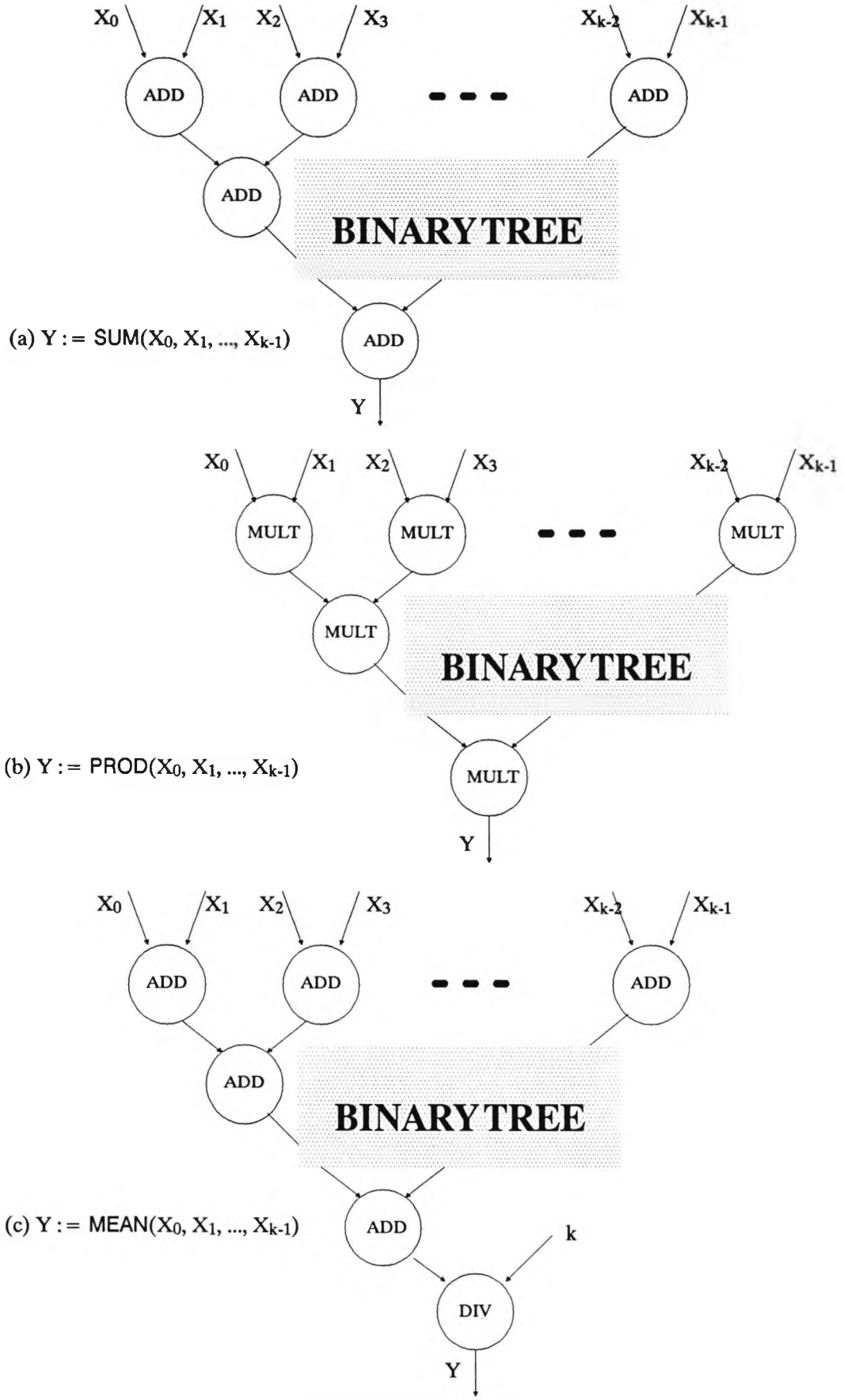


Figure 5.21 (a)SUM (b)PROD (c)MEAN graph structures

5.5.3 Conditional graph structures

A DFDL conditional expression transforms to a conditional graph structure, which typically comprises conditional, boolean and relational nodes. For example, the conditional statement below transforms to the structure shown in Figure 5.22.

```
Y := IF (P > Q) AND (R = S) THEN X ELSE W
```

A DFDL conditional expression is structured in the form of a tree, where the root node is the result and the leaves of the tree are expressions.. Expressions are combined using relational nodes, which in turn may be combined using boolean nodes. Conditional nodes evaluate all boolean conditions (true/false) and route the expression, that is pertaining to the first (in terms of hierarchy) true condition, to the root.

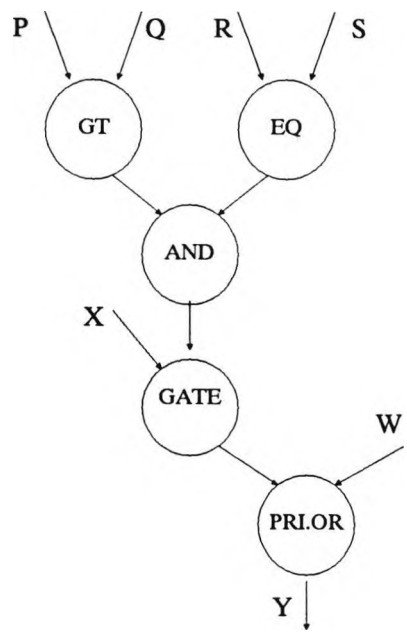


Figure 5.22 Conditional graph structure

This form of conditional evaluation is a data flow method and is unlike the conventional control flow method. It has the apparent disadvantage that all conditions are evaluated, irrespective of higher priority conditions. In comparison, a control flow scheme would evaluate each condition in turn, beginning with the highest priority, and would disregard any remaining conditions once a condition evaluated to true. The total amount of computational effort for both data flow and control flow is equal when all but the lowest priority condition are false, i.e. worst case complexity.

We find that the variable complexity of control flow has no advantage in a deterministic process, since the process must always be able to accommodate the worst case complexity. In fact, a control flow structure has a considerable disadvantage, because it is inherently sequential, whereas a data flow structure may contain parallelism.

5.5.4 Maximum and minimum graph structures

The DFDL functions MAX and MIN produce a single result which is the maximum, or minimum respectively, of the arguments to the function. The number of arguments is two or more and their type is REAL32 . The basic building blocks, MAX and MIN, are illustrated by Figures 5.23 and 5.24 respectively. More than one building block may be combined in a tree structure wherever the number of arguments exceeds two.

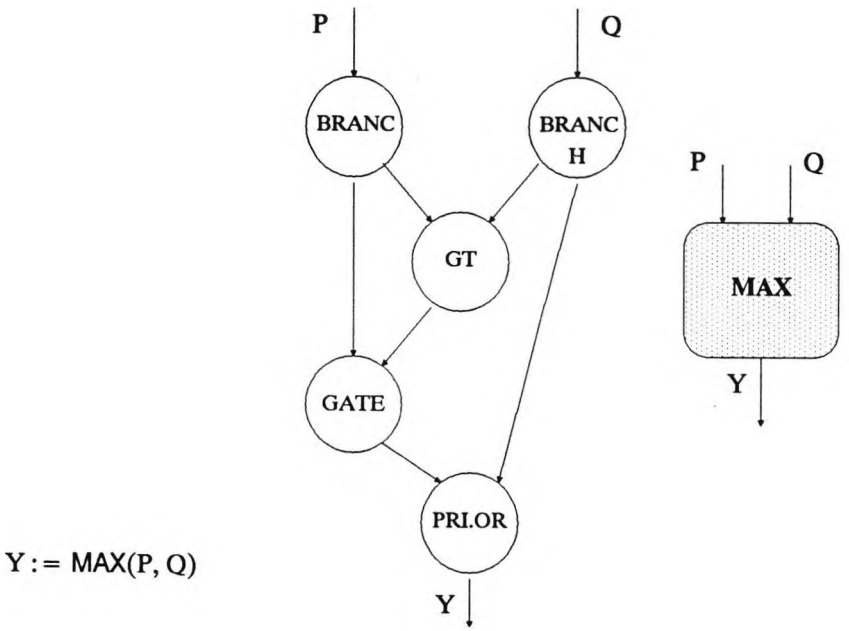


Figure 5.23 Maximum graph structure

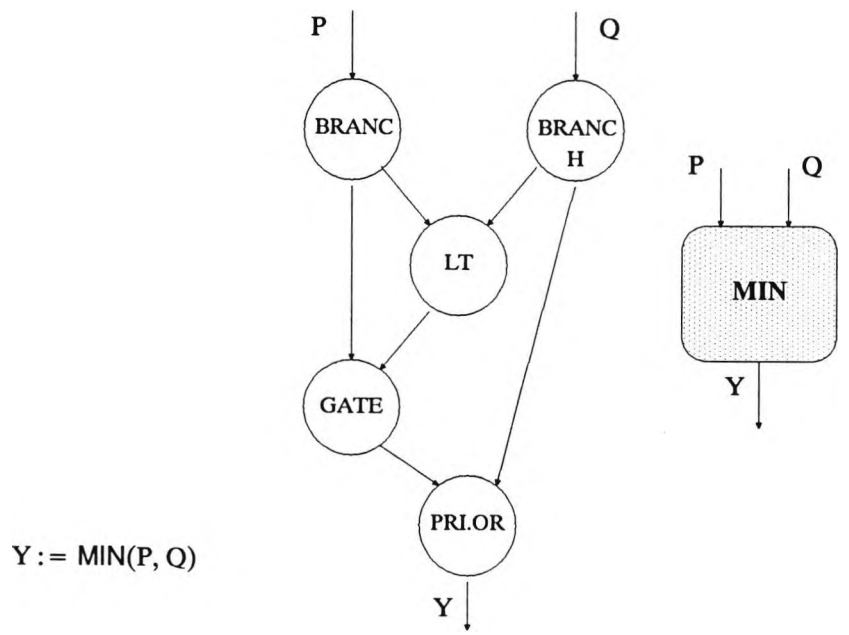


Figure 5.24 Minimum graph structure

5.5.5 Median graph structure

The final DFDL function, MED has two or more arguments of type REAL32 and produces a single result. It is transformed into a graph structure which comprises a number of MAX and MIN basic building blocks (Section 5.5.4). Figure 5.25 shows a median structure for the median of three arguments. Median is structured so that the median of even numbers of arguments is the lower median, such that MED(P, Q) is equal to MIN(P, Q).

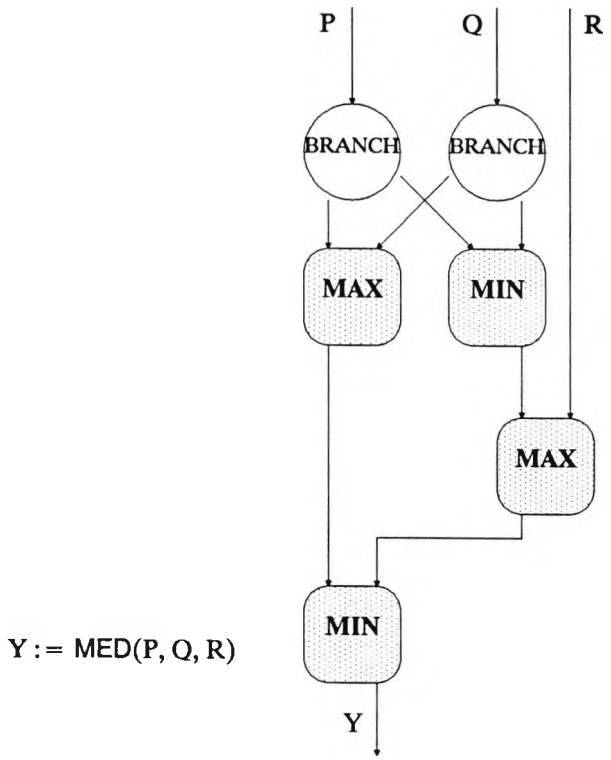


Figure 5.25 Median graph structure

5.6 Summary

In this chapter the data structures of those nodes comprising **G** have been shown. The low in/out-degree of nodes (except B and E) produces a sparsely connected DAG which can be realised using a doubly linked list, whose length is proportional to $|\mathbf{T}| + |\mathbf{C}|$.

The different types of nodes (task primitives) have been described in terms of their in/out-degree, their worst case execution cost and their function (i.e. task primitive name). These nodes are used to create a connected graph structure (the DAG), as directed by a DFDL program.

The transformation from program to graph has been described for three different categories of graph structure. The first includes external input/output, node and constant graph structures. The nodes which comprise these structures are classified as named nodes, because they correspond to named elements in the program and form the skeleton of the DAG. The second category includes all graph structures that are primitive (simple nodes), which reflect simple or part transformations. Finally, more complex graph structures were shown, each using several primitive nodes.

Chapter 6. Parallel processor model

This chapter presents an abstraction of the object machine's functional architecture, which is the parallel processor model used for scheduling parallel programs. A parallel processor is modelled as a static network of connected processing elements, where each processing element is identical and communication is achieved by message passing. The only processor element interactions considered by the model are data communication (includes synchronisation) and I/O. Other interactions, such as code distribution and memory-code allocation are ignored.

The Transputer microcomputer (INMOS, 1986) forms the basis of our object machine, however, the model is not restricted to this processor alone. Other types of processor that can form the basis of a loosely coupled static architecture are equally applicable.

The chapter begins by introducing the Transputer microcomputer and identifies the different resource types within the Transputer that are crucial to performance. It continues by describing how these resource types are represented by a data structure. The chapter concludes by defining the rules which govern the topology of the object processor's architecture and describes an algorithm that is used to check for some types of topological error.

6.1 The Transputer

The Transputer is a single chip microcomputer (von Neumann architecture), which incorporates several bi-directional serial communication channels (known as links). Each link enables point to point communication between two Transputers, or a Transputer and an I/O device (INMOS link adapter). Figure 6.1 shows a functional diagram for an INMOS Transputer.

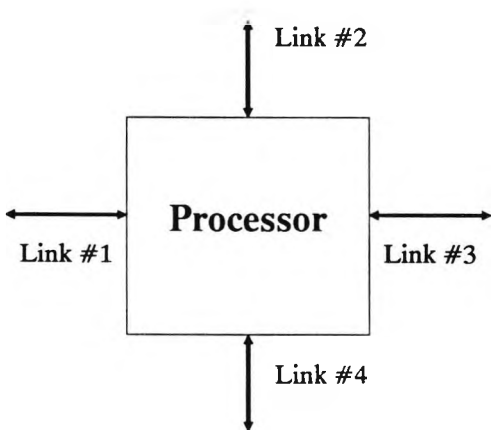


Figure 6.1 INMOS Transputer

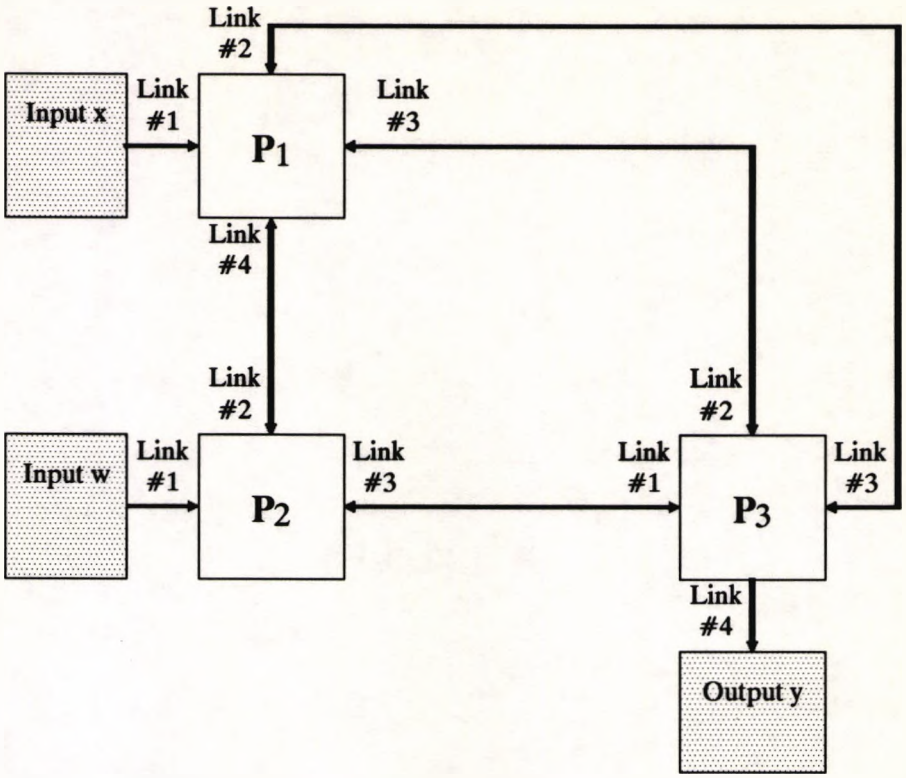


Figure 6.2 Transputer interconnection network

Static, multiple Transputer networks, such as the one shown in Figure 6.2, may be constructed using several Transputers. This form of parallel von Neumann architecture has the advantage of being completely scalable. This means the size of the architecture can be tailored to suit the processing problem. Its main advantage, however, is its MIMD architecture, which is capable of performing simultaneous multiple instruction processing and simultaneous multiple communication. This class of architecture facilitates the parallel execution of irregular, as well as regular structured processing problems.

6.2 Resources

A resource is defined as an item that supplies a need. The needs of parallel processing are processing and communication and the types of resource are processing elements and links. Both these resource types constitute items that are crucial to the performance of a parallel processor.

A limitation in the quantity or performance of either type of resource will usually prevent a system from realising its full potential. Practically speaking, performance will always be limited by one resource type or the other. Ideally though, a good parallel processor implementation should aim to balance resources, so that their limitations converge together.

Each Transputer consists of a processing resource, which will be called a processing element. The four unconnected links have to be connected to other unconnected links, from other Transputers or I/O devices, before they constitute usable resources. Once made, a link connects two devices to facilitate the flow of data and is regarded as a single resource that is shared between those two devices. The link connections in a Transputer network illustrate the topology of that network.

6.3 Activity

Resource activity is usually visualised with the aid of a Gantt chart (Clark, 1952); a diagram that displays activity in a binary sense (0 or 1) as a function of execution time. For example, Figure 6.3 shows a Gantt chart for a signal that is transmitted to processing element P1 (Figure 6.2) via link #1. The processing element is inactive until the signal arrives, whereupon it processes the signal and transmits the result to P3 via link #3. This two state view of activity is useful when considering resource use, i.e. a resource is either active (in use) or inactive (available for use).

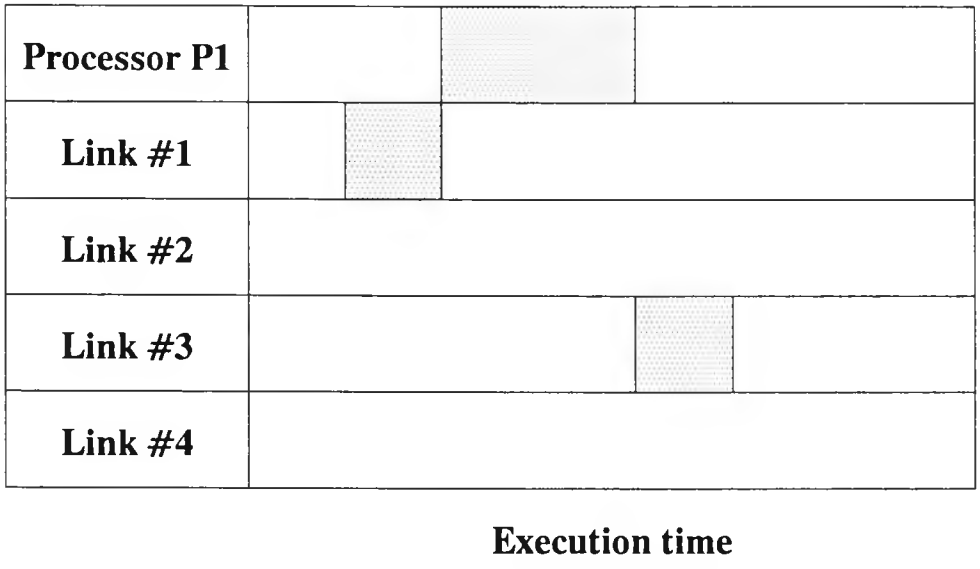


Figure 6.3 Gantt chart

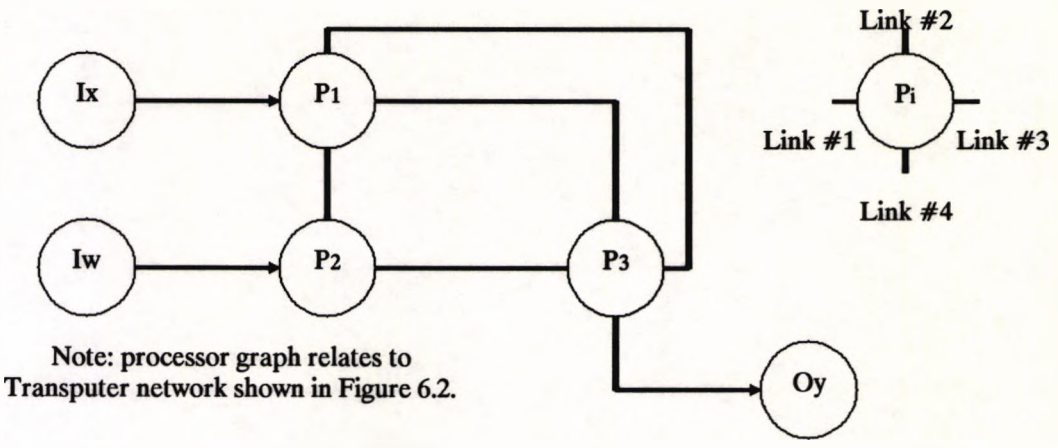


Figure 6.4 Processor graph of Transputer network

6.4 Data structures

The parallel processor model is divided into two parts and each part has its own data structure. The first part is called the processor graph, **V = (P, I, O, L)** (Figure 6.4) and its purpose is to represent the topology of a Transputer network in terms of its resources. The second part is called the schedule list, $S = \{S_1,..., S_q\}$. It contains activity schedules, one for each resource given in the processor graph.

6.4.1 Processor graph data structure

The top row of the data structure (Figure 6.5) identifies the categories as belonging to specific processing elements. Each category in the upper half of the data structure defines "to/from what" links are connected. Entries in the data structure that define link connections are either (i) a processor/link identification, (ii) an input identification, (iii) an output identification, or (iv) empty.

Empty, indicates a link is unconnected. All other entries consist of two values. This allows each link in the network to be uniquely identified by using its originating processor identification and its link number. Inputs and outputs include I/O classifiers as well as unique identification labels that originate from the symbol table.

The lower half of the data structure consists of "top of schedule" and "bottom of schedule" pointers which reference the activity schedules to their resources. These pointers are initially set equal to zero, to indicate that activity schedules are empty. Individual pointers are amended as and when entries are made to their activity schedule.

The current data structure can accommodate up to 20 inputs, 20 outputs and 99 processors. Figure 6.4 shows the processor graph of the example Transputer network (Figure 6.2), its data structure is illustrated (in a simplified form) by Figure 6.5.

Processor id	P_1	P_2	P_3
Link #1	Input	Input	P2
	x	w	Link #3
Link #2	P3	P1	P1
	Link #3	Link #4	Link #3
Link #3	P3	P3	P1
	Link #2	Link #1	Link #2
Link #4	P2	empty	Output
	Link #2		y
Processor list	top of schedule	top of schedule	top of schedule
	bottom of schedule	bottom of schedule	bottom of schedule
Link #1 list ptrs	top of schedule	top of schedule	top of schedule
	bottom of schedule	bottom of schedule	bottom of schedule
Link #2 list ptrs	top of schedule	top of schedule	top of schedule
	bottom of schedule	bottom of schedule	bottom of schedule
Link #3 list ptrs	top of schedule	top of schedule	top of schedule
	bottom of schedule	bottom of schedule	bottom of schedule
Link #4 list ptrs	top of schedule	top of schedule	top of schedule
	bottom of schedule	bottom of schedule	bottom of schedule

Figure 6.5 Processor graph data structure

6.4.2 Activity schedules data structure

The schedule data structure is accessed via the "top of schedule" or "bottom of schedule" pointers (section 6.4.1). These pointers mark the beginning and end of the activity schedule to which they belong. Activity schedules are empty prior to scheduling and are filled during the scheduling process (Chapter 7 - Compile-time scheduling). Each schedule is a list of operations (tasks), which will eventually be translated into an executable sequence. Entries to the activity schedules take the form of addresses which are the locations of nodes within the task graph, **G**. Each address represents a particular task, whose start time and completion time are found by examining the respective node in the task graph, **G**.

The data structure consists of doubly linked records. One or more of these records are used to form a single activity schedule. Each record holds up to 50 consecutive addresses to the task graph, plus link pointers to the preceding and succeeding records, where appropriate. Successive records need not be arranged in a contiguous fashion, since they are linked.

This form of data structure is chosen for reasons of flexibility and efficiency. It has the advantage of providing dynamic storage allocation which allows activity schedules to be extended on demand. Figure 6.6 illustrates how two separate schedules (denoted S1 and S2) may be stored within the schedule data structure.

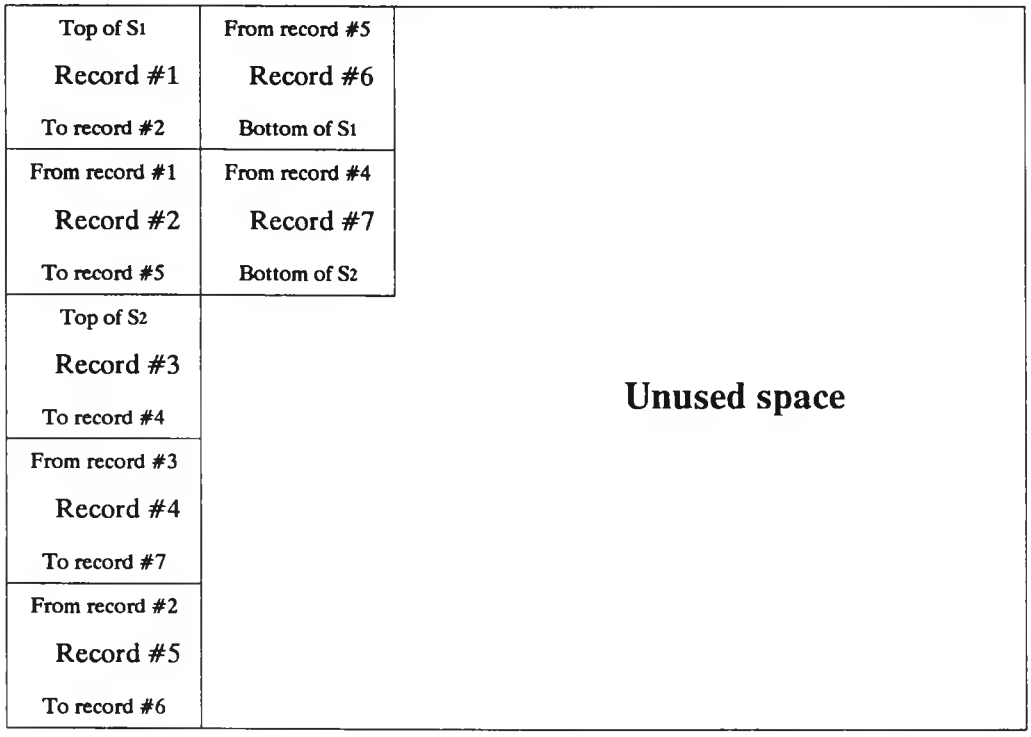


Figure 6.6 Schedule data structure

6.5 Processor graph

The processor graph **V** is defined as comprising four sets, **V** = (**P**, **I**, **O**, **L**), which are;

(i) a non-empty but finite set of nodes **P** (processors),

(ii) a finite set of nodes **I** (input ports),

(iii) a non-empty but finite set of nodes **O** (output ports) and

(iv) a set of arcs **L** (links).

The cardinality of the four sets is denoted |**P**|, |**I**|, |**O**| and |**L**|. The elements belonging to the three sets of nodes, **P**, **I**, **O**, are denoted Pi, Ij, Ok respectively.

6.5.1 Degree

The maximum number of arcs that can be connected to any node is called its degree, denoted D( ). The degree of a node corresponds to the number of links that can be connected to the element represented by the node. Nodes belonging to **P** have a degree of four (i.e. four links), while those nodes belonging to **I** and **O** have a degree of one (i.e. one link). Table 6.1 summarises the degree for the different node types.

Node type	**Pi**	**Ij**	**Ok**
Degree, D( )	**4**	**1 (out only)**	**1 (in only)**

Table 6.1 Degree of node types

6.5.2 Arc relationships

Arcs are denoted by the two nodes that they join. Where the degree of a particular node is greater than one, the individual connection must also be identified in order that arcs are unique and ambiguity is avoided. For example, an arc connecting two nodes Pi and Pj would be denoted (P_{ia},P_{jb}), where the letters a and b are positive integers in the range 1, ..., D(P_k).

First \ Second	**Pi**	**Ij**	**Ok**
Pp	$(P_{pa},P_{ib}) \longrightarrow (P_{ib},P_{pa})$ i not equal to p		(P_{pa},O_k)
Iq	(I_q,P_{ib})		
Or			

Table 6.2 Valid processor graph arcs

When an arc joins two elements in **P**, the immediate relationship is *irreflexive* and *symmetric*, hence (P_{ia},P_{jb}) implies (P_{jb},P_{ia}) provided i is not equal to j. Arcs containing elements from one of the two sets **I** or **O**, have to include an element from the set **P**, otherwise they are invalid (Table 6.1). The immediate relationship governing these arcs is *irreflexive* and *asymmetric*. Consequently, a valid arc (P_{ia},Oj) precludes the arc (Oj,P_{ia}).

In addition to the asymmetric nature of I/O arcs, a constraint on the direction of flow prevents the existence of arcs that represent flow in the wrong direction, e.g. inputting from an output port or outputting to an input port. Table 6.1 summarises the ordering for valid arcs in the processor graph, **V**.

6.5.3 Connectivity

The graph **V** is only valid for scheduling once it is *connected*. Connectivity is defined as when

- all nodes in **P** and **O** can be reached from all nodes in **I**, provided **I** is non-empty. Otherwise, if **I** is empty, all nodes in **P** and **O** can be reached from all nodes in **P**.

Once **V** is connected the set of arcs **L** is non-empty, since the sets **P** and **O** are defined as non-empty. The conditions for the minimum realisation of a connected processor graph are as follows:

$$|\mathbf{P}| = 1$$
$$|\mathbf{I}| = 0$$
$$|\mathbf{O}| = 1$$
$$|\mathbf{L}| = 1.$$

These minimum conditions for **V** comply with the minimum definition for the DFDL structure; i.e. zero or more inputs, one or more outputs and a processing system.

The maximum cardinality of set **L** for a connected processor graph is a function of the cardinality of sets **I**, **O**, **P** and the degree of their nodes, denoted D( ). The maximum cardinality of **L** is given by:

$$|\mathbf{L}|_{max} = 0.5 * ((D(\mathbf{I}) * |\mathbf{I}|) + (D(\mathbf{O}) * |\mathbf{O}|) + (D(\mathbf{P}) * |\mathbf{P}|))$$

The number of arcs belonging to **L** for a connected graph is finite, since **I**, **O**, **P**, D(**I**), D(**O**), D(**P**) are all defined as being finite. Hence, for a connected processor graph **L** is non-empty and finite.

6.5.4 Connecting the nets in V

A net is a connected graph or sub-graph which consists of one or more nodes. Before any arc is made **L** is empty and there are a total of $|\mathbf{P}| + |\mathbf{I}| + |\mathbf{O}|$ disjoint nets. During the connection process, arcs are formed and the number of nets is reduced. Connection of nodes continues until the graph is connected and there remains a single net comprising all elements in **P, I, O.**

When an arc is formed between two nets, or within a net, there is a possibility that nets may be isolated from one another. Isolation occurs whenever two or more nets can no longer join together, because one or more of the nets has an insufficient number of connections (unconnected links). Isolated nets stop **V** from being connected, consequently precautions have to be taken to prevent isolation.

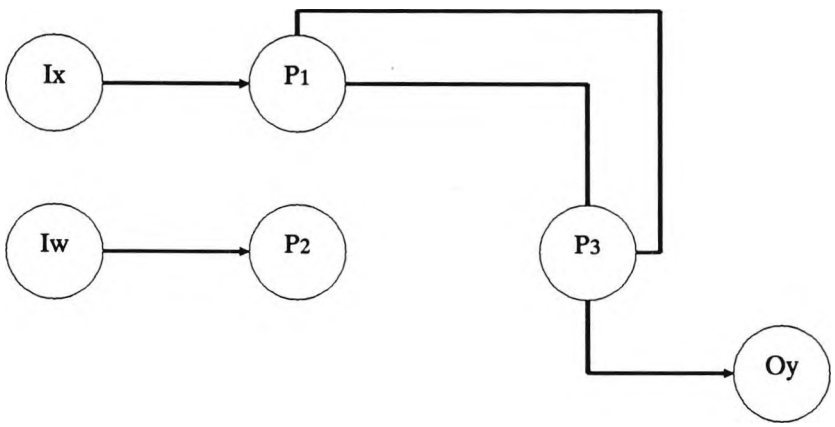


Figure 6.7 Partially connected processor graph

Consider the example (Figure 6.7) of a partially connected processor graph, **V**. If a further arc were formed between P1 and P3, then the net consisting of P2 and input Iw would be isolated. Attempts to join P2 and Iw to the larger net (comprising P1, P3, Ix and Oy) would not succeed, because no arc could be made between the two nets. Consequently, the processor graph could not be connected. A method of preventing net isolation is presented in the following section.

6.6 Checking for net isolation

To substantiate an arc and prevent the formation of isolated nets, there is a three stage check performed on the processor graph. The three stages are as follows:

- Step 1. All nodes belonging to sets **P**, **I** and **O** are grouped into $|\mathbf{N}|$ disjoint nets, $\mathbf{N} = \{N_1, N_2, \text{etc.}\}$. The members of **N** are defined as disjoint sets of nodes that form connected sub-graphs. In the example (Figure 6.7) there are two non-empty nets, N_1 and N_2. Before the arc (P_{14},P_{31}) is made, $N_1 = \{I_x, P_1, P_3, O_y\}$ and $N_2 = \{I_w, P_2\}$, after the arc is made the result is the same.

- Step 2. The number of unconnected links belonging to each net is counted. For a net N_i, the number of unconnected links belonging to N_i is $U(N_i)$. In the example (Figure 6.7) $U(N_1) = 2$, $U(N_2) = 3$ before the arc (P_{14},P_{31}) is made and $U(N_1) = 0$, $U(N_2) = 3$ after the arc is made.

- Step 3. The final stage produces a boolean result whose value is TRUE if there are one or more isolated nets:

```
RESULT := FALSE
FOR i = 1, ..., |N|
   RESULT := RESULT OR (U(Ni) < = 0)
RESULT := RESULT OR ((U(N) - ((2 * |N|) - 2)) < 0)
RESULT := RESULT AND NOT(|N| = 1)
```

The example given in Figure 6.7 evaluates to FALSE before the arc (P_{14},P_{31}) is made and TRUE once the arc is connected. Hence, connecting the arc causes net isolation. The three stage algorithm used for checking the processor graph for isolated nets has a complexity of $\mathbf{O}(|\mathbf{P}| + |\mathbf{I}| + |\mathbf{O}|)$. This algorithm is applied to **V** throughout its construction.

6.7 Summary

In this chapter a parallel processing model representing a loosely coupled Transputer based architecture has been described. The two resources important to scheduling have been identified, namely processing and communications. The model has been shown to consist of two parts: (i) a processor graph $\mathbf{V} = (\mathbf{P}, \mathbf{I}, \mathbf{O}, \mathbf{L})$ (based mainly on the two resource types) and (ii) $|\mathbf{L}|$ activity schedules $\mathbf{S} = \{S_1, ..., S_q\}$. The processor graph gives a spatial representation of the machine (i.e. topology), while the activity schedules give a temporal representation (i.e. activity). The data structures for both parts of the model have been presented and illustrated using examples.

The latter part of this chapter has concentrated on the processor graph and the rules governing nodes and arcs that comprise the graph. Input and output has been included and the minimum system realisation has been defined which complies with that of DFDL. Finally, the problems accorded to building the graph have been discussed, this has been shown to create isolated nets unless precautions are taken. A solution to net isolation has been presented in the form of a three stage algorithm. The algorithm examines the graph whenever an arc is proposed, producing a boolean result which indicates whether or not net isolation would occur if the arc were established.

Chapter 7. Compile-time scheduling

Automated scheduling is performed either at compile-time or at run-time. Both scheduling techniques are aimed at optimising the mapping between program and processor architecture, but vary in their approach and system requirement.

Run-time scheduling has the advantage of being able to respond to dynamic changes in the system, while maintaining a reasonable efficiency, however, it achieves this at the expense of an inevitable run-time overhead. Run-time overhead and scheduling efficiency are related, such that sophisticated scheduling algorithms will tend to carry a greater overhead than simple ones. Hence, there is a tendency for run-time schemes to use simple schedulers in order to minimise overhead.

In order to achieve an efficient implementation when using compile-time scheduling, the characteristics of inputs to the scheduler (i.e. program and processor architecture) must be known at compile-time, i.e. a deterministic system. Non-deterministic systems are generally unsuited to compile-time scheduling, because static methods cannot adapt to the scheduling requirements of a dynamic system. Where a system is deterministic, as it is in our case, compile time scheduling offers superior performance, because the scheduler incurs no run-time overhead. Thus, complex scheduling strategies, including features such as network communication, can be incorporated in a scheduling algorithm which optimises, or tends to optimise scheduling.

The scheduling problem presented in this thesis assumes a deterministic system: a fixed set of tasks **T** are to be scheduled on to a fixed set of processing elements **P**, in compliance with the partial ordering on **T** (represented by **A**) and the connectivity of **P** (represented by **L**). The problem is to find an efficient scheduling algorithm for sequencing the tasks to optimise, or tend to optimise some desired performance measure. The primary performance measures of concern are schedule length and the average time data spends in the system. Both these measures play an important role in determining the performance of a real-time system; schedule length, because it determines the throughput of the system and maximum time data spends in the system, because of its correspondence to latency.

A great amount of research has been carried out on classic scheduling problems and these have been shown to encompass several disciplines, e.g. operations research, management science, computing, etc. Several reviews on sequencing research (Elmaghraby, 1968; Day and Hottenstein, 1971) have been made and these cover a wide classification of problems, including dynamic scheduling. Conway, Maxwell and Miller (1967), and later Coffman (1976), both present a comprehensive study into deterministic scheduling. However, much of the research has neglected the effects of non-zero communication cost, which probably reflects the low relative cost of communication in some systems and the complexity of accounting for it in others.

More recently, consideration has been given to communication issues. Price and Pooch (1982), for example, present a scheduling method that aims to minimise inter-processor communication using a backwards shortest path algorithm, while others, have used "pairwise exchange" techniques (Lee and Aggarwal, 1987) and multiple priority heuristics (Polychronopoulos and Banerjee, 1987). Without doubt though, communication issues will play a greater role in implementation as processing speeds increase and loosely-coupled processor architectures predominate.

7.1 Scheduling model

The scheduling model includes our previous two models, the task graph **G** and the processor graph **V**, and is described with consideration to the tasks, resources, their constraints and performance measures. To begin, the task and resource models are briefly reviewed.

7.1.1 Task graph

The model of the program is represented as a directed acyclic graph, **G** = (**T**, **C**, B, E, **A**). **T** is a set of processor executable nodes, or tasks and **T** = $\{T_1, ..., T_n\}$, whose members all take a non-negative integer time to execute. For a task T_i this execution time is denoted e_i. The set of communication tasks **C** = $\{C_1, ..., C_g\}$ is empty prior to scheduling and added to during the scheduling process, whenever two connected tasks in **T** are assigned to different processing elements, or there is an external input or an external output. Each communications task C_c takes a positive integer time c_c to complete. The two single nodes B and E are the initiating and terminating nodes respectively, hence B precedes all other nodes in **G** and E succeeds all other nodes in **G**. The execution time of these nodes is zero.

A is a partial order on **T**, **C**, B and E which determines the task structure, potential parallelism and precedence constraints on the tasks. The relation **A** has on **T**, **C**, B and E is irreflexive, asymmetric and transitive, which has the effect of making **G** a directed acyclic graph, which can only represent deterministic programs. Precedence between tasks is given by the arcs belonging to **A** and is denoted by a node pair. For example, the arc (T_i,T_j) signifies that task T_j cannot commence until T_i completes.

In addition to e_i, the cost of executing T_i, there are several other parameters associated with each task. The CPM (critical path method) produces the parameters EST(T_i), LST(T_i) and FLT(T_i), which are the earliest time T_i can begin, the latest time T_i can begin without extending the length of the critical path and the float which is the difference between the latest and earliest times.

Set operators **SUC**(T_i), **IMSUC**(T_i), **PRED**(T_i) and **IMPRED**(T_i) operate on **G** to produce all tasks that are successors of T_i, all tasks that are immediate successors of T_i, all tasks that are predecessors of T_i and finally, all tasks that are immediate predecessors of T_i.

7.1.2 Processor graph

The model of the object parallel processor is represented by a partially directed graph called the processor graph **V** = (**P**, **I**, **O**, **L**). **V** includes a non-empty, finite set of nodes **P** = {P_1, ..., P_m}, where each node P_j corresponds to a processing element. It also includes a finite set of nodes **I** = {I_1, ..., I_u} and a non-empty, finite set of nodes **O** = {O_1, ..., O_v}, elements from these sets represent input ports and output ports respectively.

The non-empty, finite set of arcs **L** defines how elements in **P**, **I**, and **O** are connected and hence, it describes the topology of the object architecture. The arcs correspond to physical communication links and are expressed as a parenthesised pair of nodes, these being the nodes that the link joins. The relation **L** has on the nodes that comprise inter-processor links, (P_{ia},P_{jb}) is irreflexive and symmetric (i.e. bi-directional flow), while the relation on the nodes that comprise I/O links, (I_i,P_{jb}) and (P_{ia},O_j) is irreflexive and asymmetric (i.e. uni-directional flow from left node to right node).

The input and output ports derive from the input and output declarations made in the program that is being compiled, whereas the number of processors and link connections are entered by the user at compile-time on request from the compiler. The scheduler does allow some or all of the inter-processor link definitions to be omitted, in which case the scheduler will place links automatically (where needed) using an auto-router.

The processing elements and communication links are the system's resources and associated with each resource is an activity schedule (or just schedule). The set of schedules **S** = {S_1, ..., S_q} is finite and non-empty and has a cardinality equal to the sum of |**P**| and |**L**|. Each member of **S** is itself a set of tasks. These tasks either originate from **T**, where the schedule S_k = {T_u, ..., T_v} is for a processing element, or from **C**, where the schedule S_k = {C_u, ..., C_v} is for a link. All schedule sets are empty prior to scheduling and are filled during the scheduling process. The ordering on the tasks within S_k is strictly sequential, hence tasks are executed in the order that they are listed in their schedule.

7.1.3 Sequence constraints within schedules

There are three broad classes of scheduling; list scheduling, non pre-emptive scheduling and pre-emptive scheduling. The merits of each has to be considered before we choose the most suitable scheduling class for our problem, then we can consider some of the constraints placed on tasks during the scheduling process in order to maintain integrity.

List scheduling operates by a simple mechanism, whereby tasks are scheduled from a single pre-ordered sequential list of tasks. Whenever a processing element becomes inactive the next task in the pre-ordered list is scheduled to that resource. Provided the ordered list is constructed beforehand, list scheduling has the lowest degree of sophistication of all three scheduling classes which makes it particularly attractive when scheduling at run-time.

The second class considered here is non pre-emptive scheduling, which is a super-set of list scheduling. A non pre-emptive scheduling method considers not just one task, but all tasks that are available for scheduling (i.e. tasks that have had all their predecessors scheduled). Consequently, the complexity of this class of scheduling increases over that of list scheduling, however, the benefit of non pre-emptive scheduling is the potential for increased scheduling efficiency.

Finally, pre-emptive scheduling allows task execution to be interrupted and a task removed from the schedule, under the assumption that it will be re-scheduled at a later time. This contrasts with non pre-emptive methods which consider tasks to be atomic units whose execution cannot be suspended until the task has completed. In comparison to the previous two classes, pre-emptive scheduling has the potential to generate the most efficient schedules. One drawback with using pre-emption is the comparative cost of de-scheduling and re-scheduling, especially when tasks have short execution times. Pre-emptive scheduling is at its most efficient when task execution times are large (in comparison with overheads), as is the case for large grain structures.

The task graph illustrated by Figure 7.1 is scheduled onto the simplified processor graph (Figure 7.2), which comprises two processing elements joined by a single link. The schedules are illustrated by Figure 7.3 for each class of scheduling: (a) list scheduling, (b) non pre-emptive scheduling and (c) pre-emptive scheduling.

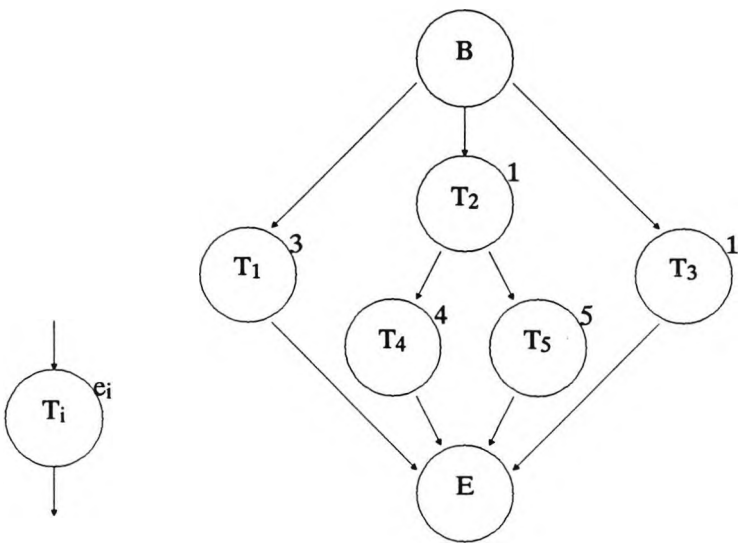


Figure 7.1 Task graph, G

Data is passed via L1 wherever tasks are scheduled on a different processing element than their predecessor was scheduled. Communication is represented by a task C_j, which has a finite communication cost, denoted c_j. In this example c_j equals 2.

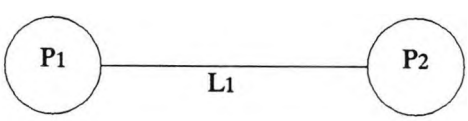
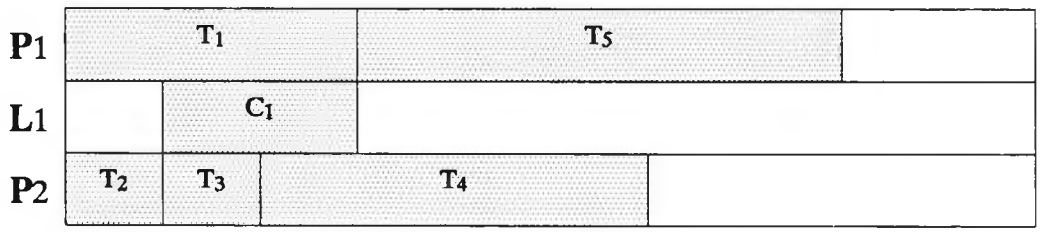
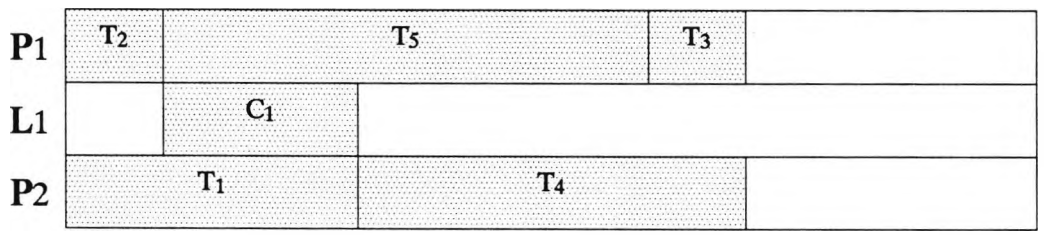


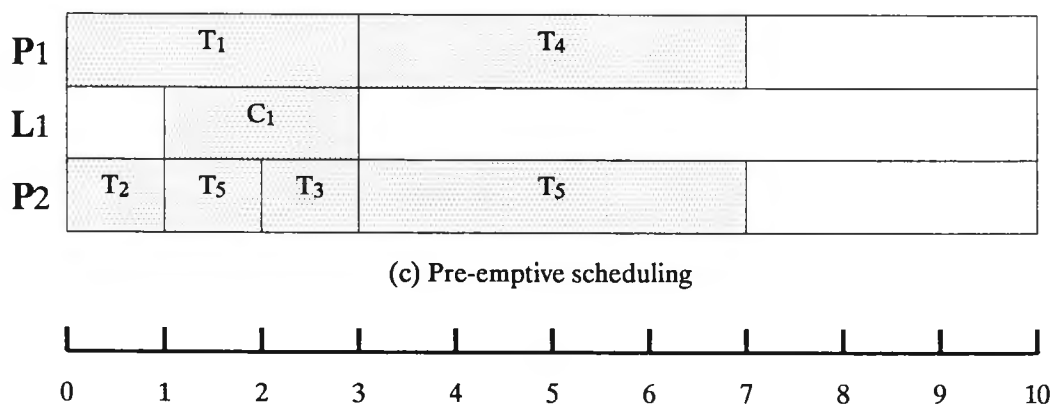
Figure 7.2 Processor graph, V



(a) List scheduling
List priority = $T_1 > T_2 > T_3 > T_4 > T_5$



(b) Non pre-emptive scheduling



(c) Pre-emptive scheduling

Figure 7.3 Scheduling methods

The class of scheduling described in this thesis is of a non pre-emptive type. It is preferred to list scheduling because it produces schedules that are potentially more efficient and is preferred to pre-emptive scheduling because the latter would incur sizeable de-scheduling and re-scheduling overheads for the medium/fine grain structure of **G**.

Program integrity has to be maintained throughout the scheduling process otherwise the object program will not fulfil the requirements of the source program. Loss of integrity would manifest itself as one or more of the following; variable misassignment, variable non-assignment, race conditions, oscillations or deadlock. Clearly, these errors are to be avoided, this may be achieved by extending the scope of **A** to all the members of **S**.

In practice integrity is maintained by adhering to two rules; the first governs the selection of tasks that are available for scheduling and the second governs the ordering of a task in a schedule, with respect to other previously scheduled tasks.

Tasks that are available for scheduling are included in the finite set **R**, which is the set of available tasks and is a sub-set of **T**. **R** only contains tasks that are independent of one another and whose predecessors have been scheduled. More formally, a task T_i in **R** is defined as a task that has all its predecessors **PRED**(T_i) belonging to **S** and the intersection of its successors **SUC**(T_i) and **S** is empty. Also, the intersection of **R** and **PRED**(T_i) is empty as is the intersection of **R** and **SUC**(T_i). Therefore, there are no precedence relationships between tasks in **R**, hence all tasks have equal precedence.

Whenever a task T_i is scheduled it is moved from **R** to the appropriate schedule in **S**. T_i's immediate successors will only be placed in **R** iff the rules for independence hold for that successor task. As scheduling progresses unscheduled tasks are included in **R**, while newly scheduled tasks are removed from **R** and placed in **S**. Consequently all tasks in **T** belong to **R** at sometime during the scheduling process, starting with those tasks belonging to **IMSUC**(B) and finishing with those tasks in **IMPRED**(E).

The ordering of a task T_i in a schedule S_k with respect to existing tasks, is a matter for the scheduling algorithm. Tasks that are already resident in S_k are either members of **PRED**(T_i), or are independent of T_i and are therefore members of **T** that do not lie in either **PRED**(T_i) or **SUC**(T_i) and are not T_i. Obviously T_i cannot be inserted in the schedule at a place where the resource is active, however, it may inadvertently be inserted ahead of some of its predecessors, assuming there is surplus resource activity at that point.

A satisfactory method of preventing precedence violations (as defined by **A**) is to allocate each scheduled task T_i (in schedule S_k) a start and finish of execution time, denoted $s_i(k)$ and $f_i(k)$ respectively. $s_i(k)$ is calculated in much the same way as is EST(T_i) (section 2.2.6), except $s_i(k)$ includes the overhead associated with finite communication cost and resource unavailability. $f_i(k)$ is simply $s_i(k) + e_i$. The rule for placing a task T_i in schedule S_k states that T_i is placed in the schedule S_k at a point where:

> (i) all scheduled tasks T_j that lie to the left of T_i (towards time zero) have a start of execution time $s_j(k)$ that is no greater than $s_i(k)$ and
>
> (ii) the resource corresponding to S_k is inactive at the time $s_i(k)$ for a period greater or equal to the time taken to execute T_i, given as e_i.

If the two conditions of the rule cannot be met for an initial value of $s_i(k)$, then $s_i(k)$ is increased until both conditions are satisfied. Figure 7.4 shows an example of task placement in several stages.

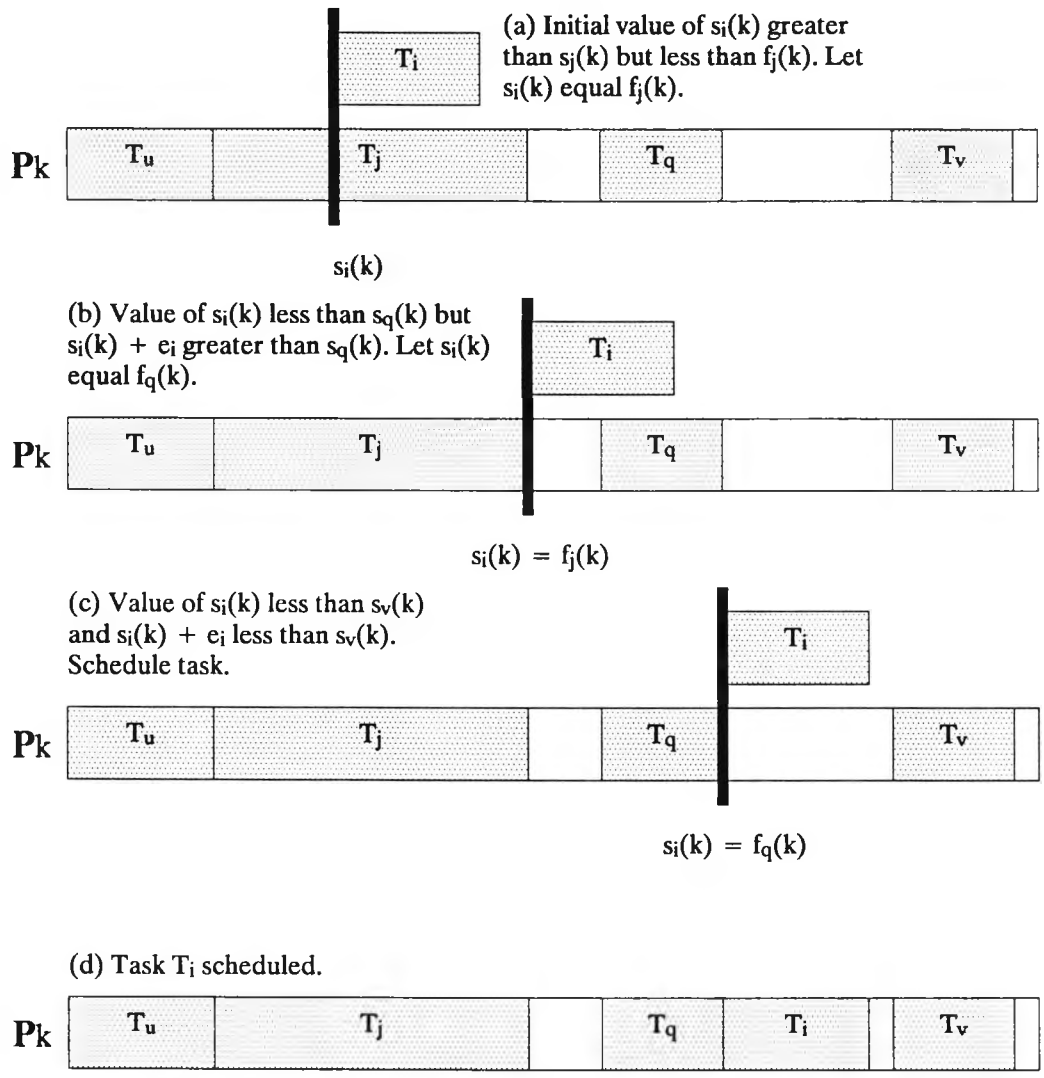


Figure 7.4 Task scheduling

When task placement involves synchronised communication, either between adjoining processors, input and processor, or output and processor, then the preceeding scheduling method becomes slightly more complex. The added complexity stems from the need to search each schedule that is participating in the communication in a simultaneous fashion. The schedule operation is similar to that described, but has the added condition that (i) and (ii) are satisfed for all participating schedules.

The Gantt chart of Figure 7.4 is an informal and intuitive notion of the schedule. Somewhat more formally, a non pre-emptive schedule can be defined as a suitable mapping that in general assigns a sequence of contiguous execution intervals in $[0, Z^+]$ to each task such that:

(i) Exactly one resource is assigned to each interval.

(ii) The sum of the intervals assigned to a task is precisely the execution time of the task.

(iii) No two execution intervals of different tasks on the same resource can overlap.

(iv) Precedence constraints are observed.

(v) There is no interval in [0, MAX{w(k)}] during which no resource in S is active (i.e. all resources are not allowed to be idle when incomplete tasks exist).

This concludes the description of schedules and the rules that govern the tasks which reside within. Usually schedules will be presented in their diagrammatic form (Gantt chart) with time along the x-axis and resource along the y-axis.

7.1.4 Performance measures

There are several different schedule measures that can be made, however, the principal measures of schedule performance considered here are schedule length (or maximum finishing time) and lateness (time tasks are overdue).

For a schedule S_k the schedule length is given as $w(k)$ and for the set of schedules S the maximum schedule length is denoted MAX{$w(1)$, ..., $w(q)$}, or just MAX{$w(k)$}. The maximum schedule length is important, since it places a limit on the throughput of the system and it is our objective to design a scheduling algorithm which will maximise throughput.

The lateness of a task T_i in S_k is defined as $l_i(k) = s_i(k) - LST(T_i)$, which is a task's scheduled time minus its latest starting time of execution (as calculated from the CPM). For a schedule S_k the mean lateness is defined as l(k) which is equal to $1 / |S_k|$ SUM{$l_i(k)$}, for all T_i in S_k. For the set of schedules S, lateness is the maximum lateness taken over all schedules. Reducing task lateness at a local level (i.e. for each $l_i(k)$) has a desirable effect on both throughput and latency, which results in an overall reduction in mean lateness for a schedule. However, if we were to reduce the global, mean lateness arbitrarily, it does not necessarily follow that throughput and latency would be reduced. Hence, lateness has to be reduced at a local level, rather than just at a global level.

Other measures of interest include the schedule usage $u(k)$ which is the accumulated time a resource has been active. This measure may be divided by MAX$\{w(k)\}$ to give the fractional schedule usage. The final measure discussed here is speed up, which can be defined in two ways. The first is a ratio of the sum of all processor executable tasks and the maximum schedule length, given as SUM$\{e_i\}$ / MAX$\{w(k)\}$, while the second is a fractional value that is the critical path length divided by the maximum schedule length, given as w^* / MAX$\{w(k)\}$.

7.1.5 Definition of the scheduling problem

As a precursor to discussing the complexity of scheduling the scheduling objective is stated thus:

- Instance: Finite set **T** of n tasks, each having an execution time $e_i = x$, x is a member of $\mathbf{Z}^+$, partial order **A** on **T**. Finite set **P** of m processors connected by a finite set **L** of q communication links. Finite set **C** of g communications tasks, each having a communication cost $c_j = y$, y is a member of $\mathbf{Z}^+$, and a deadline $w^\wedge$, $w^\wedge$ is a member of $\mathbf{Z}^+$.

- Question: Is there an (m + q) resource schedule **S** for **T** and **C** that meets the overall deadline $w^\wedge$ and obeys the precedence constraints in **A**?

This is called the scheduling problem and for convenience it is written $\overline{|\ |}$.

Scheduling is performed with the aim of producing a maximum schedule length that will be less than or equal to the deadline, i.e. MAX$\{w(k)\}$ $<= w^\wedge$. Ideally, the best possible deadline to achieve is the optimal deadline (in a minimal sense) given the constraints, since this gives the maximum throughput. Therefore, our aim can be restated thus: when a task is scheduled, it is scheduled with the aim of producing a maximum schedule length that will be equal to the optimal schedule length, i.e. MAX$\{w(k)\}$ $= w^\wedge$, where $w^\wedge$ assumes the optimal deadline.

7.2 Scheduling complexity

Assume for the moment that all communication costs e_c are zero, therefore, the time taken to send data from one processing element to another processing element in a connected network is zero. The implication of zero cost communication is that the topology of **V** becomes unimportant as far as scheduling is concerned, hence, the complexity of $\overline{|\ |}$ (the scheduling problem) is reduced. This simplified model of the scheduling problem serves to show how complexity issues affect the choice of our scheduling algorithm. This sub-problem of $\overline{|\ |}$ is denoted $\overline{|\ |}^*$.

$\overline{|\ |}^*$ can now be stated as:

- Instance: Finite set **T** of n tasks, each having an execution time $e_i = x$, x is a member of $\mathbf{Z}^+$, partial order **A** on **T**. Finite set **P** of m processors, and a deadline $w\hat{}$, $w\hat{}$ is a member of $\mathbf{Z}^+$.

- Question: Is there an m processor schedule **S** for **T** that meets the overall deadline $w\hat{}$ and obeys the precedence constraints in **A**?

7.2.1 Ordering on T is empty

In order to investigate the properties of $\overline{|\ |}^*$ we first consider the case where the partial order **A** on **T** is empty (i.e. there are no precedence relationships between tasks in **T**). At the start of the scheduling process, all tasks in **T** are equal candidates for scheduling, because **A** is empty. These tasks are said to be "available for scheduling" and therefore belong to **R** immediately after B.

The scheduling process begins by placing all tasks that belong to **IMSUC(B)** in **R** and proceeds by searching **R** for the first task to be scheduled. Once found, the process removes this task from **R** and places it in the appropriate schedule. The process is repeated for all n tasks, removing tasks from **R** and placing them in **S** until all tasks in **T** are scheduled and only E (the terminating node) remains in **R**.

For all n *levels* of the scheduling process there are alternate avenues of choice between different processors-task assignments. Hence, a decision has to be taken at each level based on which one of the r (r is the cardinality of **R**) available tasks is to be scheduled on which one of the m processors.

The degree of choice at the i^{th} level, $i = \{1, ..., n\}$, is denoted b_i, which equals the number of processors m, multiplied by the number of tasks belonging to **R** at the i^{th} level, i.e. $m * r_i$. The value of r changes from level to level and is defined for **A** being empty as equal to $(n - i + 1)$ for the i^{th} level, hence b_i is equal to $m * (n - i + 1)$.

The number of choices overall is the product of the number of choices at each of the n levels. This is called the maximum branching factor, denoted b_{max}, which represents the total number of possible ways in which n independent tasks can be scheduled onto m processors.

$$b_{max} = PROD\{b_i\}, \text{ for } i = \{1, ..., n\}$$
$$b_{max} = m^n * (n * (n - 1) * (n - 2) * ... * 1)$$
$$b_{max} = m^n * n! \qquad (7.1)$$

7.2.2 Ordering on T is strictly sequential

We now consider the properties of $\overline{|\ |}^*$ when the partial order **A** on **T** is strictly sequential (i.e. $\mathbf{A} = \{(T_1,T_2), (T_2,T_3), ..., (T_{n-1},T_n)\}$). At the start of the scheduling process, only one task T_1 in **T** is a candidate for scheduling. This task is said to be "available for scheduling" and therefore belongs to **R** at the outset.

The scheduling process is performed as before, however, at each of the n *levels* of the scheduling process there is only ever one task available for scheduling. Therefore, there are now fewer alternate avenues of choice.

The degree of choice at the i^{th} level, for $i = \{1, ..., n\}$, is denoted b_i, which equals the number of processors m, multiplied by the number of tasks belonging to **R** at the i^{th} level, i.e. $m * r_i$. The value of r at different levels is constant throughout and is equal to 1, hence b_i is equal to $m * 1$.

The number of choices overall is the product of the number of choices at each of the n levels. This is called the minimum branching factor, denoted b_{min}, and represents the total number of possible ways in which n sequentially ordered tasks can be scheduled onto m processors.

$$b_{min} = PROD\{b_i\}, \text{ for } i = \{1, ..., n\}$$
$$b_{min} = m^n \qquad (7.2)$$

7.2.3 Non-zero communication costs

In both cases, when **A** is empty and when **A** is a sequential ordering on **T**, the branching factor for $\overline{|\ |}^*$ is an exponential function of order n. The two cases represent extreme examples of the number of possible ways to schedule n tasks onto m processors. When **A** is a partial order on **T**, the branching factor lies somewhere between these two extremes.

Returning to $\overline{|\ |}$, the re-introduction of positive integer communication costs between communicating processors can affect the branching factor significantly. When a task T_i is considered for scheduling onto a processor P_t and P_t is different from processor P_s, on which T_i's immediate predecessor is scheduled, data has to be passed from P_s to P_t before T_i can be executed. Data may be routed in as many different ways through the interconnection network as there are unique acyclic paths between the two processors. Clearly, the number of unique acyclic paths depends on both the location of each processor within the network and the network's topology. In addition to this, the number of different ways data is routed may rise significantly if the task has more than one predecessor.

Choice, arising from the alternative paths, is denoted p. From the discussion, it is evident that the value of p is difficult to generalise for a particular instance, however, a maximum value for p may be calculated providing the topology of the network is known.

One extreme example of p is found in a completely connected network, which produces a maximum number of paths for m processors (Figure 7.5). The network has a degree of (m - 1) and diameter of 1. The number of unique acyclic paths p is given as:

$$p_{max} = SUM\{x! / (x - v)!\}, \text{ for } v = \{0, ..., x\} \text{ and } x = m - 2 \quad (7.3)$$

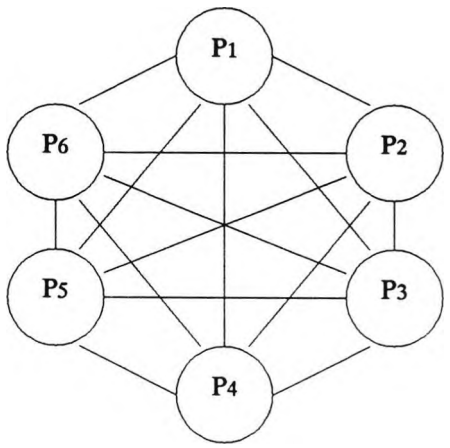


Figure 7.5 Completely connected network

At the other extreme is a linearly connected network, which produces a minimal number of paths for m processors (Figure 7.6). The network has a degree of 2 and a diameter of (m - 1). It only possesses a single path, hence:

$$p_{min} = 1 \quad (7.4)$$

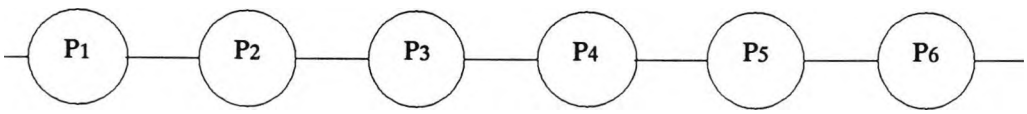


Figure 7.6 Linear connected network

Both these results remain consistent for any two processors (m > 1) within the network, this is because these example networks are both symmetrical. Equations for the maximum and minimum branching factor are re-written below to include the results for p_{max} and p_{min}. Equation (7.3) is included in (7.1) to give the upper bound, b_{max} and equation (7.4) is included in (7.2) to give the lower bound, b_{min}.

$$b_{max} = PROD\{b_i\}, \text{ for } i = \{1, ..., n\}$$
$$b_{max} = (1 + (m - 1) * p_{max})^n * n!$$
$$b_{max} = (1 + (m - 1) * SUM\{x! / (x - v)!\})^n * n!, \quad (7.5)$$
$$\text{for } v = \{0, ..., x\} \text{ and } x = m - 2$$

$$b_{min} = PROD\{b_i\}, \text{ for } i = \{1, ..., n\}$$
$$b_{min} = (1 + (m - 1) * p_{min})^n$$
$$b_{min} = m^n \qquad (7.6)$$

Note that the choice of path, p is available (m - 1) times and not m times, because there are no choices of path when $P_s = P_t$.

7.2.4 Exhaustive enumeration

The previous three sections have discussed the effects of different partial orders **A** and non-zero communication cost on the number of different ways in which n tasks can be scheduled onto m processors. The two results, (7.5) and (7.6), represent the upper and lower bounds of the number of solutions to | | in terms of n (the number of tasks in **T**) and m (the number of processing elements in **P**).

It has been established that there are between b_{min} and b_{max} solutions to | |. If the g^{th} arbitrary solution to | | is enumerated, we will produce a set of schedules whose maximum length (in time) is MAX$\{w(k)\}^g$. The complexity of enumerating this single solution is a polynomial function of n. The question we must ask in order to solve | | is, "Does MAX$\{w(k)\}^g$ of the g^{th} solution to | | equal $w^\wedge$?". If the answer to the question is yes then we have solved | |, otherwise another solution is tried. The search, enumeration and verification process is repeated until | | is solved.

Up to now, we have assumed that the optimal value for MAX$\{w(k)\}$, given as $w^\wedge$, is known prior to scheduling. Unfortunately, $w^\wedge$ cannot be found until | | is solved, hence, all solutions to | | have to be enumerated and verified against previous results in order to solve | | for $w^\wedge$. This method of solving | | is known as exhaustive enumeration, because all solutions are enumerated to find the result. The maximum schedule length for the optimal solution can be expressed as a MIN-MAX problem:

$$w^\wedge = MIN\{MAX\{w(k)\}^g\}, \text{ for } g = \{1, ..., b\} \qquad (7.7)$$
where $b_{min} <= b <= b_{max}$,
and for $k = \{1, ..., q\}$ where $q = |\mathbf{P}| + |\mathbf{L}|$.

From equation (7.7), we see that solving $\overline{|\ |}$ by exhaustive enumeration requires a minimisation over b solutions, which requires b enumerations and (b - 1) comparisons. The time complexity of $\overline{|\ |}$ is proportional to the amount of computation (performed by a deterministic computer) and hence, is proportional to b, the number of solutions. Even for the lowest bound on b, denoted b_{min}, the complexity is an exponential function of order n. The implications of exponential time complexity are illustrated by Table 7.1 for different values of n. As a comparison, polynomial time complexity is included. The example assumes the time to enumerate a single solution is 1 mS, irrespective of its length.

	n = 10	n = 50	n = 100
n^2	0.1 s	2.5 s	10 s
n^3	1.0 s	125 s	16.7 m
2^n	1.024 s	856, 850 yrs	$9.6\ 10^{20}$ yrs
3^n	59 s	$5.46\ 10^{14}$ yrs	$3.9\ 10^{38}$ yrs

Table 7.1 Polynomial and Non-polynomial complexity

The results in Table 7.1 show that the time taken to solve $\overline{|\ |}$ makes exhaustive enumeration impractical for all but the most trivial of problem. This phenomenon is known as combinatorial explosion (Cohen, 1976).

7.2.5 P and NP problems

Essentially, all problems are said to fall into one of two classes, either P or NP, where P is a sub-set of NP. It is generally accepted that if a problem lies in class P, there is an algorithm which exists that can solve the problem "quickly" and the algorithm is referred to as being "good".

Problems that lie outside P and are in the class NP (non-polynomial), can only be solved in polynomial time on a non-deterministic computer (Turing, 1936). One can conjecture that problems that are outside P do not have a "good" algorithm that can solve the problem "quickly". This reflects the viewpoint that exponential algorithms should not be considered as being "good" algorithms, and indeed this is usually the case.

Most exponential algorithms are merely variations on exhaustive enumeration, whereas polynomial time algorithms generally are made possible only through the gain of deeper insight into the structure of the problem. Problems that require exponential algorithms to solve them (since they are so hard that no polynomial algorithm can solve them), are generally considered intractable.

The question that has to be answered is, "Does $\overline{|\ |}$ lie outside the class P, in NP (known as NP-complete) ?". Some previous results are given below:

(i) Unconstrained scheduling with zero communication cost (section 7.2.1): $|\ |^*$ remains NP-complete for $m >= 2$, but can be solved in pseudo-polynomial time for any fixed number m. If all task execution times are equal, this problem is trivial to solve in polynomial time.

(ii) Precedence constrained scheduling with zero communication costs: $|\ |^*$ can be solved in polynomial time if $m = 2$ and all task execution times are equal (Coffman and Graham, 1972). This becomes NP-complete if task execution times of 1 and 2 are allowed (Ullman, 1975). Complexity remains open for all $m >= 3$ even when task execution times are equal. The NP-completeness of $|\ |^*$ may be proved by a transformation from the 3-satisfiability problem (Ullman, 1975).

(iii) Precedence constrained scheduling with non-zero communication costs: $\overline{|\ |}$ is a more complex case of (ii), therefore remains NP-complete for all fixed numbers $m >= 2$ when task execution times are unequal.

Previous results suggest that $\overline{|\ |}$ is NP-complete, which implies that the problem is inherently intractable when solved on a deterministic computer. The following section discusses some heuristic algorithms which attempt to overcome intractability and solve $\overline{|\ |}$ in polynomial time.

7.3 Search strategies

Instead of using an algorithm that enumerates every possible solution, when trying to solve a problem with an exponential number of solutions, we examine algorithms which restrict the scope of enumeration. This is achieved by pruning solutions, or partially enumerated solutions that do not show promise of solving the problem. These algorithms are in generally regarded as being more efficient than exhaustive enumeration, and some form the basis of even more complex algorithms which are described elsewhere (Pearl, 1984).

7.3.1 Solution space

When collected together, all the solutions to a problem are said to occupy the problem's solution space. Solution space can be classified as being polynomially, or exponentially related to the problem size. | |, for example, has a solution space that is exponentially related to n. All solutions, as we have seen, comprise n levels of task-processor-path assignments, where each assignment represents a decision. These points of decision are called *solution nodes* and all solution nodes belong to the solution graph, which is an arborescence.

Each series of n solution nodes begins at the root of the graph and terminates at one of the many leaf nodes. Each series represents a uniquely ordered solution to | |. However, many of these uniquely ordered solutions will result in identical schedules, because in some cases, changing the order in which two (or more) independent tasks are scheduled has the same outcome.

7.3.2 Generating, expanding and exploring solution nodes

The most elementary step of graph searching is node generation, that is producing a solution node from its predecessor. A new successor node is then said to have been generated and its predecessor is said to have been explored. A coarser step is called node expansion, which consists of generating all immediate successors of a given node. The node is then said to have been expanded.

7.3.3 Informed and uninformed search

Search strategies fall into one of two categories, informed and uninformed. The difference between the two categorisations originates from the mechanism that decides the order in which nodes are expanded. In uninformed search, node expansion order depends only on information gathered by the search and is unaffected by the character of the unexplored portion of the graph. Informed search, however, uses information about the problem domain and about the nature of the goal to help guide the search towards the more promising solutions.

Three different search strategies are now considered (Backtracking, Hill-climbing and Best-first). All these strategies are far more efficient that exhaustive enumeration, because they prune the solution graph and only concentrate on what they believe to be promising solutions. Pruning the solution graph helps reduce the amount of enumeration needed to solve the problem.

7.3.4 Backtracking

The first search strategy we discuss is called *backtracking*. This strategy is classified as uninformed, in the sense that the order in which the search proceeds does not depend on the nature of the solution that is sort. Being uninformed, backtracking is somewhat inefficient and is usually deemed to be impractical for solving large problems, however, it is worthy of description so we can compare and contrast informed, heuristically guided strategies.

Backtracking (BT) applies a last-in-first-out policy to node generation. When a node is first selected for exploration, only one of its successors is generated and this new node is submitted for exploration. If, however, the generated node meets some stopping criterion, the search program backtracks to the closest unexpanded predecessor, that is the predecessor still having un-generated successors.

BT is a version of depth-first search, in that it explores downwards in preference to exploring sideways, across the solution graph. It operates by maintaining a record of the minimum length solution encountered so far and continues to search until it becomes certain that no cheaper solution lies ahead. The stopping criterion employed by BT, for our problem, would be the length of the best solution encountered so far, or when the bottom of the solution graph is met. When this criterion is met part way through the graph, the solution sub-tree which lies ahead offers no further improvement over the current best solution. Consequently, the sub-tree is pruned and no further enumeration takes place within that sub-tree. Pruning un-promising solutions before they are fully enumerated offers significant reductions over exhaustive enumeration.

The run-time of BT depends heavily on where the optimal solution lies within the solution graph, because BT cannot adapt its order of search but mechanically searches in a pre-determined uninformed fashion; say depth-first from left to right. Should $w\hat{}$ lie to the far left of the solution graph, for example, then pruning would be more severe than if $w\hat{}$ lay to the right of the graph. Another influence on the run-time of BT is the number of solutions whose lengths are close to that of $w\hat{}$ and consequently, only fail to offer a promising solution once they are almost completely enumerated. Conceivably, either of these phenomenon would cause BT to enumerate a vast part of the solution graph, hence, BT has a worst-case complexity that is exponential in n and is therefore regarded as an inefficient algorithm.

One note of praise for BT is its memory efficiency, since it only requires n temporary storage elements to explore the graph.

7.3.5 Hill-climbing

The second search strategy we discuss is called *hill-climbing*. This strategy is classified as informed, in the sense that the order of search depends on the nature of the solution that is sort.

Hill-climbing (HC), a strategy based on local optimisation, is the most popular search strategy among human problem solvers. It is called hill-climbing because like a climber who wishes to reach the top of the mountain quickly, HC chooses the path of steepest ascent form its current position.

In terms of our graph search model, HC amounts to repeatedly expanding a node, inspecting the newly generated successors and choosing and expanding *only* the best among the successors. No further reference to the predecessor, or its other successor nodes is retained. This strategy is irrevocable, because the process does not allow the search to shift its scope back to previously suspended alternatives, even though they may eventually offer the promise of a better solution.

HC has problems with graphs that are not commutative (Nilsson, 1980), because these graphs contain paths that lead to incomplete solutions. Hence, when HC is applied, there is the likelihood it will run up a dead end and will not be able to reverse and seek an alternative path. In our case the property of commutivity holds and irrevocable strategies can be applied without the risk of missing a solution. Expanding the wrong node will lead to a non-optimal solution, but will not prevent a solution from being sort.

HC is a useful strategy when we possess a highly informative heuristic which steers the search away from "false" locally optimal solutions and towards a more global solution. It advances quickly towards a solution at the expense of missing the optimal solution and has the advantage of requiring a minimum of temporary memory storage for operation.

The run-time of HC is independent of where the optimal solution lies within the solution graph, because the search is steered by the results attained from node expansion (informed). Run-time is also independent of the number of solutions whose lengths are close to that of $w\hat{}$, because HC only pursues a single locally optimal solution. Analysing HC, we find it has a time complexity *TC* that is related to the sum of the number of nodes generated at each of the n levels. This is given as follows:

$$TC_{max} = (1 + (m - 1)p_{max}) * n(n + 1) / 2$$
$$TC_{max} = (1 + (m - 1) * SUM\{x! / (x - v)!\}) * n(n + 1) / 2, \quad (7.8)$$
$$\text{for } v = \{0, ..., x\} \text{ and } x = m - 2.$$

$$TC_{min} = (1 + (m - 1)p_{min}) * n$$
$$TC_{min} = m * n \quad (7.9)$$

Hence, HC has a worst-case time complexity that is a polynomial of n. Order of time complexity is written as below:

$$TC_{max} = \mathbf{O}(n^2) \tag{7.10}$$

7.3.6 Best-first

The final search strategy discussed here is known as *best-first*. This strategy, like hill climbing is classified as informed. Best first (BF) is also similar to HC in that the forward searching motion is always taken from the last decision through the most promising successor. However, what sets BF apart from other search strategies is the commitment to select the best from all nodes currently expanded, no matter where they are in the partially developed tree. The implication of this strategy is that the optimal solution will always be discovered after the fewest possible number of decisions.

BF works like many co-operating teams of mountaineers who approach the mountain from different paths, all seeking the highest peak on the mountain. Whenever a team meets a branching path it divides and only the team on the most promising path moves ahead while all others wait until their path becomes more favourable (if ever) than all others.

Unfortunately, the breadth-first spread of a BF strategy means it pays dearly in node storage, since it has to allow the search to resume at any previously suspended alternative. However, this spreading of alternate paths means that BF can solve graphs that are non-commutative.

The run-time of BF is independent of where the optimal solution lies within the solution graph because the search is informed, however, unlike HC, run-time is dependent of the number of solutions which show promise. Graphs could be contrived where all solutions showed reasonable promise so that BF enumerated a large portion of the solution graph. Hence, BF has a worst-case complexity that is exponential in n.

7.3.7 Hybrid search

Pearl (1984) describes the three search strategies (BT, HC and BF) as three extreme points of a continuous spectrum of search strategies. Different search strategies can be visualised as lying on the plane whose axes are *scope of recovery* and *scope of selection*, such a diagram is illustrated in Figure 7.7. The area within the three extreme points represents hybrid search strategies which are compromises between scope of recovery and scope of selection. Note that only those strategies that lie on the right hand edge of our diagram are guaranteed to yield an optimal solution.

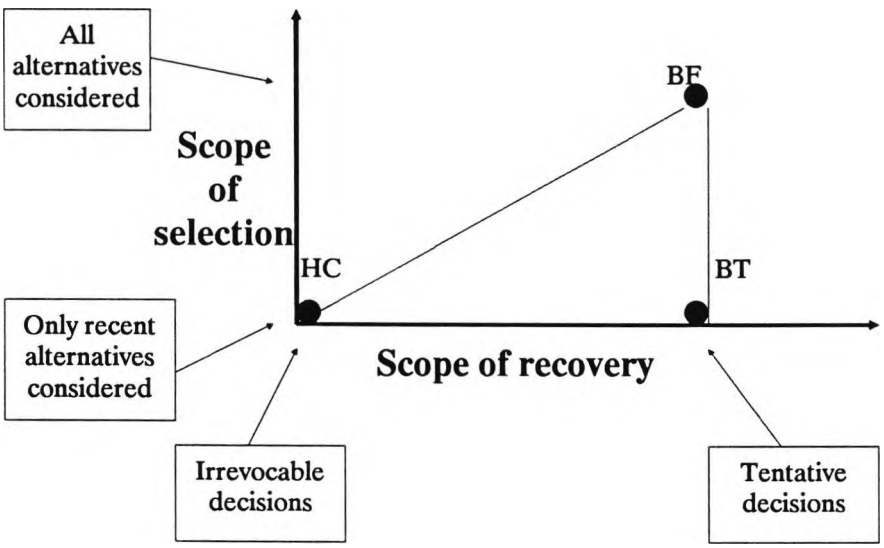


Figure 7.7 2-dimensional space of hybrid strategies

One example of a hybrid strategy employs a BT-BF combination to cut down on the storage requirement, however, this is at the expense of reducing the number of alternatives considered. These strategies although more memory efficient, remain exponentially bounded in their computational complexity.

One consequence of introducing irrevocable decisions are that optimallity can no longer be guaranteed. However, the benefits of reduced computational complexity and reduced memory requirements may outweigh the disadvantage of not being able to guarantee the result as optimal. Nilsson (1971) introduced a hybrid BF-HC strategy called staged search, which performs a BF search that halts when the memory allocation is used up. From the partial result, only a small number of the most promising paths are kept and remainder are discarded. The search then continues in a BF manner, beginning from the most promising paths. The routine is repeated until the memory is exhausted once again. Another BF-HC strategy, proposed by Pearl (1984), is similar to Nilsson's staged search, but discards all but the most promising path and consequently has a much decreased run-time.

The type of search strategy selected depends on both the problem characteristics and the characteristics of the executing system. In the following section | | is shown to comprise, not one, but two search problems; an inner shortest path search and an outer decision problem. A hybrid strategy is adopted which applies BT-BF and HC strategies to these problems respectively.

7.4 Scheduling algorithm

This section describes a scheduling algorithm which employs a hybrid search strategy that is a composite of the strategies previously discussed. The scheduling algorithm is divided into two major parts, known as the *inner part* and *outer part*. The inner part uses a BT-BF strategy to expand nodes and find the next locally optimum solution node (i.e. task-processor-path assignment). The inner part is repeated at each of the n levels of scheduling and is controlled by the outer part, which employs an HC strategy.

The algorithm is designed to find a solution to | | in polynomial time, however, this is not possible unless the condition of optimallity is relaxed. Hence, $w\hat{}$ is redefined as a deadline that tends to the optimal, i.e. an optimal solution cannot be guaranteed. The polynomial time complexity and non-optimallity characteristics are an effect of introducing irrevocable decisions by way of the HC strategy.

It has been suggested that efficiency and accuracy of a scheduling algorithm greatly depend on the heuristic that is used. To illustrate this point, we begin the description of the algorithm by showing two identical examples that employ different heuristic measures to guide informed search. The results of the two heuristic measures are compared against the optimal solution.

7.4.1 Heuristic measures

7.4.1.1 Minimum length heuristic

The first heuristic we discuss is known here as the minimum length heuristic. It approaches the decision problem by selecting the solution node that produces a minimal increase in the overall schedule length. More formally, this can be stated as below:

For the h^{th} level of any partial solution to | | the maximum schedule length is given as:

$$w_h = \mathrm{MAX}\{w(k)\}_h,$$

where subscript h indicates the solution has only been enumerated to the h^{th} level, $1 <= h < n$.

Similarly, the maximum schedule length for the $(h + 1)^{th}$ level is given as:

$$w_{h+1} = \text{MAX}\{w(k)\}_{h+1}$$

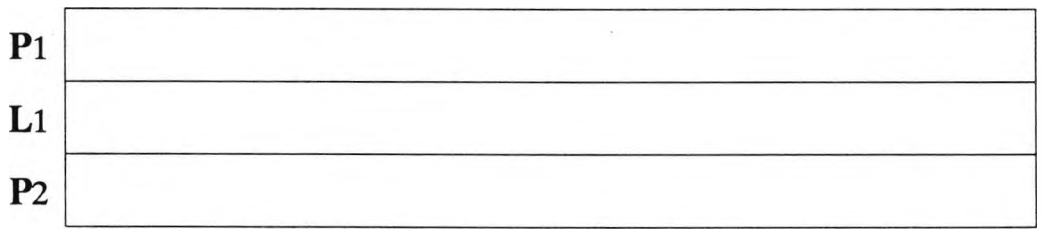
The heuristic always chooses the solution node, denoted Y_{h+1}, that results in the smallest difference between w_h and w_{h+1} for $h = \{0, ..., (n-1)\}$ and $w_0 = 0$.

$$Y_{h+1} \mid \text{MIN}\{w_{h+1} - w_h\}, \text{ or}$$

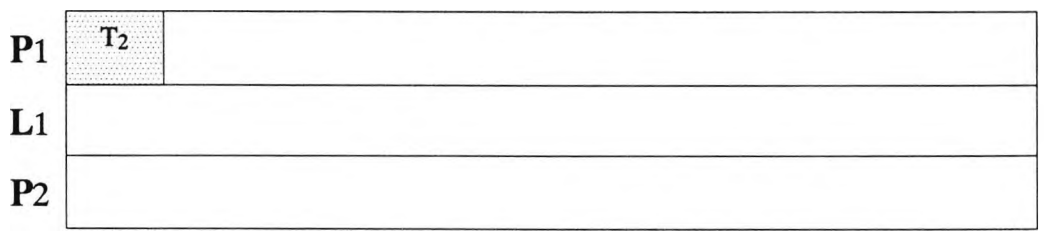
$$Y_{h+1} \mid \text{MIN}\{\text{MAX}\{w(k)\}_{h+1}\}, \text{ hence}$$

$$Y_h \mid \text{MIN}\{\text{MAX}\{w(k)\}_h\}, h = \{1, ..., n\} \qquad (7.11)$$

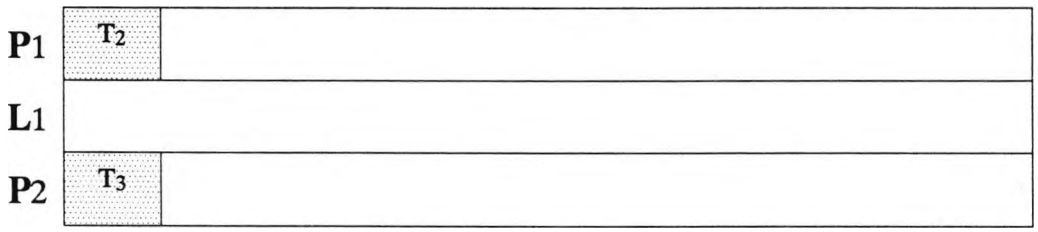
Using the minimum length heuristic to tend to optimise $\overline{|\ |}$ would appear to be a reasonable strategy, since it optimises locally the very measure which it aims to optimise globally. Some results are given for scheduling the example task graph (Figure 7.1) onto the processor graph (Figure 7.2) using the minimum length heuristic. These are illustrated by Figure 7.8. Task-processor-path assignment is given as T_i-Pj-C_q and where there is no inter-processor communication X replaces C_q.



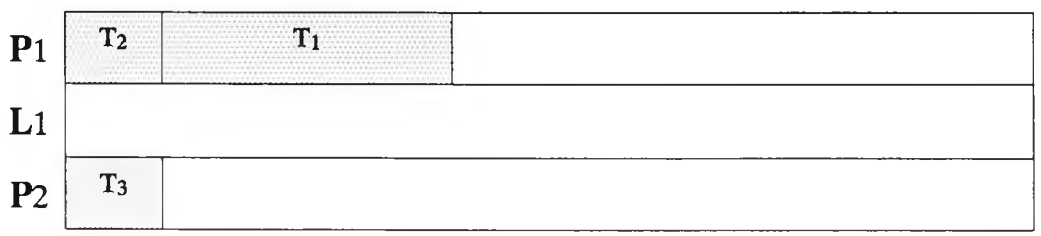
$R = \{T_1, T_2, T_3\}, \text{MAX}\{w(k)\}_0 = 0$



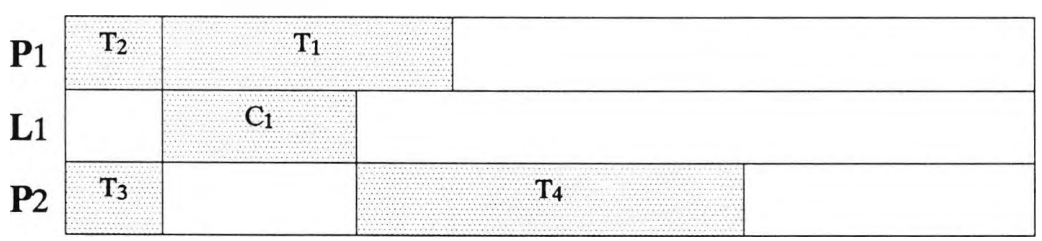
$R = \{T_1, T_3, T_4, T_5\}, Y_1 = T_2\text{-P1-X}, \text{MAX}\{w(k)\}_1 = 1$



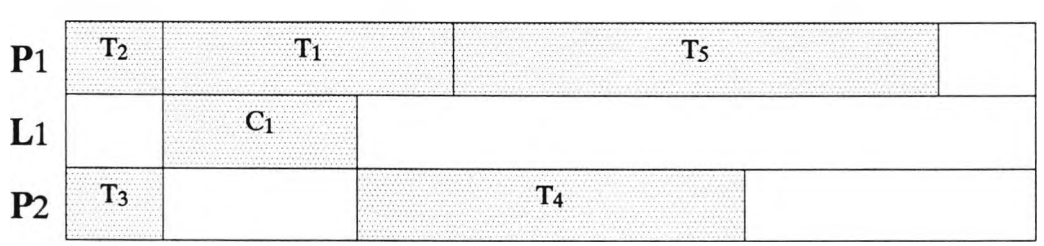
$\mathbf{R} = \{T_1, T_4, T_5\}, Y_2 = T_3\text{-}P_2\text{-}X, \text{MAX}\{w(k)\}_2 = 1$



$\mathbf{R} = \{T_4, T_5\}, Y_3 = T_1\text{-}P_1\text{-}X, \text{MAX}\{w(k)\}_3 = 4$



$\mathbf{R} = \{T_5\}, Y_4 = T_4\text{-}P_2\text{-}C_1, \text{MAX}\{w(k)\}_4 = 7$



$\mathbf{R} = \{E\}, Y_5 = T_5\text{-}P_1\text{-}X, \text{MAX}\{w(k)\}_5 = 9$

Figure 7.8 Scheduling; minimum length heuristic

The heuristic fails to recognise the future consequences of choosing between different solution nodes, because it has no "look ahead" mechanism, and therefore bases its decisions on what it has already scheduled and the immediate choices that lie in front of it.

7.4.1.2 Minimum lateness heuristic

An improved heuristic (Goddard and Lawson, 1988b) derives some of its information from the CPM results (section 2.3.6), namely the parameter $LST(T_i)$, which is effectively a measure of what lies ahead on a particular data path. The heuristic uses $LST(T_i)$ in-conjunction with the solution node parameter $s_i(k)$ to give a lateness figure to each potential solution node.

The minimum lateness heuristic, as it is known here, aims to reduce the overall mean lateness of $\overline{|\ |}$ by minimising lateness locally. It was suggested in section 7.1.4 that localised minimisation of lateness resulted in a global reduction in lateness, which produced good throughput and latency figures. Indeed, this appears to be so. The minimum lateness heuristic is derived as follows:

A potential solution node has a lateness measure $l_i(k)$, which is derived from the latest start time of a task T_i and the task's potential start time $s_i(k)$, i.e.

$$l_i(k) = s_i(k) - LST(T_i) \tag{7.12}$$

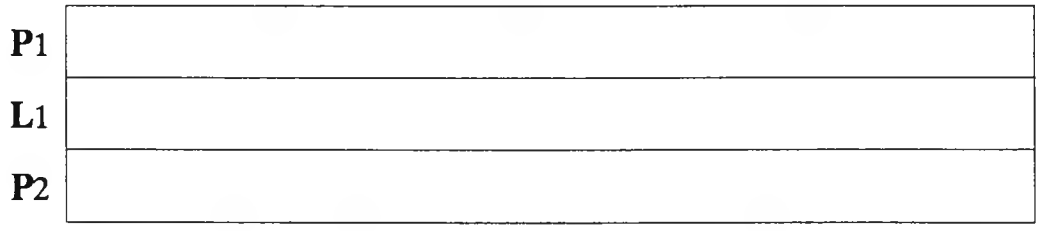
If for each scheduling level h, we find the minimum value of $s_i(k)$ for each task T_i in **R** and use this result in equation (7.12), then we can find the minimum lateness measure for each task T_i. In order to keep localised lateness to a minimum we adopt the strategy of always scheduling the solution node that has the maximum lateness measure. Hence, for the h^{th} scheduling level, where $h = \{1, ..., n\}$, the solution node Y_h is given as:

$$Y_h \mid MAX\{l_i(k)\}_h,$$

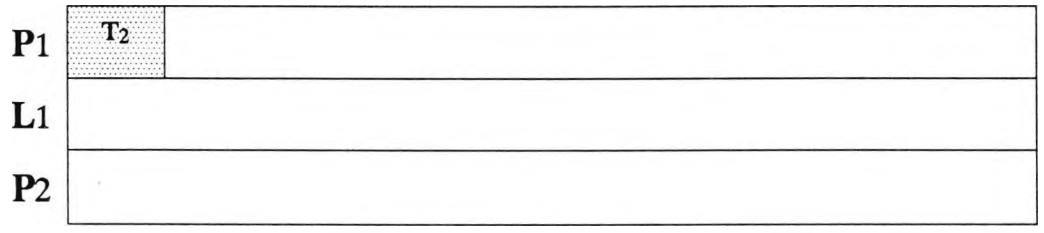
where subscript h indicates the solution has only been enumerated to the h^{th} level, $1 <= h <= n$.

$$Y_h \mid MAX\{MIN\{s_i(k)\} - LST(T_i)\}_h \tag{7.13}$$

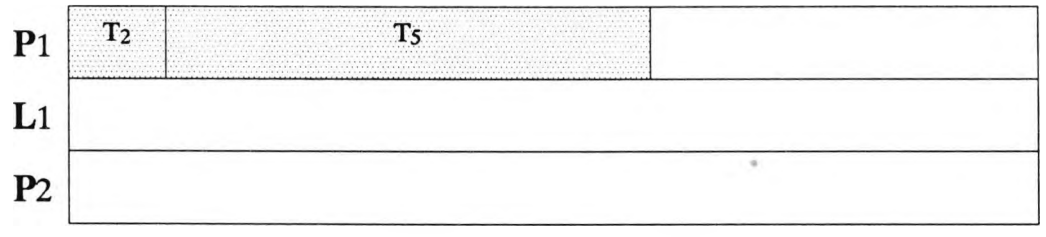
An example of scheduling, using the minimum lateness heuristic, is shown below for the task graph (Figure 7.1) and processor graph (Figure 7.2).



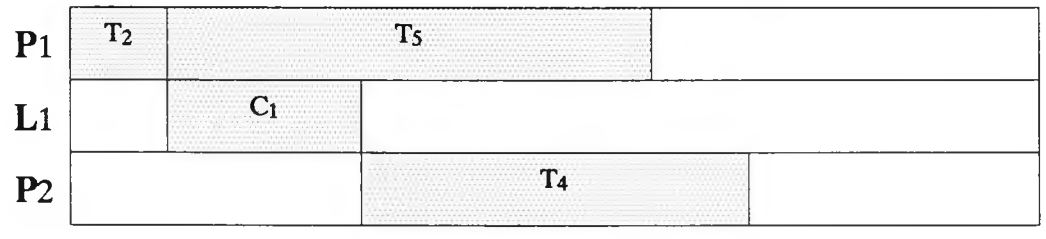
$\mathbf{R} = \{T_1, T_2, T_3\}, MAX\{l_i(k)\}_0 = 0$



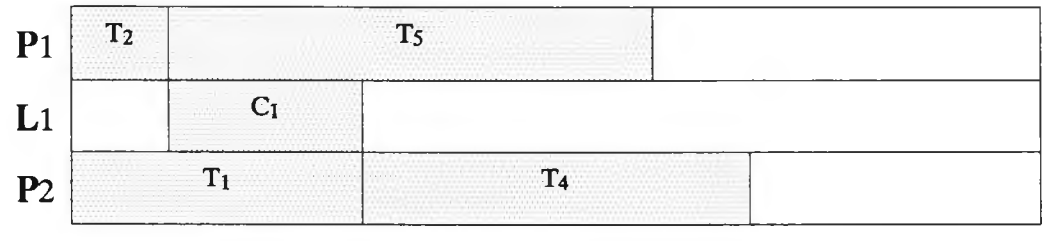
$\mathbf{R} = \{T_1, T_3, T_4, T_5\}, Y_1 = T_2\text{-}P_1\text{-}X, MAX\{l_i(k)\}_1 = 0$



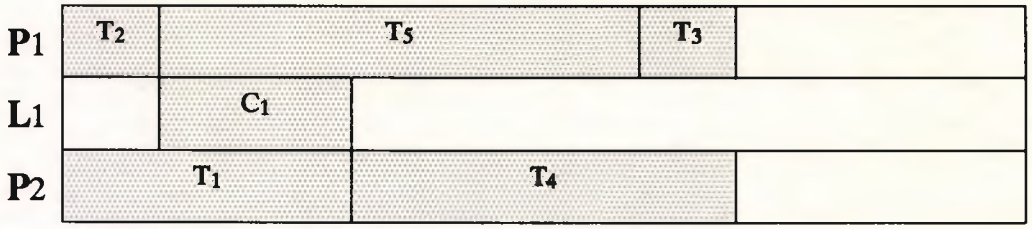
$\mathbf{R} = \{T_1, T_3, T_4\}, Y_2 = T_5\text{-}P_1\text{-}X, MAX\{l_i(k)\}_2 = 0$



$\mathbf{R} = \{T_1, T_3\}, Y_3 = T_4\text{-}P_2\text{-}C_1, MAX\{l_i(k)\}_3 = 1$



$\mathbf{R} = \{T_3\}, Y_4 = T_1\text{-}P_2\text{-}X, MAX\{l_i(k)\}_4 = -3$



$\mathbf{R} = \{E\}, Y_5 = T_3\text{-}P_1\text{-}X, MAX\{l_i(k)\}_5 = 1$

Figure 7.9 Scheduling; minimum lateness heuristic

Scheduling the task graph onto the processor graph using the minimum lateness heuristic gives an improved result over scheduling using the previous heuristic. In fact, the result (Figure 7.9) is known to be optimal for this particular example. The minimum lateness heuristic guides the scheduling algorithm, which is described over the following three sections.

7.4.2 Shortest path, a BF approach

The shortest path algorithm operates on the processor graph, **V**. The algorithm is a BF strategy that is a modified form of Dijkstra's shortest path algorithm (Dijkstra, 1959), which is generally acknowledged to be one of the most efficient algorithms for solving this type of problem. The modified algorithm works at two levels; the first level aims to find the shortest path from a starting processing element P_s to any other processing element P_x, while the second level scrutinises those schedules belonging to P_x in order to find the earliest available start of execution time for task T_i on P_x, denoted $s_i(x)$. The unmodified shortest path algorithm is described below.

The main idea underlying Dijkstra's shortest path algorithm is quite simple: Suppose we know the p processing elements that are closest (in terms of communication time) to processing element P_s in **V** and also know the shortest path from P_s to each of these processing elements. Colour processing element P_s and these p processing elements. Then, the $(p + 1)^{st}$ closest processing element to P_s is found as follows:

> For each uncoloured processing element P_y, construct p distinct paths from P_s to P_y by joining the shortest path from P_s to P_x with link (P_x,P_y) for all coloured processing elements P_x. Select the shortest of these p paths and let it tentatively be the shortest path from P_s to P_y.
>
> Which uncoloured processing element is the $(j + 1)^{th}$ closest processing element to P_s ? It is the uncoloured processing element with the shortest tentative path from P_s as calculated above. So, if the p closest processing elements are known then the $(p + 1)^{th}$ closest processing element can be determined as above. Starting with $p = 0$ this process can be repeated until the shortest path from P_s to any other processing element is found.

With this in mind, we can now formally state Dijkstra's shortest path algorithm, then later the modified version will be introduced.

7.4.2.1 Dijkstra's shortest path algorithm

- Step 1. Task T_i is available for scheduling and **IMPRED**(T_i) = T_k. T_k is already scheduled on P_s. Initially, all links and processing elements in **V** are uncoloured. Assign a number $d_i(x)$ to each processing element P_x to denote the shortest path (in terms of communication time) from P_s to P_x that uses only coloured processing elements as intermediate points of communication. Initially, set $d_i(s) = f_k(s)$, the finishing time of T_k on P_s, and set $d_i(x)$ = Inf (infinity) for all x not equal to s. Let P_y denote the last processing element to be coloured. Colour P_s and let $P_y = P_s$.

- Step 2. For each uncoloured processing element P_x redefine $d_i(x)$ as follows:

  $$d_i(x) = \text{MIN}\{d_i(x), d_i(y) + a_i(P_{sa},P_{xb})\} \qquad (7.14)$$

  If $d_i(x)$ = Inf for all P_x, then stop because no path exists from P_s to any uncoloured processing element. Otherwise, colour the uncoloured processing element P_x which has the smallest value of $d_i(x)$. Also colour the link directed into this processing element from the coloured processing element that determined $d_i(x)$ in the above minimisation. Let $P_y = P_x$.

- Step 3. If processing element P_t (target processor) has been coloured then stop because the shortest path from P_s to P_t has been discovered. This path consists of the unique path of coloured links from P_s to P_t. If processing element P_t has not been coloured yet, repeat step 2.

Note that whenever the algorithm colours a processing element (except P_s) the algorithm also colours a link directed into this processing element. Thus, each processing element has at most one coloured link directed into it, and the coloured links cannot contain a cycle since no link is coloured if both its endpoints have a coloured link incident to it. Therefore, we can conclude that the coloured links form an arborescence rooted at P_s. This arborescence is called a shortest path arborescence. The unique path from P_s to any other processing element P_x contained in the shortest path arborescence is the shortest path from P_s to P_x.

Note that the derivation of a "path length", denoted $a_i(P_{sa},P_{xb})$, is the sum of the communication cost and resource waiting time for (P_{sa},P_{xb}), at that particular moment in time. Hence, path length is not only determined by the communication cost, but also by the activity of the schedule.

7.4.2.2 Modified Dijkstra's shortest path algorithm

A modified version of Dijkstra's algorithm includes two stopping criteria which prune the solution graph to prevent fruitless searching, these are a lower stopping criteria L_i and an upper stopping criteria U_i. U_i is defined as the minimum starting time for T_i on P_y, where P_y represents all processing elements that lie on the shortest path arborescence. The lower stopping criterion L_i is a bound derived from outside the shortest path algorithm (section 7.4.3) which causes abandonment of the shortest path algorithm if violated. The algorithm works as follows:

- Step 1. Task T_i is available for scheduling and **IMPRED**(T_i) = T_k. T_k is already scheduled on P_s. Initially, all links and processing elements in **V** are uncoloured. Assign a number $d_i(x)$ to each processing element P_x to denote the shortest path (in terms of communication time) from P_s to P_x that uses only coloured processing elements as intermediate points of communication. Initially, set $d_i(s) = f_k(s)$, the finishing time of T_k on P_s, and set $d_i(x)$ = Inf (infinity) for all x not equal to s. Let P_y denote the last processing element to be coloured. Colour P_s and let $P_y = P_s$.

  Starting from time $d_i(y)$, search the schedule belonging to P_y (increasing in time) until a start time is found where the schedule can accommodate task T_i of execution time e_i (section 7.1.3). Assign this start time to $s_i(y)$. If $s_i(y) <= L_i$ abandon attempts to schedule T_i, otherwise let $U_i = s_i(y)$ and $P_t = P_y$.

- Step 2. For each uncoloured processing element P_x redefine $d_i(x)$ as follows:

  $$d_i(x) = \text{MIN}\{d_i(x), d_i(y) + a_i(P_{ya}, P_{xb})\}$$

  If $d_i(x)$ = Inf for all P_x, then stop because no path exists from P_s to any uncoloured processing element. Otherwise, colour the uncoloured processing element P_x which has the smallest value of $d_i(x)$. Also colour the link directed into this processing element from the coloured processing element that determined $d_i(x)$ in the above minimisation. Let $P_y = P_x$.

  Starting from time $d_i(y)$, search the schedule belonging to P_y (increasing in time) until a start time is found where the schedule can accommodate task T_i of execution time e_i. Assign this start time to $s_i(y)$. If $s_i(y) <= L_i$ stop; abandon attempts to schedule T_i. Otherwise, if $s_i(y) > L_i$ and $s_i(y) < U_i$ then let $U_i = s_i(y)$ and $P_t = P_y$.

- Step 3. If $d_i(y) >= U_i$ stop; all further paths exceed the minimum value of $s_i(y)$, hence no earlier start time is possible and therefore the shortest path from P_s to P_t has been discovered. Otherwise, if all processing elements are coloured then stop; the shortest path from P_s to P_t has been discovered. Otherwise, repeat step 2.

The two stopping criteria, L_i and U_i, greatly reduce the amount of enumeration required to find the shortest path from P_s to P_t, such that $s_i(t)$ is the minimum value of $s_i(y)$. Below are several extensions to the BF algorithm, these are necessary for the different types of tasks that exist in **G**.

7.4.2.3 Extension 1

The above illustration of the shortest path algorithm is only suitable when T_i has a monadic input (i.e. $|\mathbf{IMPRED}(T_i)| = 1$). When T_i has a dyadic input, two BF algorithms are executed concurrently. Each algorithm starts from the processing element that their member of $\mathbf{IMPRED}(T_i)$ is scheduled. The two algorithms stop once the minimum value of $s_i(y)$ has been found for the same processor, such that $s_i(y) = MAX\{s_i(y)^1, s_i(y)^2\}$ (the superscripts ^{1} and ^{2} indicate the results are from the two concurrent algorithms).

Care has to be taken when paths from each of the two algorithms use the same link, otherwise link activity already allotted to one path may be used by a succeeding path. The possibility of this contention only arises when data from two paths are moving in the same direction and not when they are in opposing directions, since in the latter case, only one path at most would remain once P_t is established.

7.4.2.4 Extension 2

Another difference between the BF algorithm and that which is implemented, occurs when tasks are pre-assigned to processing elements. This condition only arises for tasks that are of type external output or of type internal output. The only change that is made is that the algorithm seeks the shortest path to a known processing element, irrespective of $s_i(t)$.

7.4.2.5 Extension 3

The final extension due to task differences arises for tasks that have the initiating node B as their immediate predecessor and are not of type external input, e.g. reals, constants, etc. These tasks can be scheduled on any of the m processing elements without incurring any communications overhead.

7.4.2.6 Extension 4

Perhaps the largest extension to the algorithm is auto-routing. When selected, auto-routing affects the shortest path algorithm by temporarily establishing paths via links that are not formed. Temporary links are formed by finding the shortest path from P_s to P_f, where P_f is a processor that has one or more "free links". Auto-routing temporarily connects this unconnected link to all other unconnected links (in turn) on other processing elements. The algorithm then continues as normal, using these temporary links as part of its shortest path arborescence. Auto-routing finishes by removing all temporary links so they do not interfere with successive executions of the algorithm.

If the BF algorithm is successful (i.e. $s_i(t) > L_i$), then the parameters $s_i(t)$ and (P_{sa}, ..., P_{tb}) are saved and passed up to the BT algorithm described in the following section.

7.4.3 Comparing task solution nodes, a BT approach

The remainder of the inner BT-BF scheduling algorithm is a simple backtracking strategy which manages the BF inner core. For each task in **R** (the set of tasks available for scheduling), BT selects a task T_i, calculates L_i, runs the inner core shortest path algorithm and acts on the result. The BT algorithm is repeated r times and terminates with a locally optimal task-processor-path assignment (i.e. a solution node). The algorithm is described below in detail:

- Step 1. Initially assign FLT = -(Inf). Let the first task in **R** equal T_i.

- Step 2. Calculate lower stopping criterion for T_i, L_i :

  $$L_i = LST(T_i) - FLT \qquad (7.15)$$

  Call SP algorithm (section 7.4.1). Passed parameters are:
  In: T_i, L_i, **G**, **V**. Out: $s_i(t)$, (Ps, ..., Pt).

- Step 3. If $s_i(t) > L_i$ then $TASK = T_i$, $START = s_i(t)$, $PATH$ = (Ps, ..., Pt) and $FLT = LST(T_i) - s_i(t)$. Otherwise, $TASK$, $START$, $PATH$ and FLT remain unaltered.

  If T_i is the last task in **R** stop; $TASK$ and $PATH$ contain the locally optimal task-processor-path assignment and $START$ contains the start of execution time of T_i. Otherwise, let T_i equal the next task in **R** and repeat step 2.

The BT algorithm is extremely memory efficient and as previously shown works in an uninformed way, inspecting all tasks belonging to **R** in a sequential fashion. The locally optimal task-processor-path assignment is passed up to the outer part of the scheduling algorithm which is described in section 7.4.4.

7.4.4 Scheduling assignment, an HC approach

The outer part of the scheduling algorithm operates according to a hill-climbing strategy. The heuristic, which this part depends, is based on the lateness of a task (section 7.4.1). HC is an irrevocable search strategy and consequently, moves towards a solution quickly using very little memory. The irrevocable characteristic of HC means it cannot return to previously suspended alternative solutions, but presses ahead steered by its heuristic information. HC calls the inner BT-BF part of the scheduling algorithm exactly n times and each time, BT-BF returns with a locally optimal solution node (optimal with respect to the heuristic) which is scheduled by HC.

The operation of the outer part of the scheduling algorithm is illustrated below:

- Step 1. **R** is initially empty. Place all tasks belonging to **IMSUC**(B) in **R**.
- Step 2. Call BT-BF algorithm (section 7.4.3). Passed parameters are: In: **R**, **G**, **V**. Out: *TASK*, *START*, *PATH*.

  Let T_i = *TASK*, $s_i(t)$ = *START* and (P_{sa}, ..., P_{tb}) = *PATH*.
- Step 3. Remove T_i from **R**. Examine all tasks belonging to **IMSUC**(T_i), if any member task T_k has no predecessors in **R** and all predecessors belong to **S** then place T_k in **R**.

  Place T_i in schedule S_t (the schedule corresponding to the processing element P_t) starting at time $s_i(t)$.

  If P_s is the same processing element as P_t then skip to step 5.
- Step 4. Check that all links on the path (P_{sa}, ..., P_{tb}) are established, if not then links have been temporarily made by auto-routing. Establish these links.

  For all links on the path (P_{sa}, ..., P_{tb})
  (i) schedule communication tasks in the schedule S_q which corresponds to the link
  (ii) insert communication tasks in the task graph **G**, between **IMPRED**(T_i) and T_i.
  This second point is achieved by appending the communication tasks to **C** (the set of communication tasks) and by altering **A** (the partial order on **T** and **C**) to include the communications task in **G** between **IMPRED**(T_i) and T_i.
- Step 5. If |**R**| = 1 and that one task is the terminating node E then stop; scheduling is complete. Otherwise, repeat step 2.

7.4.5 Complexity

The time complexity of the scheduling algorithm depends on the constraints placed on **T** and **P**, which are **A** and **L** respectively. For a worst-case situation, **A** would be empty and **L** would represent a completely connected inter-processor network. Hence, the upper bound to time-complexity is as follows:

For the BF part:

$$TC_{max} = 1 + (m - 1) * p_{max}$$
$$TC_{max} = 1 + (m - 1) * SUM\{x! / (x - v)!\}, \quad \text{for } v = \{0, ..., x\} \text{ and } x = (m - 2). \tag{7.16}$$

For the BT part, including (7.16):

$$TC_{max} = (1 + (m - 1) * SUM\{x! / (x - v)!\}) * r_h$$
$$TC_{max} = (1 + (m - 1) * SUM\{x! / (x - v)!\}) * (n + 1 - h), \quad (7.17)$$
for the h^{th} level of scheduling,
and for $v = \{0, ..., x\}$ and $x = (m - 2)$.

For the HC part, including (7.17):

$$TC_{max} = (1 + (m - 1) * SUM\{x! / (x - v)!\}) * (n + 1 - h) * n,$$
for $h = \{1, ..., n\}$, and for $v = \{0, ..., x\}$ where $x = (m - 2)$..
$$TC_{max} = (1 + (m - 1) * SUM\{x! / (x - v)!\}) * ((n + 1) / 2) * n,$$
for $v = \{0, ..., x\}$ and $x = (m - 2)$.

$$TC_{max} = \mathbf{O}(n^2) \quad (7.18)$$

Hence, the worst-case complexity is polynomial in n and of order 2. Complexity remains exponential of order m for completely connected networks, however, the low degree of most processing elements restricts strongly connected networks to low orders of m. Consequently, this exponential factor is not dominant. The lower bound for the algorithm's time complexity can be derived in a similar fashion, it is given as:

$$TC_{min} = m * n$$

$$TC_{min} = \mathbf{O}(n) \quad (7.19)$$

7.5 Summary

In this chapter the merits and limitations of deterministic, compile-time scheduling have been discussed and a comparison has been drawn between it and non-deterministic, run-time scheduling. Three different classes of scheduling have been illustrated; list scheduling, non pre-emptive scheduling and pre-emptive scheduling, and it was found that the non pre-emptive class is best suited to deterministic, compile-time scheduling.

Two major performance measures, schedule length and lateness, have been introduced and the relationship between these and throughput and latency has been discussed. The former measure is used in the definition of the scheduling problem, which is essentially a problem of schedule length minimisation.

The complexity of the scheduling problem was then investigated for different constraints on **T** and **P**, and the problem was found to have a number of solutions that is exponential of order n. To show the effect this has on inefficient algorithms that attempt to solve the problem, an exhaustive enumeration approach was illustrated. The problem was shown to be computationally intractable when using this type of algorithm. This result is reinforced by previous results which classify the problem as NP-complete.

Three basic heuristic search strategies, backtracking, hill-climbing and best-first have been introduced. The characteristics of each search strategy have been discussed and it was considered necessary to introduce irrevocable scheduling decisions to ensure that time complexity became polynomial in n. However, this step has been shown to lose the guarantee of an optimal solution being found.

The final part of this chapter describes the design of the hybrid HC BT-BF scheduling algorithm. At the heart of the algorithm is the heuristic, which guides the search strategy and so determines the scheduling decisions. Two heuristics have been compared, a minimum length heuristic and a minimum lateness heuristic, both originating from their relative performance measures. The minimum lateness heuristic has been shown to produce better results, because it uses information from the CPM algorithm which imparts "knowledge" of what lies ahead and so allows the algorithm to prioritise tasks accordingly. The chapter concludes with the derivation of the time complexity for the scheduling algorithm, which confirms it as polynomial in n, of order 2.

Chapter 8. Experimental results

This chapter presents the experimental results from which the effectiveness of the implementation strategy can be assessed. The chapter is divided into three main sections:

The first section examines the relationship between scheduling time and the parameters n and m. These results are compared to the predictions made in chapter 7. The second section looks at factors which affect the performance, or quality of scheduling, e.g., processor use, speedup, etc. This section focuses on the reasons for performance degradation, from some ideal, pre-calculated figure. Finally, the last section presents several worked examples. These are in the form a DFDL source program, pre-schedule program profile, schedule results and Occam object program.

Please note that the coefficient values given in the examples are for illustration only.

8.1 Scheduling time

Scheduling time complexity was estimated (in section 7.4.5) to lie in the region $\mathbf{O}(n)$ to $\mathbf{O}(n^2)$. This estimate excludes possible exponential growth due to m, since it is assumed that as m increases the degree of connectivity reduces and so n dominates the relationship. This section aims to show that this relation holds in practice.

As an example, the scheduling time for an n by n vector operation scheduled on an m processor network is illustrated. The example multiplies corresponding elements from each vector and outputs the product in a stream of n values. The program for the example is shown below and the results are illustrated in Figure 8.1.

```
PROG mult.vectors(OUTPUT(REAL32) y[n])
  BEGIN
    REPEAT FOREVER
      y[i FROM 0 FOR n] := a * b
  END
```

In the example, factors that rely on topology are removed, by setting the link communication cost to zero.

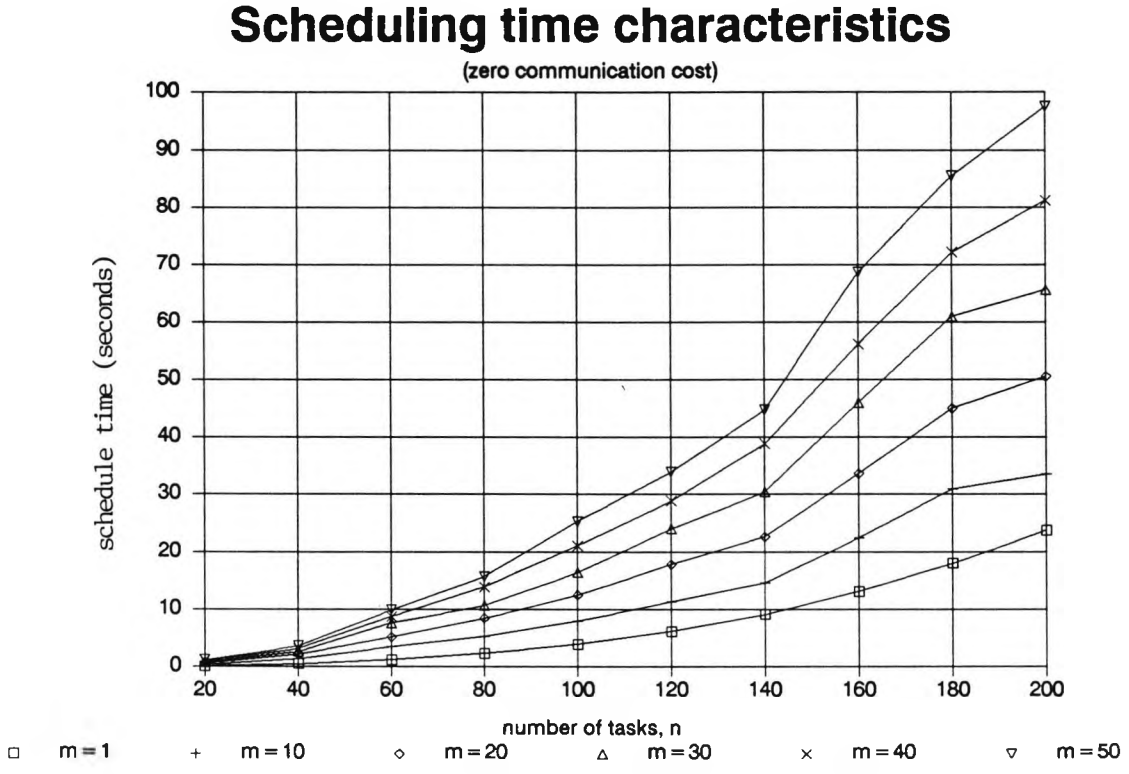


Figure 8.1

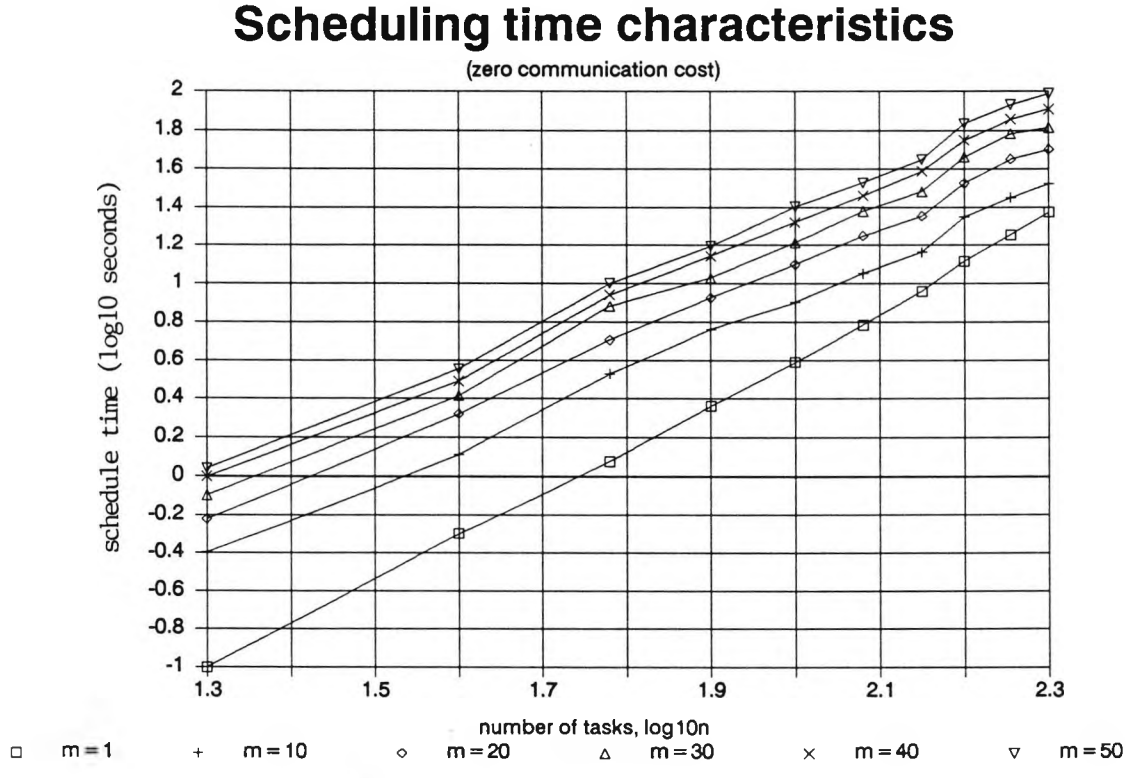


Figure 8.2

From Figure 8.1 we observe that scheduling time is far more sensitive to changes in n than it is to changes in m. For example, consider the point n = 100, m = 20, then doubling n increases scheduling time by 285% whereas doubling m only increases scheduling time by 69%. Sensitivity to changes in m will tend to rely on the size of m and the connectivity of the processor network.

On the assumption that the relation between n and scheduling time is a polynomial in n, then the plot of $\log_{10}(n)$ against $\log_{10}$(scheduling time) should produce a straight line whose gradient is the composite index of that polynomial. Figure 8.2 illustrates this relationship. All the plots in Figure 8.2 are close approximates to straight lines and the majority (m > = 10) have a gradient of 2. Using the equation for a straight line, $y := mx + c$, the scaling factor $a\log_{10} c$ can be found from Figure 8.2 for each value of m for this particular algorithm. These are summarised below in Table 8.1. The equation which relates n to scheduling time is described by (8.1), this equation gives a reasonable approximation over the range 10 < = m < = 100.

$$\text{scheduling time} = f(m) * n^2 \text{ seconds}$$

$$\text{scheduling time} = 4x10^{-5} * (10 + m) * n^2 \text{ seconds} \qquad (8.1)$$

m	f(m)
10	0.0008
20	0.0012
30	0.0017
40	0.0021
50	0.0025

Table 8.1 Scheduling time scale factor

8.2 Factors affecting performance

Examination of several different scheduling results reveal that optimal speedup for a given number of processors is seldom achieved. This section introduces and discusses several effects which lead to a degradation in performance. The first effect is called integer effect.

8.2.1 Integer effect

Integer effect becomes apparent whenever the number of independent tasks is not wholly divisible by the number of available processors, this produces schedules of dissimilar duration, which in turn creates less than optimal scheduling. The cause of this effect originates from the fact that tasks take a finite time to execute and cannot be shared between processors, since they are considered to be atomic. Integer effect is most noticeable when m approaches n, assuming m < n.

Consider the case where there are n independent tasks (i.e. **A** is empty) and each task has the same duration. Such a case is shown in the program below. These n tasks are scheduled onto m processors, where m < = n.

```
PROG mult.vectors(OUTPUT(REAL32) y[n])
  BEGIN
    REPEAT FOREVER
      y[i FROM 0 FOR n]  := a * b
  END
```

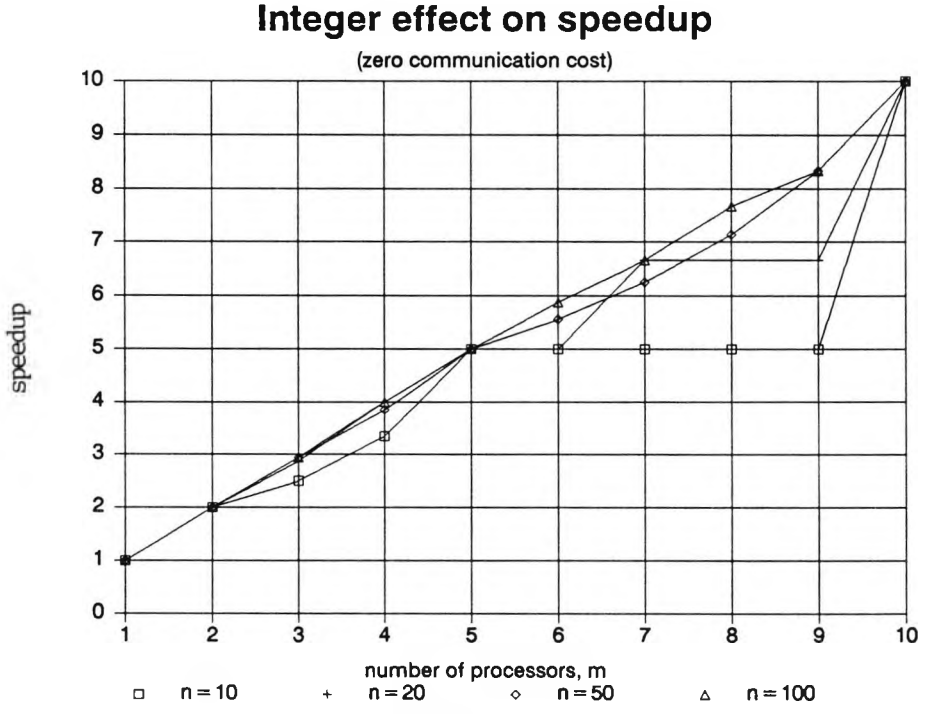


Figure 8.3

If all tasks were infinitely divisible then speedup would be

$$\text{speedup} = n / (n / m) = m \tag{8.2}$$

For a set of independent tasks this is regarded as the ideal case. However, tasks are not divisible, hence

$$\text{speedup} = n / \lceil n / m \rceil \tag{8.3}$$

Figure 8.3 shows the results of scheduling n independent, equally sized tasks onto m processors. The results confirm the relation given in (8.3) and show that possible degradation due to integer effect worsens as m approaches n.

To what extent can integer effect degrade speedup? From (8.3),

$$(n / m) + 1 > \lceil n / m \rceil >= n / m$$

Substituting the above limits into (8.3) shows that speedup lies in the range

$$(m * n) / (m + n) < \text{speedup} <= m$$

We have observed that integer effect worsens as m approaches n, thus let $m = n$

$$m / 2 < \text{speedup} <= m \tag{8.4}$$

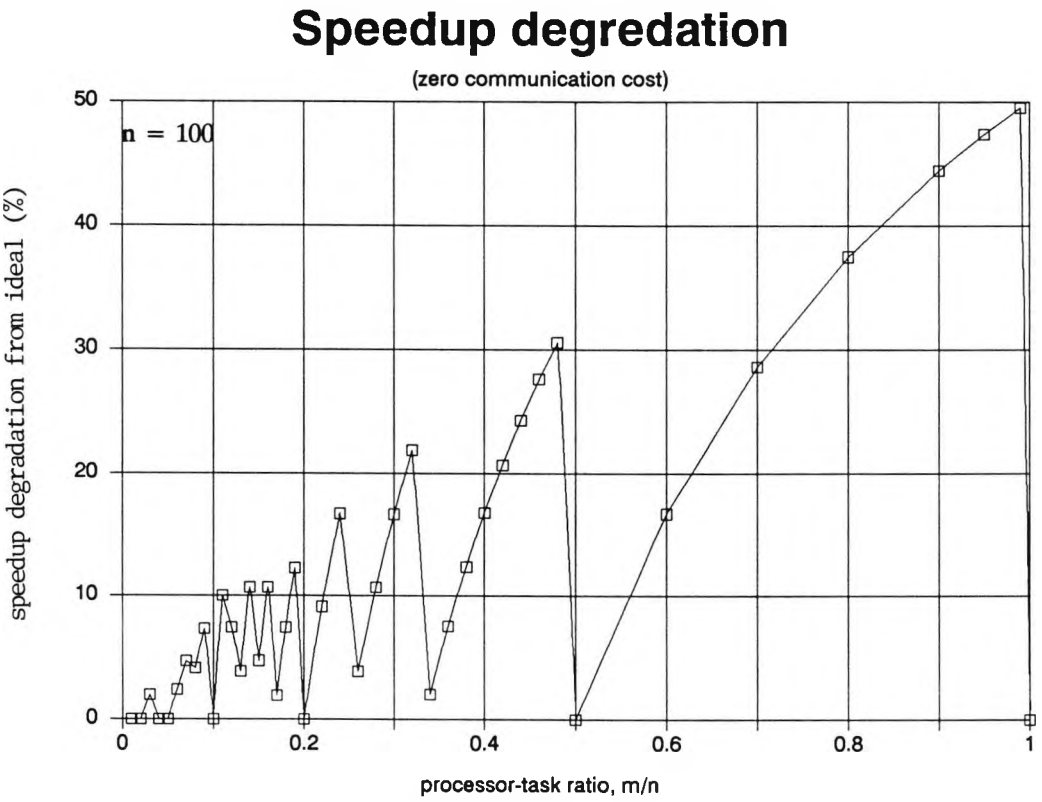


Figure 8.4

Equation (8.4) shows that integer effect can inflict a loss of up to 50% on speedup. This result is confirmed by Figure 8.4, which shows the percentage degradation across the entire range of the processor/task ratio (m / n). The plot is for n = 100 and percentage degradation is calculated from equation (8.5).

$$\text{percentage degradation} = (1 - (n / (m * \lceil n / m \rceil))) * 100\% \qquad (8.5)$$

The possibility of integer effect will always be present, however degradation can be minimised by employing a low processor/task ratio i.e. $m << n$.

8.2.2 Synchronisation effect

Synchronisation effect is the first of four distinct factors associated with communication which tend to reduce performance.

Synchronisation effect stems from Occam's unbuffered, synchronised model of communication which requires that communicating, parallel processes synchronise to commence data transfer. Consequently, a communication is not at liberty to occur freely, but has to wait until both sender and receiver have synchronised. In the current implementation of Occam on the Transputer, this cannot occur until both sender and receiver have met one of the following conditions: (i) have just completed processing a task, (ii) just about to process a task or (iii) inactive. Once synchronised, data transfer can proceed in parallel with processing.

The results of synchronisation effect are discussed in the following section.

8.2.3 Latent scope effect

When considering parallelism between processor and communication channels then Occam can express parallelism between one or more resources. For example

```
SEQ
  A
  B
  PAR
    C
    ch1 ? X
    ch2 ! Y
```

where ? is the Occam construct for a communication input and ! the Occam construct for a communication output.

Occam can also express nested parallel events

```
SEQ
  A
  B
  PAR
    C
    ch1 ? X
    ch2 ? Y
    PAR
      D
      ch3 ! Z
```

Occam can express separate, joint and hierarchical parallel activity, however, Occam cannot express parallel activity which partially overlaps. This language restriction leads to communication channels being held in scope for some time before and after they pass data. Consequently, channels appear to be busy when in fact they are inactive.

Synchronisation and latent scope effect are difficult to separate, since they both originate from Occam's concurrent model of computation. To illustrate their joint effect, a sum of products example is employed, this is shown below. The example is of fixed size (n = 199) and is scheduled onto a binary hypercube processor network (m = 16).

```
PROG madd100c(OUTPUT(REAL32) y)
  NODE k[100]
  BEGIN
    REPEAT FOREVER
      k[i FROM 0 FOR 100] := 1.0 * 1.0
      y := SUM(k[j FROM 0 FOR 100])
```

Figure 8.5 illustrates the composite processor activity over a single cycle of the example scheduled algorithm. The schedule assumes that communication between processors is zero cost. Superimposed upon this profile is a second profile of an ideal schedule. This ideal schedule accounts for integer effect and sequential dependency between tasks, but disregards synchronisation and latent scope effects.

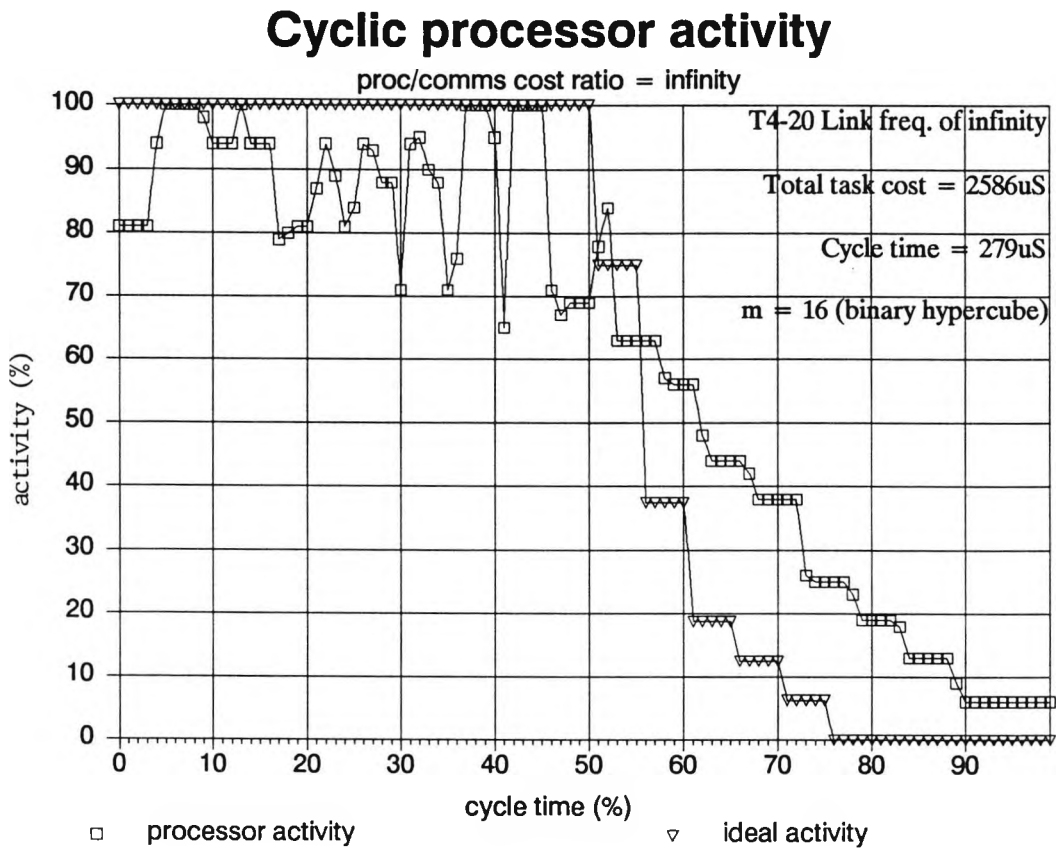


Figure 8.5

Figure 8.5 shows that cycle time for the ideal and actual cases is 212uS and 279uS respectively. These times relate to speedups of 12.2 and 9.3. In this example, synchronisation and latent scope effects reduce speedup by about 24%. Generally though, it is difficult to quantify such effects prior to scheduling, since they are not generated until the scheduling process. Both effects are a product of the Occam language which are reflected back into the requirements of the scheduler. Removal of these effects would only be possible by replacing the Occam/Transputer concurrent model of computation for a true data-driven model.

8.2.4 Finite communication cost effect

Finite communication cost tends to impede the spread of data from one processor to another. This effect increases as the processor/communication cost ratio decreases. Figures 8.6 through to 8.11 illustrate cyclic processor and cyclic inter-processor activity. Each figure represents a different processor/communication cost ratio, ranging from 30:1 to 0.3:1. This ratio is defined by equation (8.6). The series of figures reveal that as the processor/communications cost ratio is decreased then processor use decreases and communications use increases. Also, a decrease in this ratio produces an increase in the length of the cycle time.

$$\text{processor/communication cost ratio} = \bar{e} / \bar{c} \tag{8.6}$$

where $\bar{e}$ is the mean task cost and $\bar{c}$ is the mean communication cost.

One noticeable feature of all these activity profiles is that changes in processor activity are highly correlated to changes in communication activity and that profile changes are inversely proportional. This correlated feature is attributed to synchronisation and latent scope effect which give rise to interference between processor and communication activity.

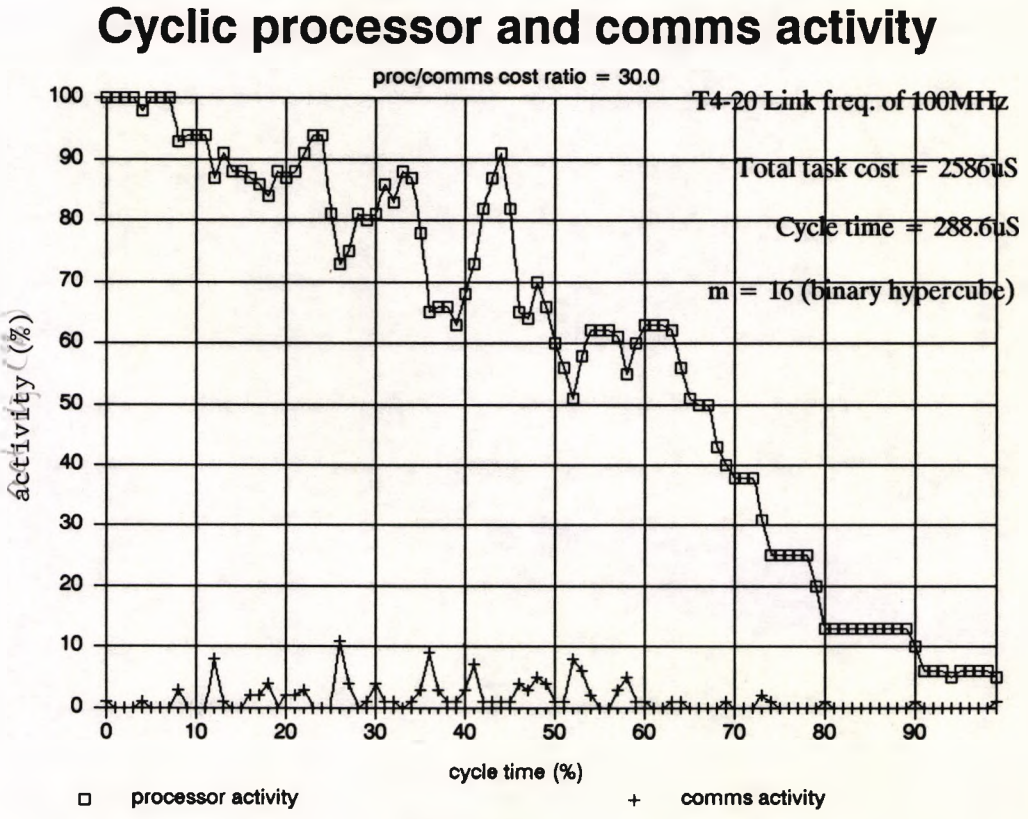


Figure 8.6

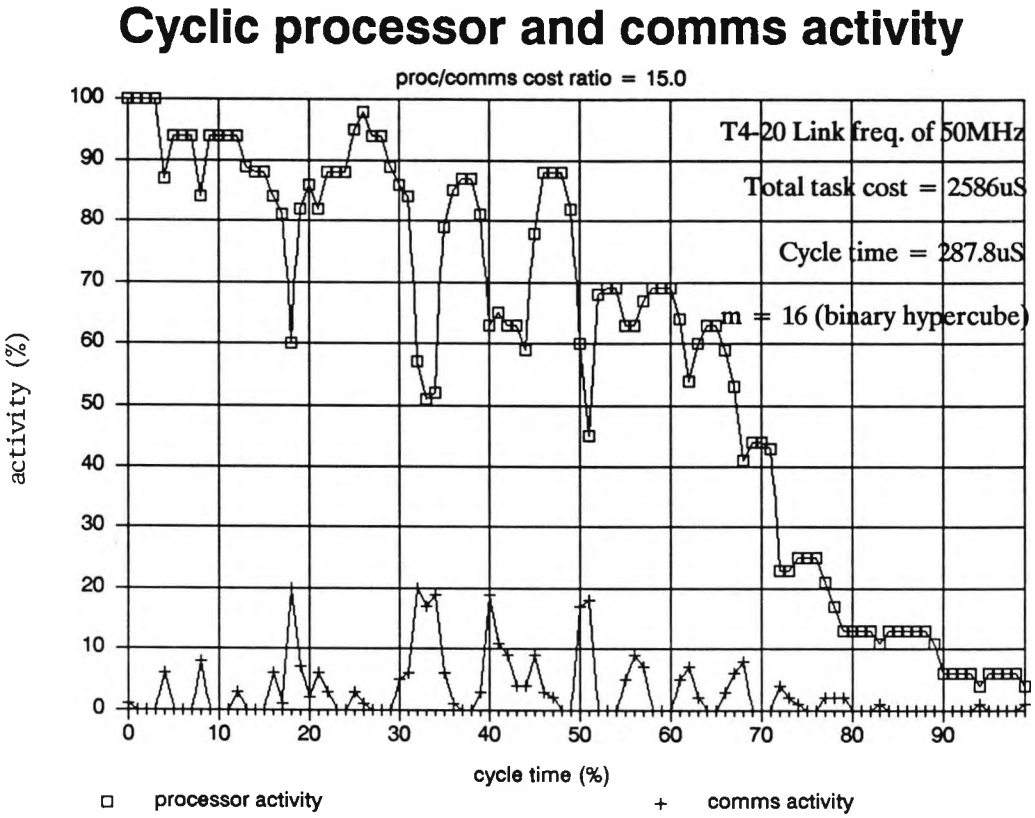


Figure 8.7

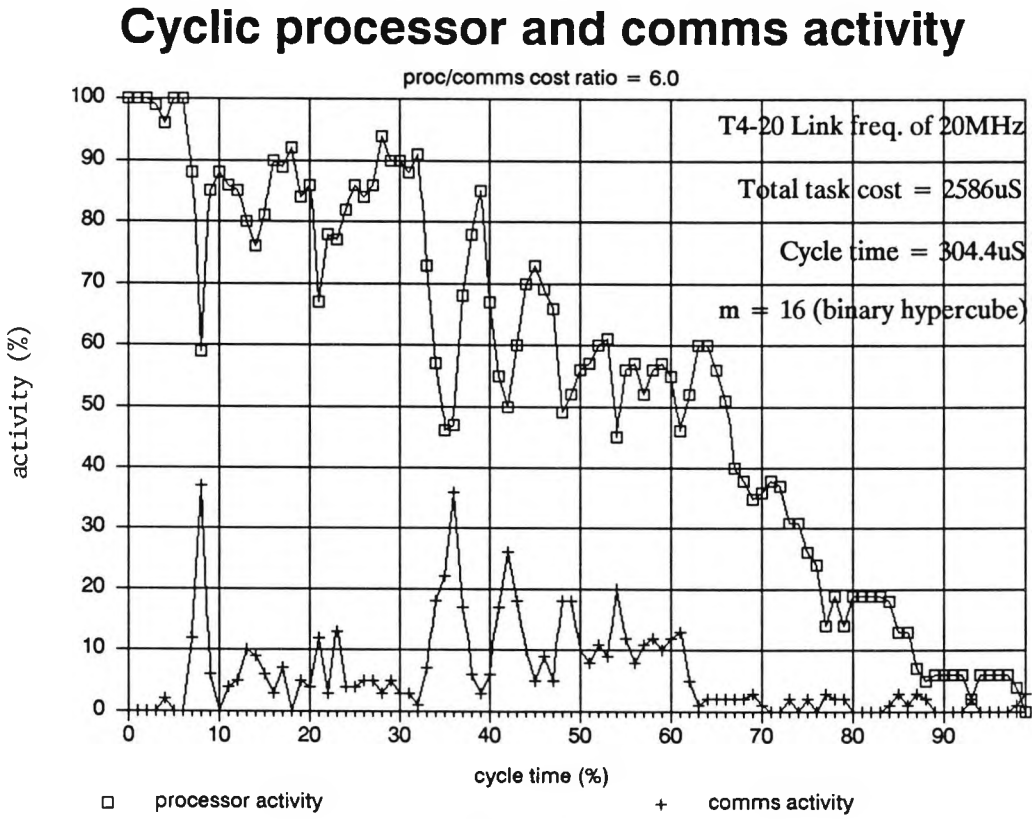


Figure 8.8

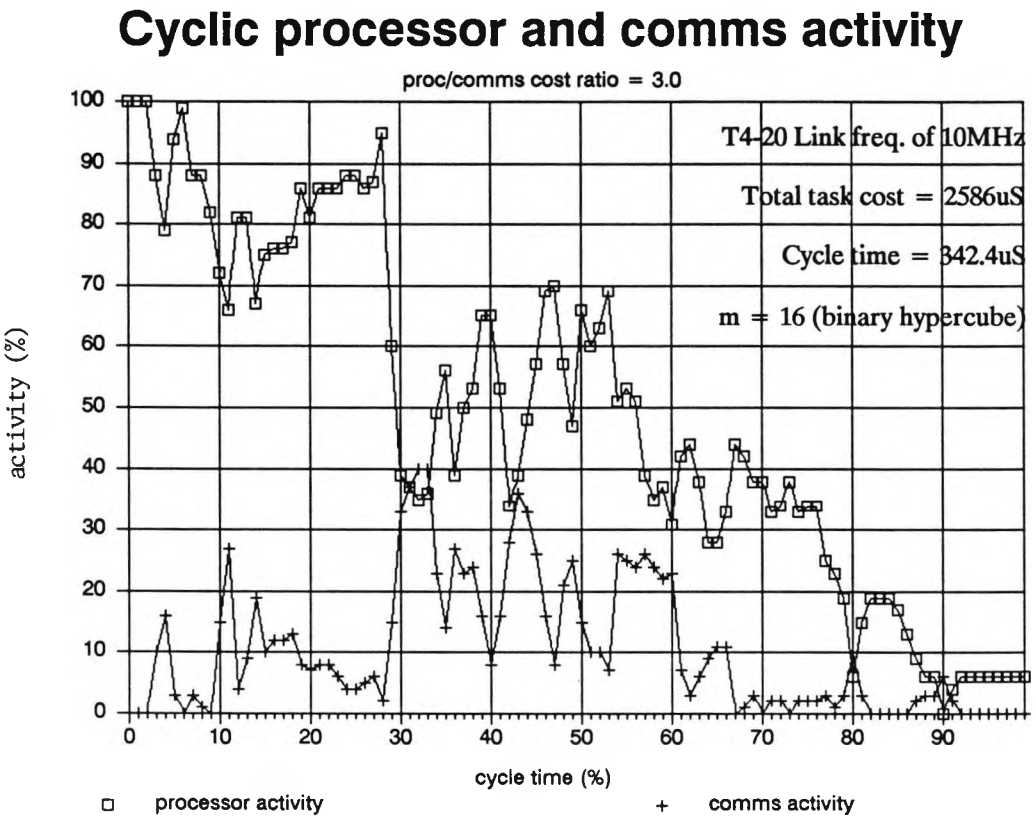


Figure 8.9

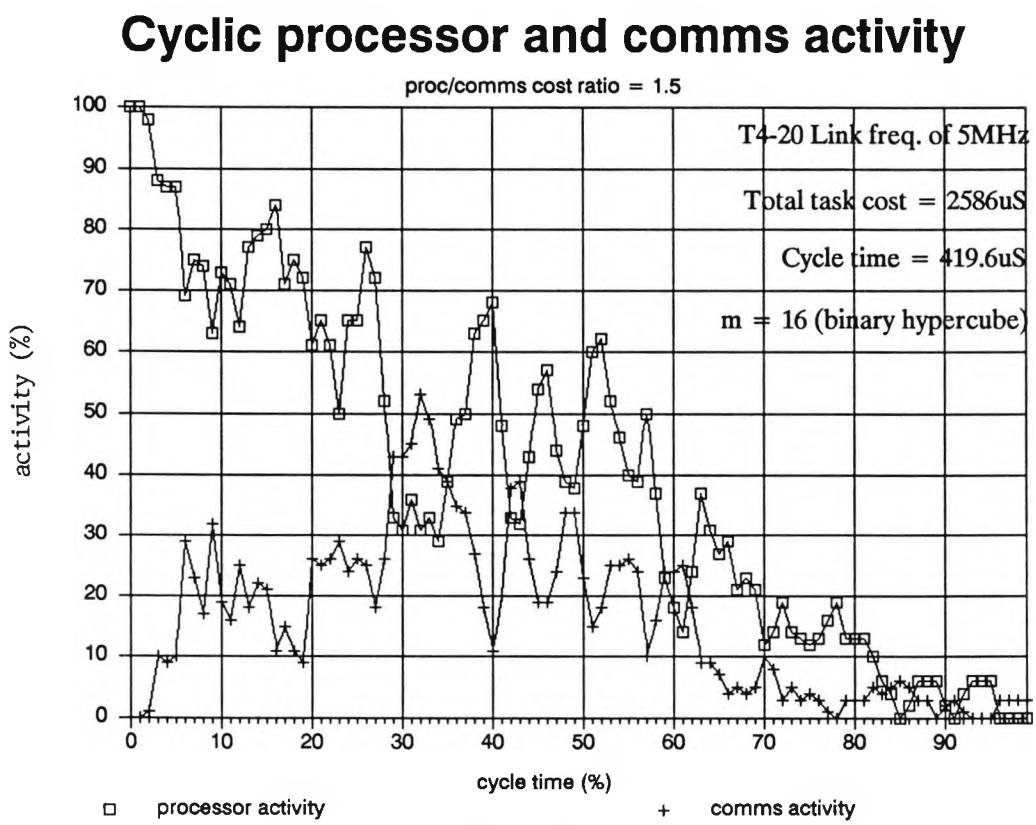


Figure 8.10

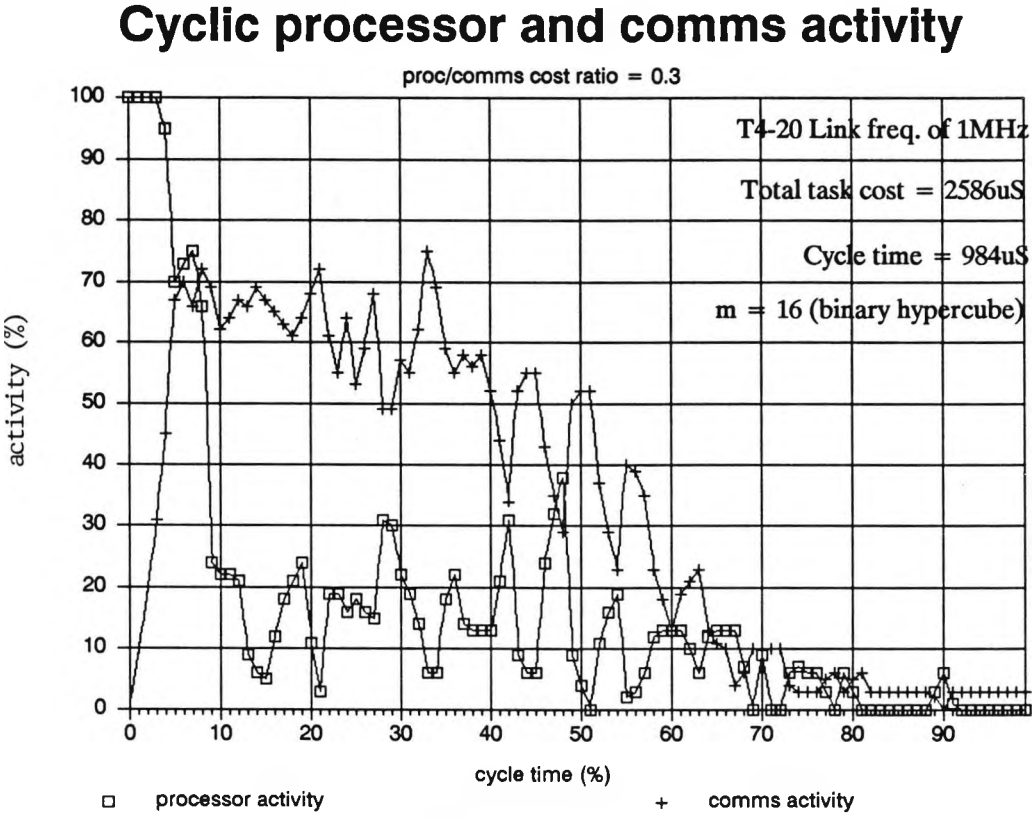


Figure 8.11

Figures 8.6 through to 8.11 suggest that the number of communications increase as the processor/communications ratio decreases. However, Figure 8.12 shows that the opposite is generally true. We can conclude, therefore, that the increase in communications use is attributable to an increase in communications cost and not the number of communications.

Figure 8.13 summarises the relation between processor/communication cost ratio and speedup. This plot illustrates that for ratios above a 10:1 speedup degradation is almost entirely due to synchronisation and latent scope effect. Below this ratio, degradation attributable to finite communication cost becomes significant; in the range 10:1 to 5:1 there is a noticeable down turn in speedup and below a ratio of 5:1 degradation becomes severe.

Both Figures 8.12 and 8.13 have a vertical scale that is normalised to the results for a processor/communications cost ratio of infinity.

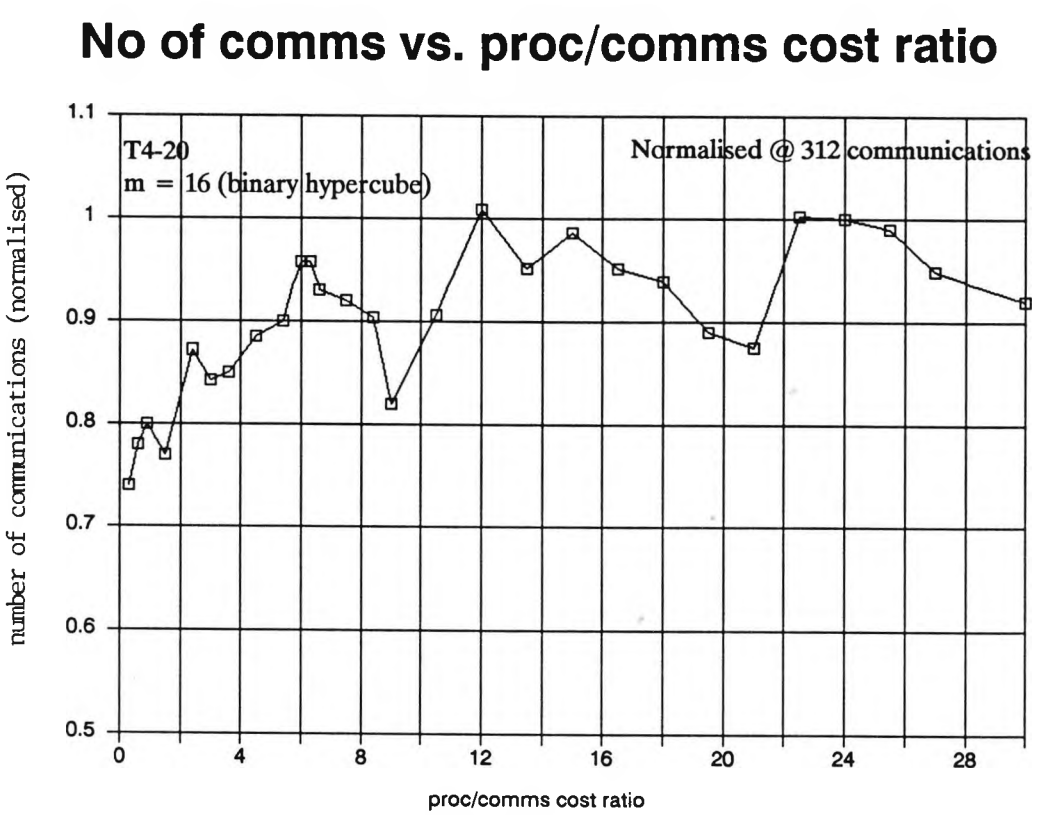


Figure 8.12

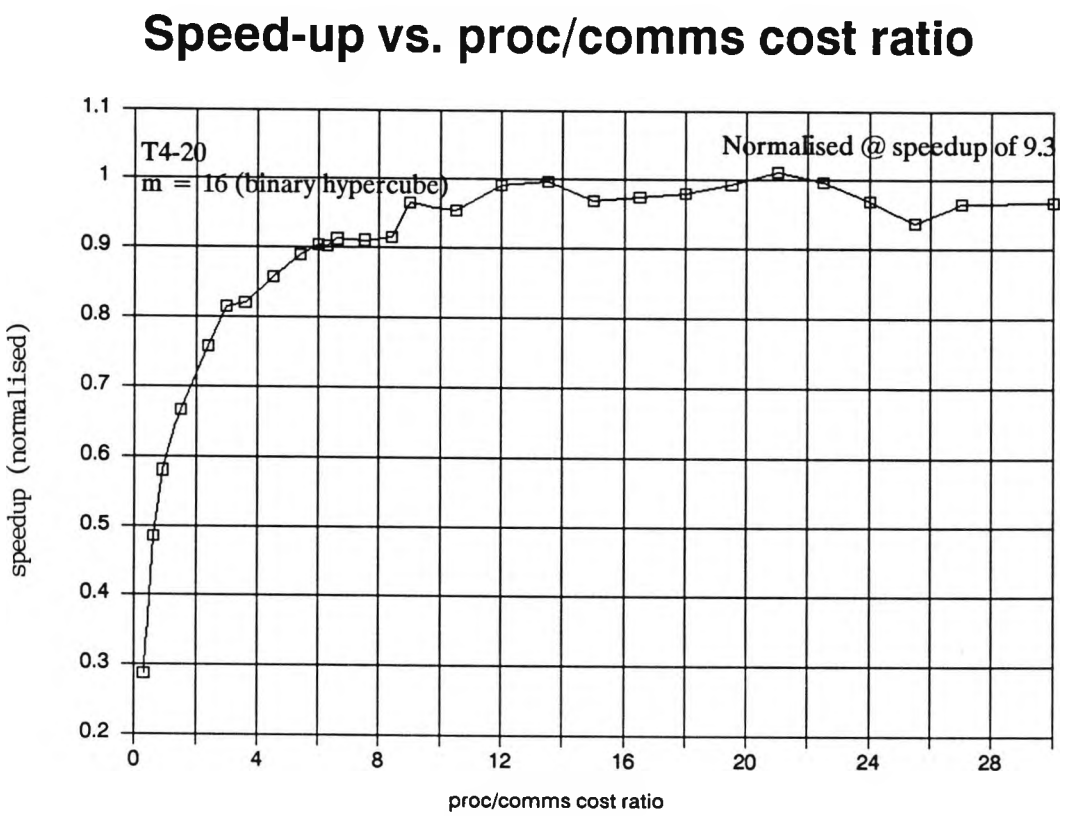


Figure 8.13

8.2.5 Topology effect

This section discusses the final effect of communication on performance, namely topology effect. Communication time between any two processors is affected by communication cost and can be affected by synchronisation and latent scope effect. The alternate number of independent communication channels into and out of a processor (degree) and the length of a communication path (diameter) between communicating processors, also have a bearing on performance.

For a specific topology the degree and diameter can be found, these figures are used in equation (8.7) to assign a merit factor, Q to a topology.

$$Q = (\overline{\text{degree}} / \overline{\text{diameter}}) * (m - 1)^{-1}, m > 1 \qquad (8.7)$$

Table 8.2 presents several different topologies, their mean degree, mean diameter and Q factor.

m	Topology	$\overline{\text{degree}}$	$\overline{\text{diameter}}$	Q	speedup	speedup(norm)
4	single ring	2.0	1.33	0.5	2.88	0.75
5	double ring	1.5	3.6	0.6	3.75	0.79
6	chordal ring	3.67	1.4	0.52	4.35	0.79
8	single linear	1.75	2.98	0.08	3.23	0.46
9	mesh	2.67	2.0	0.167	5.41	0.69
16	binary hypercube	3.875	2.13	0.12	8.65	0.71

Table 8.2 Topology factor

Figure 8.14 illustrates the relation between 1/Q and speedup. Speedup is normalised by dividing the actual speedup by the ideal speedup for m processors. The ideal value is not affected by any of the communications effects. The results include synchronisation and latent scope effects, which may vary from case to case.

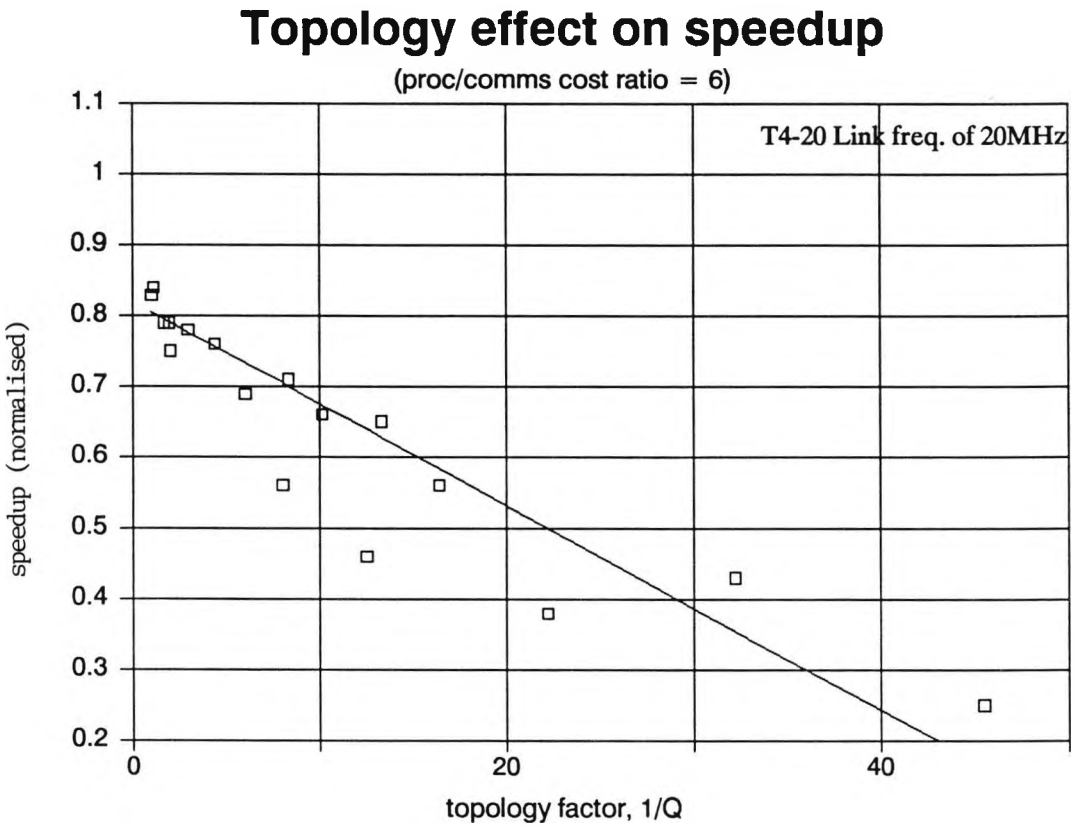


Figure 8.14

8.3 Examples

This section presents four algorithms that are commonly used in digital signal processing. These are:

(i) fir8: 8 tap finite impulse response filter
(ii) iir2: 2nd order infinite impulse response filter
(iii) wdf4: 4th order wave digital filter
(iv) fft8: 8 point complex fast Fourier transform

and one modified, multiple version of fir8 called fir8x8.

Each example is followed through from DFDL program description to Occam program translation. Results include pre-schedule program activity profiles, speedup figures for different numbers of processors and post-schedule activity profiles.

Pre-schedule activity profiles are generated from the critical path results. Two profiles are produced for each DFDL program; the first from the earliest times that tasks can be executed and the second from the latest times that tasks can be executed. These un-scheduled views of program execution ignore timing losses due to integer and communication effects. Both profiles indicate where potential parallelism exists and how many processors may be utilised. The cycle time of the two profiles is identical and is known as the critical path length, or minimum cycle time. Information from these profiles can be useful when analysing algorithms and choosing a value for m, since they give insight into the upper speedup bound, the likely maximum number of useful processors and critical path density.

Speedup results are presented graphically, plotted against number of processors, m. Results are all based on a T4-20 Transputer operating at a link frequency of 20MHz. Two bounds are included on the graph, a diagonal line "speedup = m" and a horizontal line "speedup = total task cost / minimum cycle time". For completeness the results for zero cost communication are also plotted.

Finally, post-schedule results are presented in the form of four superimposed activity profiles; processor, input, inter-processor communication and output activity. All are plotted against scheduled cycle time. These profiles show when resources are used, their use throughout the duration of the cycle and how different resources interact.

8.3.1 Finite impulse response filter

The DFDL program below, inputs a single value each cycle. This value, and the seven most recent inputs from previous cycles are multiplied by the coefficients. The eight products are summed and the result outputted. Input and output have a data type of INT32. This data type is different to arithmetic operations, which are of data type REAL32, and causes data type conversion to take place at the input and output.

```
PROG fir8(INPUT(INT32) in OUTPUT(INT32) out)

  NODE value[8]
  VALUE TABLE coeff[8]  IS  [  0.121, 0.146, -2.345, 0.5,
                               0.5, -2.345, 0.146, 0.121]

  BEGIN
    REPEAT FOREVER
      % Multiply delayed inputs by coefficients
      value[i FROM 0 FOR 8] : = coeff[i FROM 0 FOR 8] * ...
                                  in Z[i FROM 0 FOR 8]
      % Sum products
      out : = SUM(value[i FROM 0 FOR 8])
  END
```

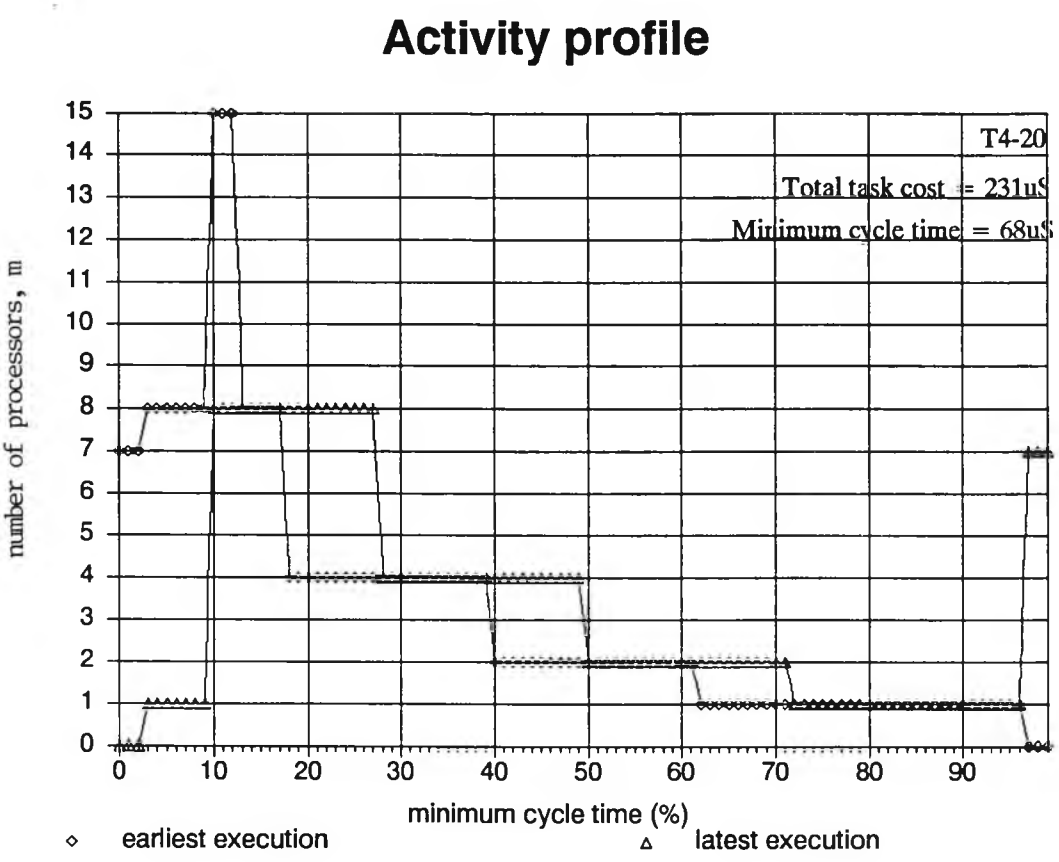


Figure 8.15

The earliest and latest profiles of the program suggest that using in excess of 15 processors would not aid performance. Lower, more practical estimates for m are suggested from the latest profile, which peaks at 8 processors, or from the quotient (total task cost / minimum cycle time) which is calculated to be 3.4.

Figure 8.16 illustrates the actual speedup when the program is scheduled onto an m processor network connected in a chordal ring topology. Speedup, for a link frequency of 20MHz, peaks at 2.18 when m equals 5. This gain represents 65% of the upper bound.

The scheduled results are translated into Occam for the case m equals 3. The Occam program is functionally identical to the DFDL program, however, it appears to be far more complex. This is in part due to communication between the three parallel procedures, but is also due to the fragmentation of spatial arrays by the DFDL compiler. Temporal arrays are generally left intact.

The composite characters {{{, }}} and ... are visible in both DFDL and Occam programs, these are editor constructs from the TDS folding editor, and have no syntatic or semantic effect on the programs. The {{{ construct indicates that a fold is open and marks the top of the open fold. A matching }}} construct marks the bottom of the open fold. The three full stops in succession ... show that a fold is closed and that text is hidden. Text that appears on the same line as {{{ or ... is a comment pertaining to the fold.

Note: The upper bound (Figure 8.16) defines the maximum speedup due to the algorithm's parallelism. This value is independent of m and is derived from the CPM, i.e., upper speedup bound = total task cost / critical path cost.

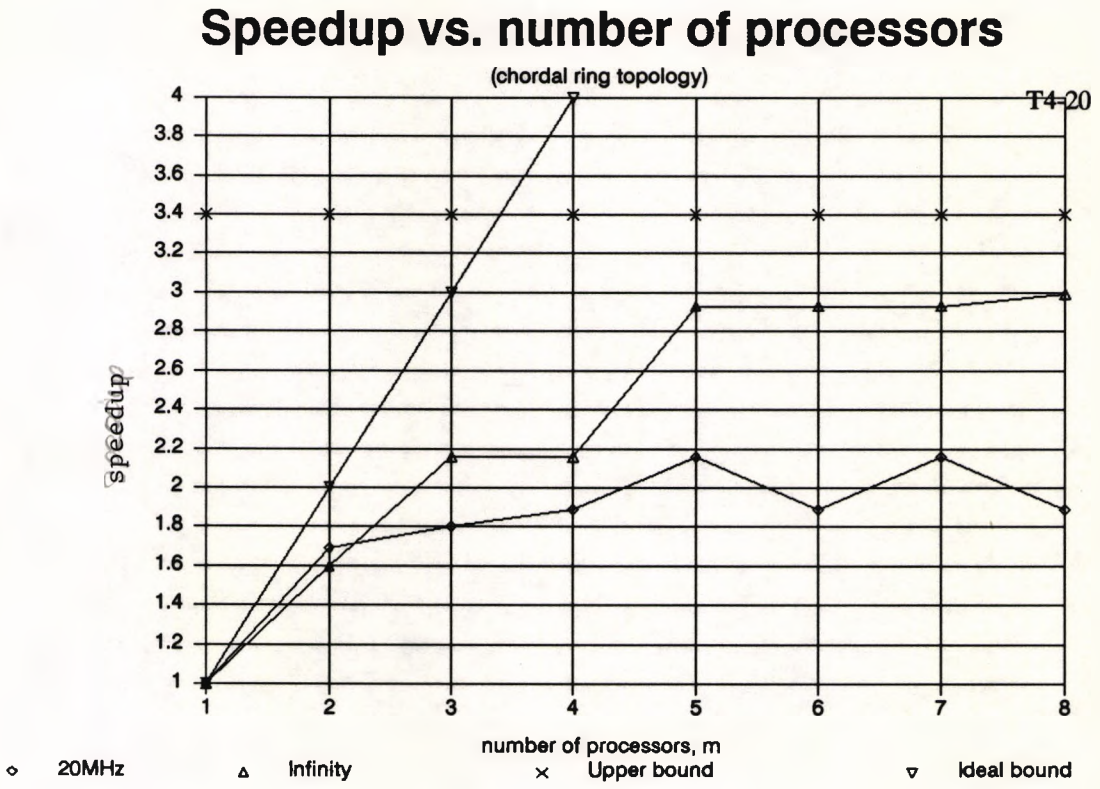


Figure 8.16

```
PROC fir8(CHAN OF INT32 in, out)
  {{{  CHANNELS
  CHAN OF REAL32 ch.real32.01.1, ch.real32.01.2, ch.real32.01.3,
ch.real32.02.3:
  CHAN OF ANY ch.any.02.2:
  }}}
  {{{  PROCEDURES

  {{{  fir801
  PROC fir801(CHAN OF INT32 ch.1,
              CHAN OF REAL32 ch.2, ch.3, ch.4)

    VAL coeff.0 IS 0.0380601995(REAL32):
    VAL coeff.6 IS 0.961939692(REAL32):
    VAL coeff.5 IS 0.853553295(REAL32):
    VAL coeff.7 IS 1.0(REAL32):

    INT32 in:
    REAL32 value.0:
    REAL32 value.6:
    REAL32 value.5:
    REAL32 value.7:
    REAL32 aAa.real32:
    REAL32 aAb.real32:
    INT INDEX.0:
    [7]REAL32 DELAY.0:

    SEQ
      SEQ i = 0 FOR 7
        DELAY.0[i] := 0.0(REAL32)
      INDEX.0 := 0

      WHILE TRUE
        SEQ
          ch.2 ! DELAY.0[(INDEX.0 + 3) REM 7]
          ch.3 ! DELAY.0[(INDEX.0 + 0) REM 7]
          ch.4 ! DELAY.0[(INDEX.0 + 1) REM 7]
          ch.1 ? in
          aAa.real32 := REAL32 ROUND in
          aAb.real32 := coeff.0 * aAa.real32
          value.0 := aAb.real32
          ch.2 ! value.0
          aAb.real32 := coeff.6 * DELAY.0[(INDEX.0 + 5) REM 7]
          ch.3 ! DELAY.0[(INDEX.0 + 2) REM 7]
          value.6 := aAb.real32
          aAb.real32 := coeff.5 * DELAY.0[(INDEX.0 + 4) REM 7]
          value.5 := aAb.real32
          ch.2 ! value.5
          aAb.real32 := coeff.7 * DELAY.0[(INDEX.0 + 6) REM 7]
          value.7 := aAb.real32
          aAb.real32 := value.6 + value.7
          ch.2 ! aAb.real32
          INDEX.0 := (INDEX.0 + 6) REM 7
          DELAY.0[INDEX.0] := aAa.real32
```

```
:
}}}
{{{  fir802
PROC fir802(CHAN OF REAL32 ch.1, ch.2,
            CHAN OF ANY ch.3,
            CHAN OF REAL32 ch.4)

  VAL coeff.1 IS 0.146446601(REAL32):
  VAL coeff.2 IS 0.308658212(REAL32):
  VAL coeff.3 IS 0.5(REAL32):

  REAL32 value.1:
  REAL32 value.2:
  REAL32 value.3:
  INT32 aAa.int32:
  REAL32 aAa.real32:
  REAL32 aAb.real32:

  SEQ
    WHILE TRUE
      SEQ
        PAR
          ch.2 ? aAa.real32
          ch.1 ? aAb.real32
        aAa.real32 := coeff.1 * aAa.real32
        value.1 := aAa.real32
        ch.3 ! value.1
        aAa.real32 := coeff.2 * aAb.real32
        value.2 := aAa.real32
        ch.2 ? aAa.real32
        aAa.real32 := coeff.3 * aAa.real32
        value.3 := aAa.real32
        aAa.real32 := value.2 + value.3
        ch.4 ? aAb.real32
        aAa.real32 := aAb.real32 + aAa.real32
        ch.4 ? aAb.real32
        aAa.real32 := aAa.real32 + aAb.real32
        aAa.int32 := INT32 ROUND aAa.real32
        ch.3 ! aAa.int32
```

```
:
}}}
{{{  fir803
PROC fir803(CHAN OF REAL32 ch.1,
            CHAN OF ANY ch.2,
            CHAN OF REAL32 ch.3,
            CHAN OF INT32 ch.4)

  VAL coeff.4 IS 0.691341698(REAL32):

  REAL32 value.4:
  INT32 aAa.int32:
  REAL32 aAa.real32:
  REAL32 aAb.real32:

  SEQ
    WHILE TRUE
      SEQ
        ch.3 ? aAa.real32
        aAa.real32 := coeff.4 * aAa.real32
        value.4 := aAa.real32
        ch.2 ? aAa.real32
        ch.3 ? aAb.real32
        aAa.real32 := aAb.real32 + aAa.real32
        ch.3 ? aAb.real32
        aAb.real32 := value.4 + aAb.real32
        ch.1 ! aAa.real32
        ch.3 ? aAa.real32
        aAa.real32 := aAb.real32 + aAa.real32
        ch.1 ! aAa.real32
        ch.2 ? aAa.int32
        ch.4 ! aAa.int32
:
}}}
}}}

PAR
  fir801(in, ch.real32.01.1, ch.real32.01.2, ch.real32.01.3)
  fir802(ch.real32.01.3, ch.real32.01.2, ch.any.02.2, ch.real32.02.3)
  fir803(ch.real32.02.3, ch.any.02.2, ch.real32.01.1, out)
```

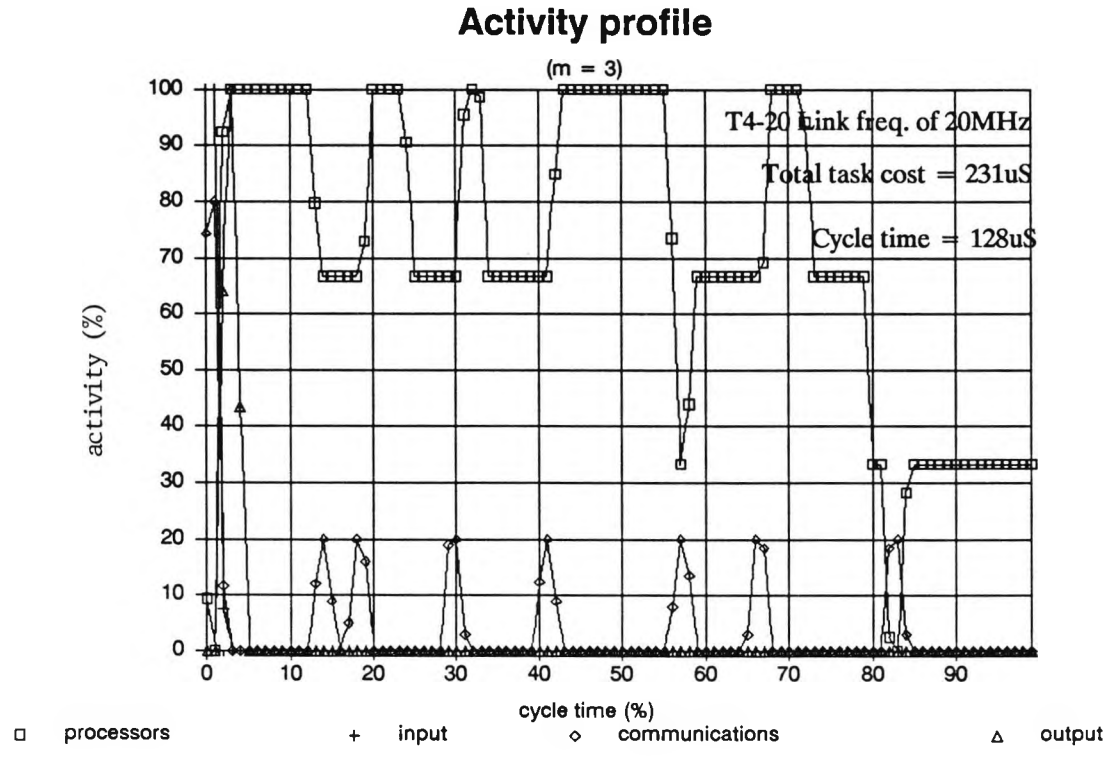


Figure 8.17

Figure 8.17 illustrates scheduled activity for m equals 3. The plot reveals that there is significant under use of the processors in the latter part of the cycle. Attempts to improve efficiency and make use of this slack are covered in section 8.3.5. Another observation is that output appears to occur immediately after data is inputted, it should be noted that this is the output from the previous cycle. The output for the current cycle is slightly to the right of the right hand edge of the graph, but since the x-axis represents cycle time, then the extreme ends of the graph are effectively joined as far as cycle activity is concerned.

8.3.2 Infinite impulse response (bi-quad) filter

The DFDL program below, inputs and outputs a single value each cycle. This input is multiplied by a scaling factor to give an intermediate value. A second intermediate value is produced by summing the scaled input and two delayed and scaled versions of this intermediate value; delayed by one and two cycles respectively. The output is derived in a similar fashion, but is the sum of the second intermediate value, and two delayed and scaled versions of this value, once again delayed by one and two cycles. Input and output have a data type of INT16. This data type is different to arithmetic operations, which are of data type REAL32, and causes data type conversion to take place at the input and output.

```
PROG iir2(INPUT(INT16) in OUTPUT(INT16) out)
  NODE value1, value2
  VALUE scale IS 0.684561
  VALUE TABLE coeff.a[2] IS [ 0.444566,  0.675343]
  VALUE TABLE coeff.b[2] IS [-0.684389, -0.475112]
  BEGIN
    REPEAT FOR 100
      value1 :=  scale * in
      value2 := ((coeff.b[0] * value2 Z[1]) + ...
        (value2 Z[2] * coeff.b[1])) + value1
      out := value2 + ((coeff.a[0] * value2 Z[1]) + ...
        (value2 Z[2] * coeff.a[1]))
  END
```

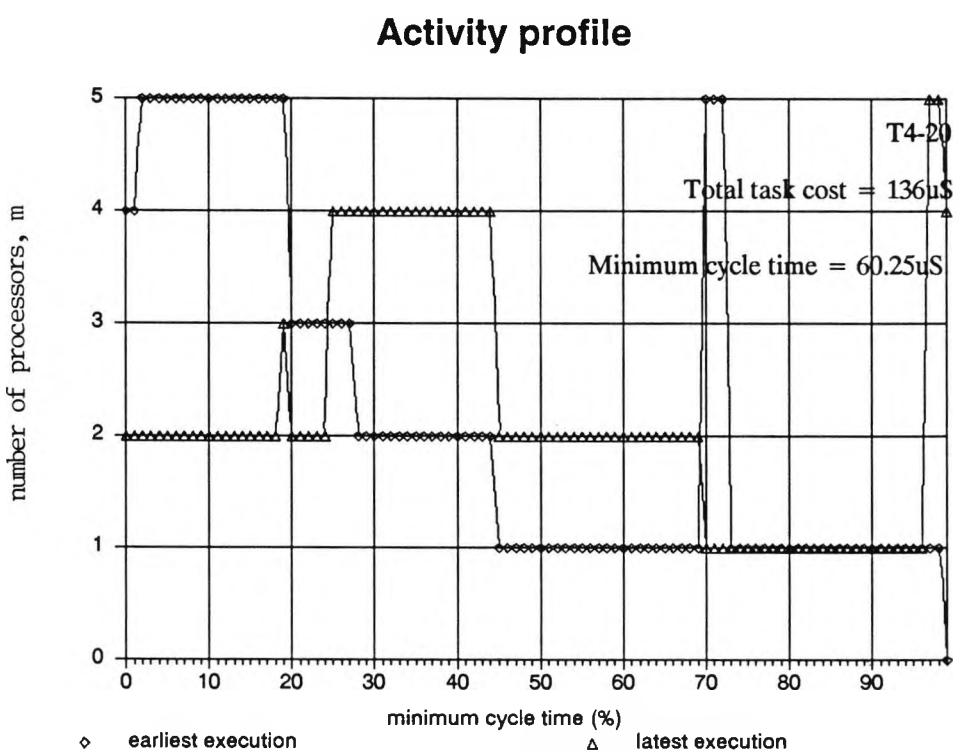


Figure 8.18

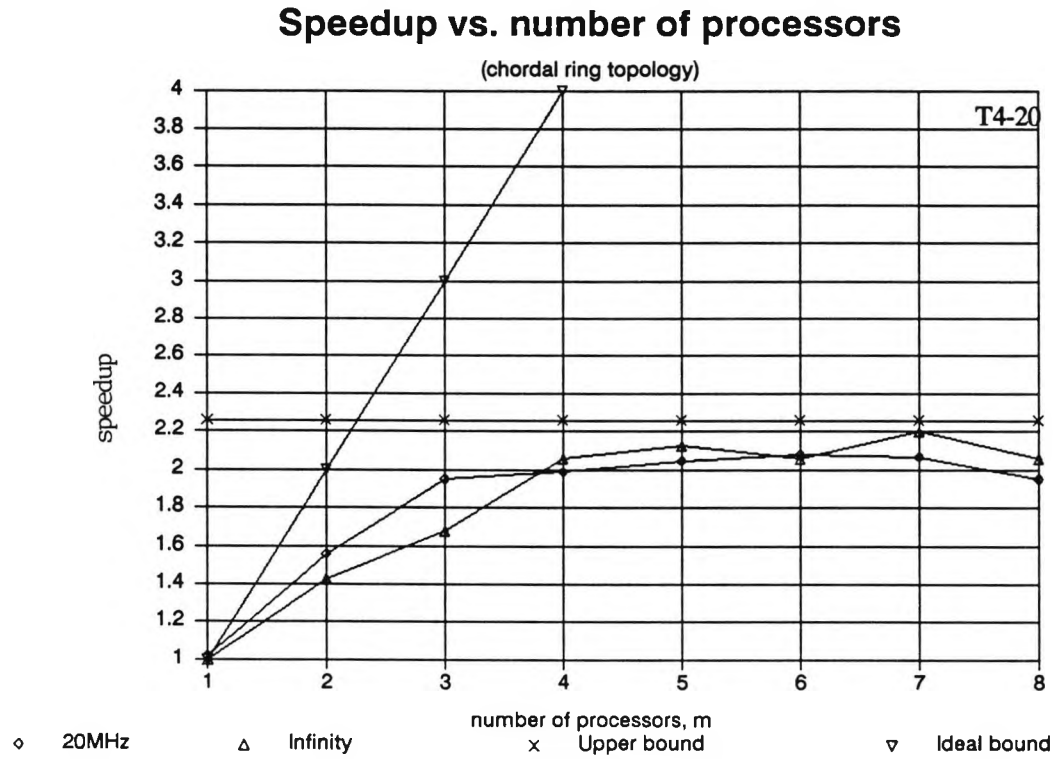


Figure 8.19

The earliest and latest profiles of the program suggest that using in excess of 5 processors would not aid performance. A lower estimate for m is suggested from the quotient (total task cost / minimum cycle time) which is calculated to be 2.26.

Figure 8.19 illustrates the actual speedup when the program is scheduled onto an m processor network connected in a chordal ring topology. Speedup, for a link frequency of 20MHz, peaks at 2.08 when m equals 6. This gain represents 92% of the upper bound.

Scheduled results are translated into Occam for the case m equals 2. The Occam program comprises a channel declaration, two procedures and a parallel process that calls the two procedures. Each procedure consists of declared values, declared variables, initialisation and a repetitive main body. In common with the source DFDL program, the main body of both procedures is repeated 100 times. The second procedure iir202 shows clearly that the DFDL compiler takes advantage of parallelism between individual communication channels and processor.

Where possible, values and variables are given names that originate from the DFDL source program. When the source name originates from an array the index is appended to the name to distinguish between different elements from the same array. When it is not possible to use a name from the source program then one is generated automatically. Automatically generated names comprise a head and a tail. The head consists of three letters, ranging from aAa to zZz. The combination of upper and lower case letters prevents duplication of reserved words or names originating from the source program. The tail part of the name identifies the data type. Variables with automatically generated names are re-used once the data they contain becomes redundant.

```
PROC iir2(CHAN OF INT16 in, out)
  {{{  CHANNELS
  CHAN OF REAL32 ch.real32.01.1, ch.real32.01.2, ch.real32.01.3:
  }}}
  {{{  PROCEDURES

  {{{  iir201
  PROC iir201(CHAN OF INT16 ch.1,
              CHAN OF REAL32 ch.2, ch.3, ch.4)

    VAL coeff.b.1 IS -0.475111991(REAL32):
    VAL scale IS 0.684561014(REAL32):

    INT16 in:
    REAL32 value1:
    REAL32 value2:
    REAL32 aAa.real32:
    INT INDEX.0:
    [2]REAL32 DELAY.0:

    SEQ
      SEQ i = 0 FOR 2
        DELAY.0[i] := 0.0(REAL32)
      INDEX.0 := 0

      SEQ i = 0 FOR 100
        SEQ
          aAa.real32 := DELAY.0[(INDEX.0 + 1) REM 2] * coeff.b.1
          ch.2 ! aAa.real32
          ch.3 ! DELAY.0[(INDEX.0 + 1) REM 2]
          ch.4 ! DELAY.0[(INDEX.0 + 0) REM 2]
          ch.1 ? in
          aAa.real32 := REAL32 ROUND (INT32 in)
          aAa.real32 := scale * aAa.real32
          value1 := aAa.real32
          ch.4 ? aAa.real32
          aAa.real32 := aAa.real32 + value1
          value2 := aAa.real32
          ch.2 ! value2
          ch.3 ! value2
          INDEX.0 := (INDEX.0 + 1) REM 2
          DELAY.0[INDEX.0] := value2
  :
  }}}
```

```
{{{  iir202
PROC iir202(CHAN OF REAL32 ch.1, ch.2, ch.3,
            CHAN OF INT16 ch.4)

  VAL coeff.b.0 IS -0.684388995(REAL32):
  VAL coeff.a.1 IS 0.675342977(REAL32):
  VAL coeff.a.0 IS 0.444566011(REAL32):

  INT16 aAa.int16:
  REAL32 aAa.real32:
  REAL32 aAb.real32:
  REAL32 aAc.real32:
  REAL32 aAd.real32:
  REAL32 DELAY.0:

  SEQ
    DELAY.0 := 0.0(REAL32)

    SEQ i = 0 FOR 100
      SEQ
        aAa.real32 := coeff.b.0 * DELAY.0
        PAR
          PAR
            ch.3 ? aAb.real32
            ch.2 ? aAc.real32
          ch.1 ? aAd.real32
        aAa.real32 := aAa.real32 + aAb.real32
        ch.1 ! aAa.real32
        aAa.real32 := aAc.real32 * coeff.a.1
        aAb.real32 := coeff.a.0 * aAd.real32
        PAR
          aAc.real32 := aAb.real32 + aAa.real32
          ch.3 ? aAd.real32
        PAR
          aAa.real32 := aAd.real32 + aAc.real32
          ch.2 ? aAb.real32
        aAa.int16 := INT16 ROUND aAa.real32
        ch.4 ! aAa.int16
        DELAY.0 := aAb.real32
:
}}}
}}}

PAR
  iir201(in, ch.real32.01.1, ch.real32.01.2, ch.real32.01.3)
  iir202(ch.real32.01.3, ch.real32.01.2, ch.real32.01.1, out)
:
```

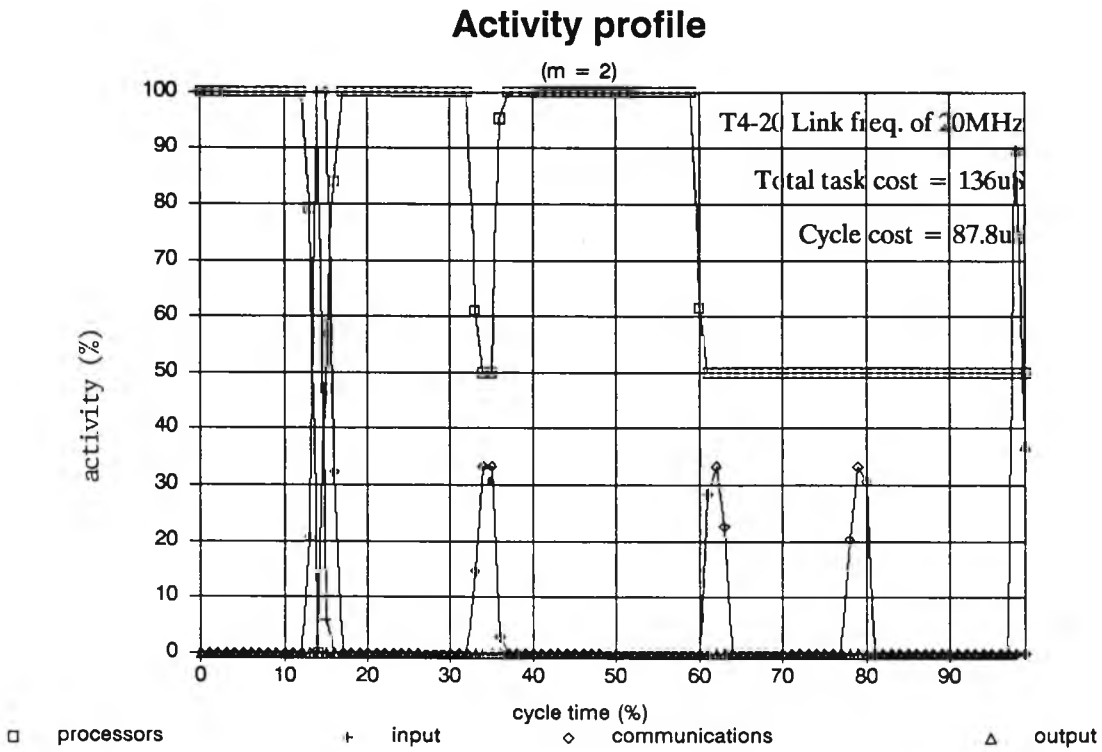


Figure 8.20

Figure 8.20 illustrates scheduled activity for 2 processors. One interesting observation is that a significant amount of processing is performed prior to input. This early processing is carried out in preference to input because the critical path passes through the feedback path, and the feedback path provides executable operands from the start of the cycle.

The 2 processor schedule results in a speedup of 1.55. This figure represents 76% of ideal speedup.

8.3.3 Wave digital filter

The third example is of a wave digital filter. This filter has a lattice structure which comprises four structurally identical sections. Each section has two inputs and two outputs, these are connected to adjacent sections, except at the ends, where they are taken to the input and output of the algorithm. Each section is described by the two difference equations

$$Y_1 := X_1 + (k * (X_1 - X_2Z^{-1}))$$
$$Y_2 := X_2Z^{-1} + (k * (X_1 - X_2Z^{-1}))$$

where Y_1 and Y_2 are outputs and X_1 and X_2 are inputs.

```
PROG wdf.4( INPUT(REAL32) in.x  OUTPUT(REAL32) out.x
            INPUT(REAL32) in.z  OUTPUT(REAL32) out.z)
  {{{ VALS
  NODE node.w[4], node.z[4], node.x[4]
  VALUE TABLE coeff[4] IS [0.668, 0.856, 0.386, 0.394]
  }}}
  BEGIN
    REPEAT FOREVER
      {{{ 1st stage
      % 1st stage
      node.x[0] := in.x
      node.w[0] := (node.z[1]Z[1] - node.x[0]) * coeff[0]
      node.z[0] := node.w[0] + node.z[1]Z[1]
      out.z := node.z[0]
      }}}
      {{{ 2nd stage
      % 2nd stage
      node.x[1] := node.w[0] + node.x[0]
      node.w[1] := (node.z[2]Z[1] - node.x[1]) * coeff[1]
      node.z[1] := node.w[1] + node.z[2]Z[1]
      }}}
      {{{ 3rd stage
      % 3rd stage
      node.x[2] := node.w[1] + node.x[1]
      node.w[2] := (node.z[3]Z[1] - node.x[2]) * coeff[2]
      node.z[2] := node.w[2] + node.z[3]Z[1]
      }}}
      {{{ 4th stage
      % 4th stage
      node.x[3] := node.w[2] + node.x[2]
      node.w[3] := (in.z - node.x[3]) * coeff[3]
      node.z[3] := node.w[3] + in.z
      out.x := (node.w[3] + node.x[3])
      }}}
  END
```

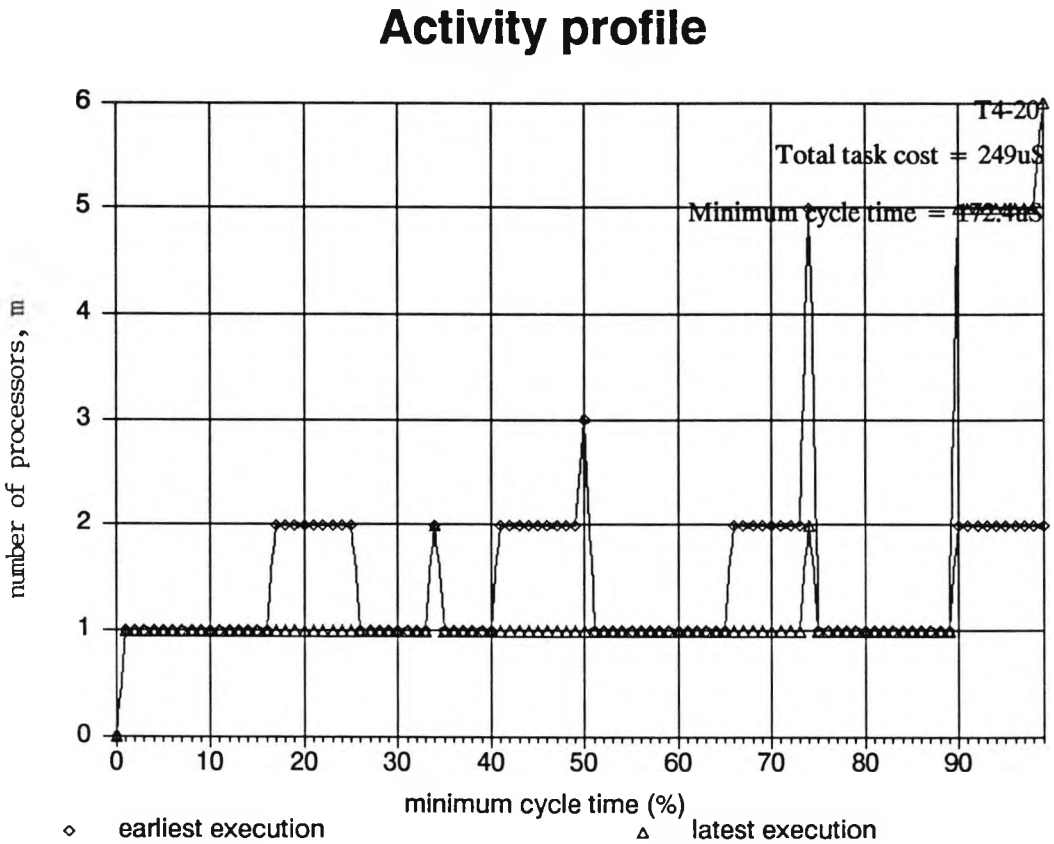


Figure 8.21

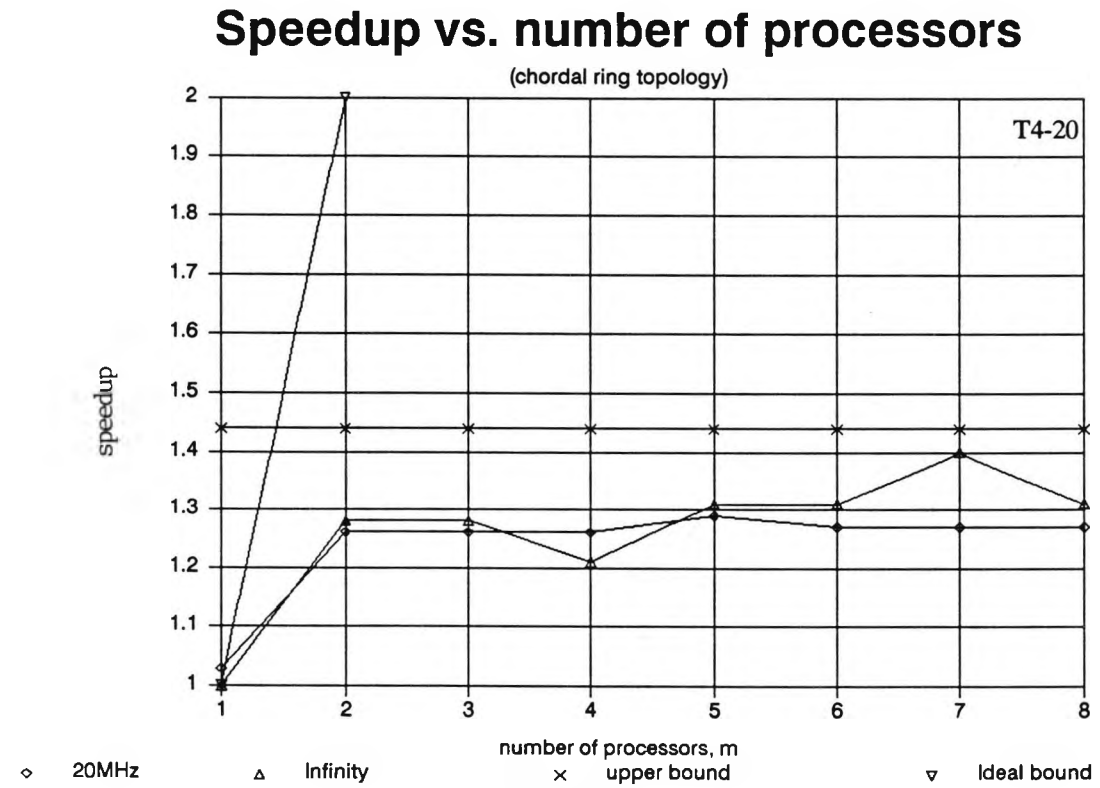


Figure 8.22

```
PROC wdf(CHAN OF REAL32 in.x, in.z, out.z, out.x)
  ...  CHANNELS
  {{{  PROCEDURES
  {{{  wdf01
  PROC wdf01(CHAN OF REAL32 ch.1, ch.2, ch.3, ch.4)
    ... VALS
    ... VARS
    SEQ
      ... INIT
      WHILE TRUE
        SEQ
          ch.3 ! DELAY.1
          ch.4 ! DELAY.3
          PAR
            ch.1 ? in.x
            ch.2 ? in.z
          node.x.0 := in.x
          aAa.real32 := DELAY.2 - node.x.0
          ch.3 ! DELAY.0
          aAa.real32 := aAa.real32 * coeff.0
          node.w.0 := aAa.real32
          aAa.real32 := node.w.0 + node.x.0
          ch.3 ! node.w.0
          node.x.1 := aAa.real32
          aAa.real32 := DELAY.4 - node.x.1
          aAa.real32 := aAa.real32 * coeff.1
          node.w.1 := aAa.real32
          aAa.real32 := node.w.1 + node.x.1
          ch.3 ! node.w.1
          node.x.2 := aAa.real32
          aAa.real32 := DELAY.0 - node.x.2
          aAa.real32 := aAa.real32 * coeff.2
          node.w.2 := aAa.real32
          PAR
            PAR
              SEQ
                aAa.real32 := node.w.2 + node.x.2
                ch.3 ! node.w.2
              ch.4 ? aAb.real32
            ch.3 ? aAc.real32
          node.x.3 := aAa.real32
          aAa.real32 := in.z - node.x.3
          aAa.real32 := aAa.real32 * coeff.3
          node.w.3 := aAa.real32
          PAR
            PAR
              aAa.real32 := node.w.3 + node.x.3
              ch.4 ? aAd.real32
            ch.3 ? aAe.real32
          aAf.real32 := node.w.3 + in.z
          ch.3 ! aAa.real32
          node.z.3 := aAf.real32
          DELAY.1 := aAb.real32
          DELAY.4 := aAe.real32
          DELAY.3 := aAd.real32
          DELAY.2 := aAc.real32
          DELAY.0 := node.z.3
  :
  }}}
```

```
{{{  wdf02
PROC wdf02(CHAN OF REAL32 ch.1, ch.2, ch.3, ch.4)
  ... VARS
  SEQ
    WHILE TRUE
      SEQ
        PAR
          ch.2 ? aAa.real32
          ch.1 ? aAb.real32
        ch.2 ? aAc.real32
        ch.2 ? aAd.real32
        aAa.real32 := aAd.real32 + aAa.real32
        node.z.0 := aAa.real32
        ch.3 ! node.z.0
        ch.2 ? aAa.real32
        aAa.real32 := aAa.real32 + aAb.real32
        node.z.1 := aAa.real32
        ch.2 ! node.z.1
        ch.1 ! node.z.1
        ch.2 ? aAa.real32
        aAa.real32 := aAa.real32 + aAc.real32
        node.z.2 := aAa.real32
        ch.2 ! node.z.2
        ch.1 ! node.z.2
        ch.2 ? aAa.real32
        ch.4 ! aAa.real32
:
}}}
}}}

PAR
  wdf01(in.x, in.z, ch.real32.01.2, ch.real32.01.3)
  wdf02(ch.real32.01.3, ch.real32.01.2, out.z, out.x)
```

The latest program profile (Figure 8.21) suggests that using in excess of 8 processors would not aid performance. A lower estimate for m is suggested by the earliest profile, 5 processors, and the quotient (total task cost / minimum cycle time) which is calculated to be 1.44.

Figure 8.22 illustrates the actual speedup when the program is scheduled onto an m processor network connected in a chordal ring topology. Speedup, for a link frequency of 20MHz, peaks at 1.28 when m equals 5. This gain represents 89% of the upper bound. However, speedup is reasonably constant for m greater or equal to 2.

Scheduled results are translated into Occam for the case m equals 2. The Occam program comprises a channel declaration, two procedures and a parallel process that calls the two procedures. Each procedure consists of declared values, declared variables, initialisation and a repetitive main body. Only the main body is shown in detail. At the end of first procedure wdf01 the assignment of feedback variables is clearly visible, since the translator gives these variables the name DELAY.#

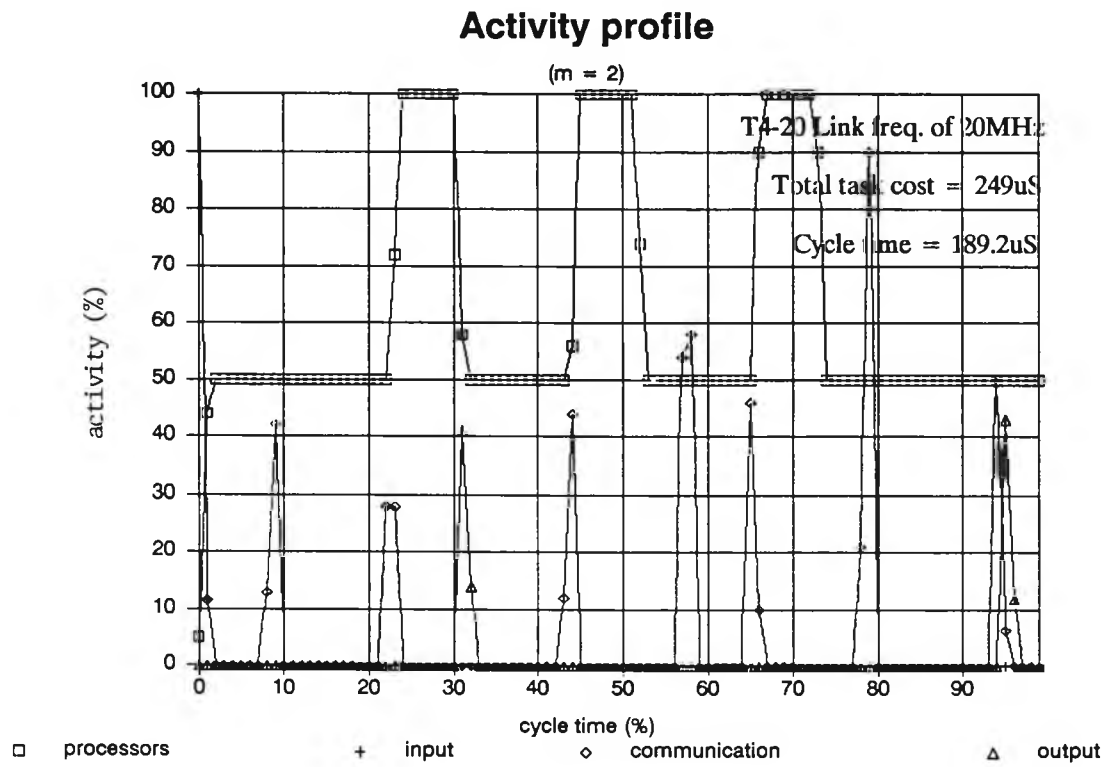


Figure 8.23

Figure 8.23 illustrates scheduled activity for 2 processors. One interesting observation is that of the two outputs, one appears much earlier in the cycle than does the other. The reason for this is that operands for the earlier output are from previous cycles (i.e. feedback) and these undergo only a single stage of processing, whereas the later output relies on the input data rippling through each section. The algorithm, in its current form, exhibits little parallelism and speedup would tend to remain low regardless of the number of sections used. One common method to increase parallelism is to treat the Z^{-1} delays as two $Z^{-1/2}$ delays and reorganise the distribution of delays so the ripple effect is reduced, this has the effect of reducing the critical path length.

8.3.4 Fast Fourier transform (decimation in time, radix 2)

The DFDL program for a complex fast Fourier transform is based around the complex butterfly algorithm. An example of the butterfly is illustrated in full below. The majority of the DFDL FFT program is concealed in folds, since each stage is structurally the same. When writing the FFT, no attempt was made to optimise the algorithm, e.g. removal of multiplication when coefficients are 1.0.

The program below illustrates that for every program cycle, two streams of 8 inputs are read into the algorithm and two streams of 8 values are outputted. The two input and two output streams consist of real and imaginary data. The real and imaginary input data is fed into an N/2 wide, $\log_2(N)$ deep array of butterfly sections (N = 8). Each input enters the array at an address that corresponds to the bit reversal of its index. This data shuffle is common to all in place FFT algorithms.

Examining Figure 8.24, the latest program profile suggests that using in excess of 24 processors would not aid performance. A lower estimate for m is suggested by the earliest profile, 16 processors, and the quotient (total task cost / minimum cycle time) which is calculated to be 10.71.

Figure 8.25 illustrates the actual speedup when the program is scheduled onto an m processor network connected in a chordal ring topology. Speedup, for a link frequency of 20MHz, peaks at 6.4 when m equals 13. This gain represents 60% of the upper bound. Speedup tends to level off above the region m equals 9.

```
PROG fft8(OUTPUT(REAL32) real.y[8]
          INPUT(REAL32) real.x[8], imag.x[8]
          OUTPUT (REAL32) imag.y[8])
  ... VALS
  BEGIN
    REPEAT FOREVER
      ... column 1
      {{{ column 2
      ... butterfly in.a[0][0], in.a[1][0], out.a[0][1], out.a[1][1]
      {{{ butterfly in.b[0][0], in.b[1][0], out.b[0][1], out.b[1][1]
      real.h[1][1] := (real.b[1][0] * real.cos[2]) - (imag.b[1][0] * imag.sin[2])
      imag.h[1][1] := (imag.b[1][0] * real.cos[2]) + (real.b[1][0] * imag.sin[2
      real.b[0][1] := real.b[0][0] + real.h[1][1]
      imag.b[0][1] := imag.b[0][0] + imag.h[1][1]
      real.b[1][1] := real.b[0][0] - real.h[1][1]
      imag.b[1][1] := imag.b[0][0] - imag.h[1][1]
      }}}
      ... butterfly in.a[2][0], in.a[3][0], out.a[2][1], out.a[3][1]
      ... butterfly in.b[2][0], in.b[3][0], out.b[2][1], out.b[3][1]
      }}}
      ... column 3
  END
```

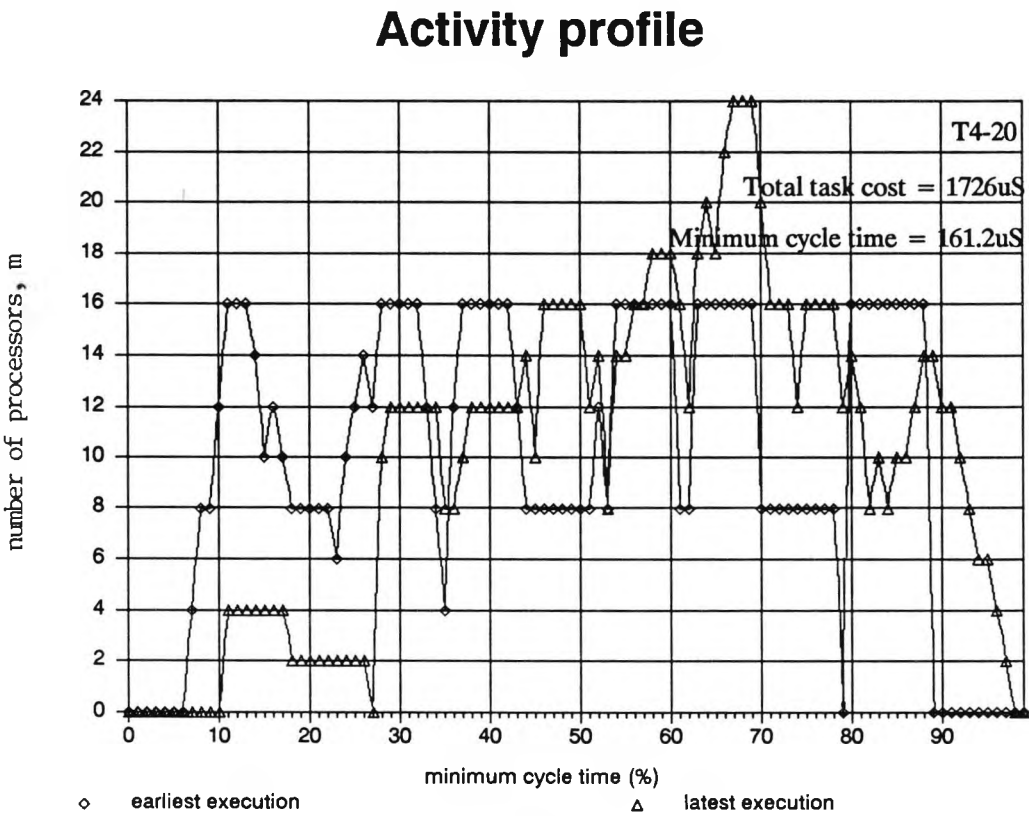


Figure 8.24

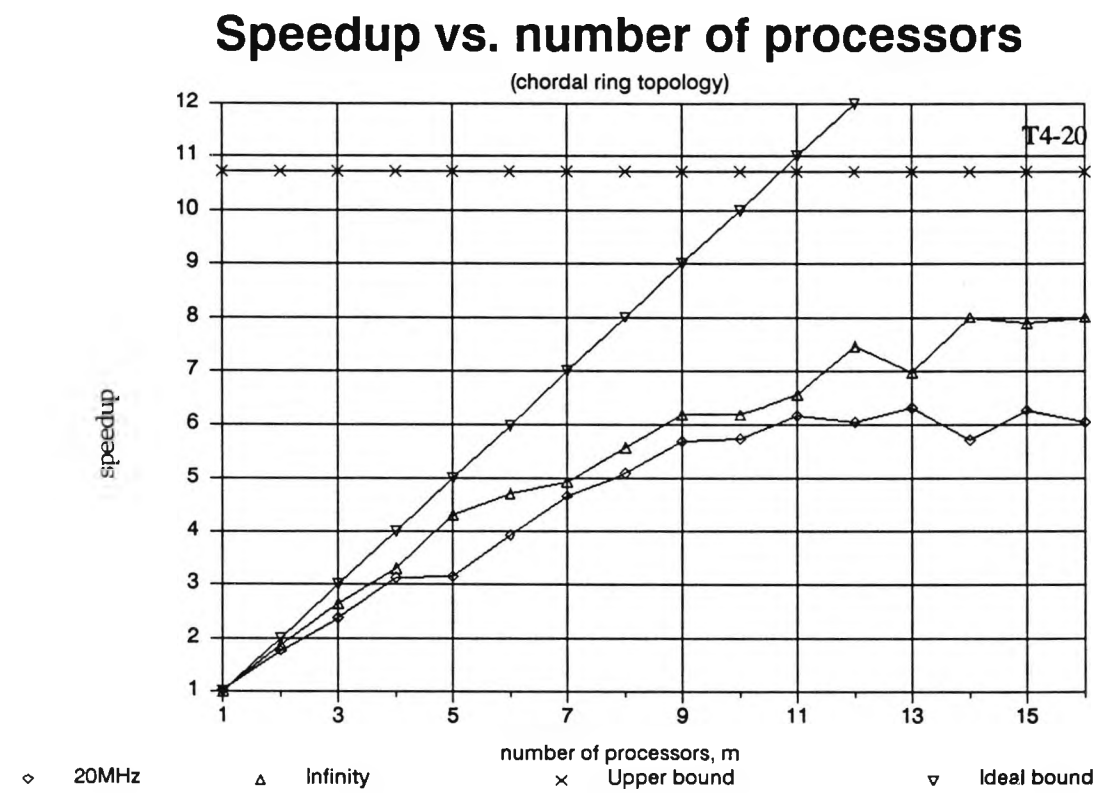


Figure 8.25

```
PROC fft8(CHAN OF REAL32 imag.x, real.x, real.y, imag.y)
  ...  CHANNELS
  {{{  PROCEDURES
  ...  fft801
  {{{  fft802
  PROC fft802(CHAN OF REAL32 ch.1, ch.2, ch.3, ch.4)
    ...  VALS
    ...  VARS
    SEQ
      WHILE TRUE
        SEQ
          ...  Page - 1
          ...  Page - 2
          ...  Page - 3
          {{{  Page - 4
          ch.1 ! real.a.2.1
          ch.3 ! real.b.2.0
          PAR
            ch.1 ? aAa.real32
            ch.3 ? aAb.real32
          ch.3 ! imag.h.2.0
          ch.4 ? aAe.real32
          aAe.real32 := aAe.real32 * real.cos.2
          aAc.real32 := aAe.real32 + aAc.real32
          imag.h.1.1 := aAc.real32
          aAc.real32 := real.b.2.0 - aAd.real32
          real.b.3.1 := aAc.real32
          aAc.real32 := real.b.3.1 * imag.sin.3
          aAd.real32 := real.b.3.1 * real.cos.3
          ch.3 ! imag.h.1.1
          ch.1 ! imag.h.1.1
          aAa.real32 := real.x.0 - aAa.real32
          real.b.0.0 := aAa.real32
          ch.3 ! real.b.0.0
          ch.1 ! real.b.0.0
          }}}
          ...  Page - 5
          ...  Page - 6
          ...  Page - 7
  :
  }}}
  ...  fft803
  ...  fft804
  }}}
  PAR
    fft801(ch.real32.01.0, imag.x, ch.real32.01.2, ch.real32.01.3)
    fft802(ch.real32.01.3, real.x, ch.real32.02.2, ch.real32.02.3)
    fft803(ch.real32.02.3, ch.real32.01.2, real.y, ch.real32.03.3)
    fft804(ch.real32.03.3, ch.real32.02.2, imag.y, ch.real32.01.0)
:
```

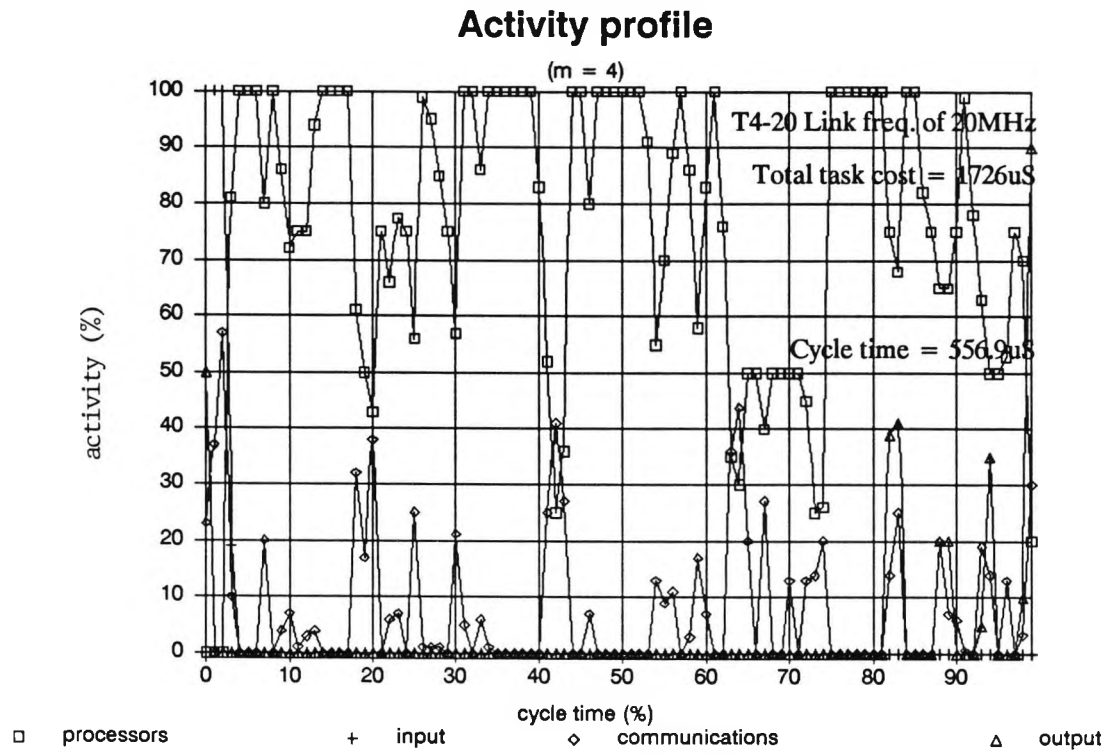


Figure 8.26

The Occam translation and Figure 8.26 illustrate scheduled activity for 4 processors. The majority of the program is hidden inside folds, since it is in total some 12 pages long. The results for m equals 4 is a speedup of 3.1 which represents 77.5% of ideal speedup. The activity schedule shows that the four processors have a reasonably high utility throughout the cycle time.

8.3.5 Multiple cycle finite impulse response filter

In the example that follows, the 8 tap FIR filter described in section 8.3.1 is re-written such that the algorithm describes a multiple number of cycles. Functionally, the two algorithms are similar, in that they produce the result

$$out_k := (c_0 * inZ^{-(k+0)}) + (c_1 * inZ^{-(k+1)}) + ... + (c_7 * inZ^{-(k+7)})$$

Fir8x8 describes 8 complete cycles, which allows the scheduler to exploit parallelism between successive cycles and the increased value n reduces integer effect

```
PROG fir8x8(INPUT(INT32) in[8] OUTPUT(INT32) out[8])
  NODE value[8][8]
  VALUE TABLE coeff[8] IS  [0.0380602, 0.1464466, 0.3086582, 0.5000000,
                            0.6913417, 0.8535533, 0.9619397, 1.0000000]
  BEGIN
    REPEAT FOREVER      % Multiply delayed inputs by coefficients
      value[0][0] := coeff[0] * in[0] % then sum the products
      value[i FROM 1 FOR 7][0] := coeff[i FROM 7 FOR 7 EVERY -1] * ...
        in[i FROM 1 FOR 7]Z[1]
      value[j FROM 0 FOR 2][1] := coeff[j FROM 1 FOR 2 EVERY -1] * ...
        in[j FROM 0 FOR 2]
      value[j FROM 2 FOR 6][1] := coeff[j FROM 7 FOR 6 EVERY -1] * ...
        in[j FROM 2 FOR 6]Z[1]
      value[k FROM 0 FOR 3][2] := coeff[k FROM 2 FOR 3 EVERY -1] * ...
        in[k FROM 0 FOR 3]
      value[k FROM 3 FOR 5][2] := coeff[k FROM 7 FOR 5 EVERY -1] * ...
        in[k FROM 3 FOR 5]Z[1]
      value[m FROM 0 FOR 4][3] := coeff[m FROM 3 FOR 4 EVERY -1] * ...
        in[m FROM 0 FOR 4]
      value[m FROM 4 FOR 4][3] := coeff[m FROM 7 FOR 4 EVERY -1] * ...
        in[m FROM 4 FOR 4]Z[1]
      value[n FROM 0 FOR 5][4] := coeff[n FROM 4 FOR 5 EVERY -1] * ...
        in[n FROM 0 FOR 5]
      value[n FROM 5 FOR 3][4] := coeff[n FROM 7 FOR 3 EVERY -1] * ...
        in[n FROM 5 FOR 3]Z[1]
      value[p FROM 0 FOR 6][5] := coeff[p FROM 5 FOR 6 EVERY -1] * ...
        in[p FROM 0 FOR 6]
      value[p FROM 6 FOR 2][5] := coeff[p FROM 7 FOR 2 EVERY -1] * ...
        in[p FROM 6 FOR 2]Z[1]
      value[q FROM 0 FOR 7][6] := coeff[q FROM 6 FOR 7 EVERY -1] * ...
        in[q FROM 0 FOR 7]
      value[7][6] := coeff[7] * in[7]Z[1]
      value[r FROM 0 FOR 8][7] := coeff[r FROM 7 FOR 8 EVERY -1] * ...
        in[r FROM 0 FOR 8]
      out[h FROM 0 FOR 8] := ...
        SUM(value[i FROM 0 FOR 8][h FROM 0 FOR 8])
  END
```

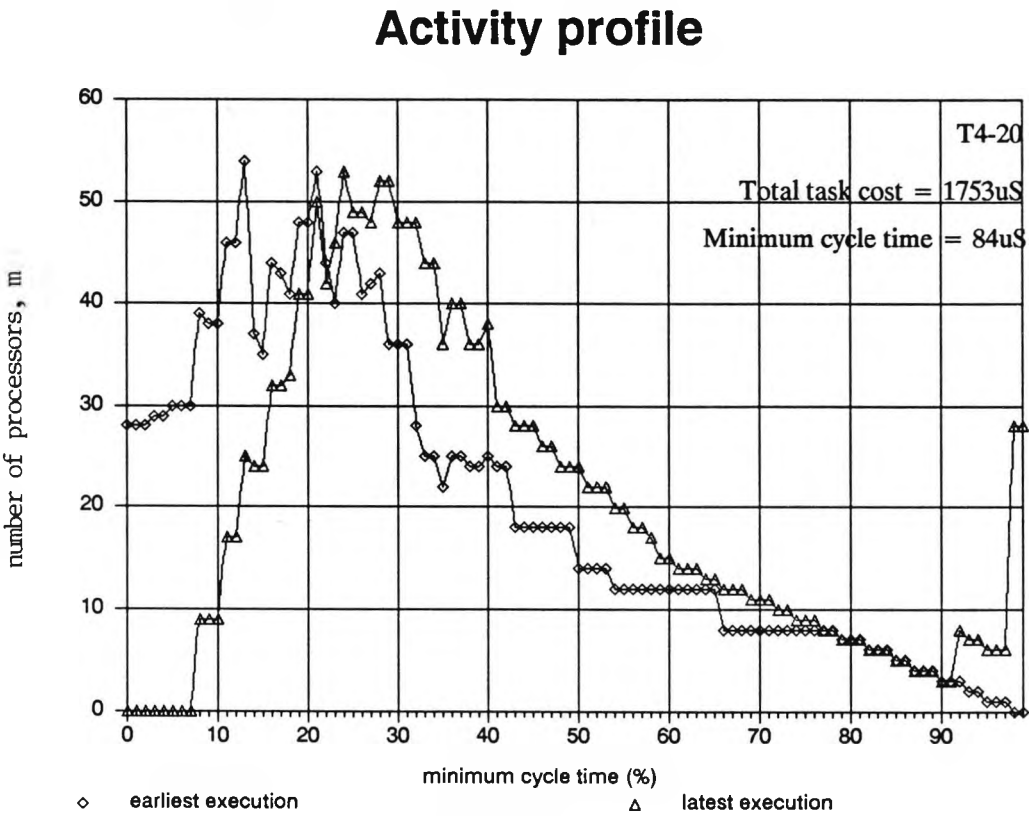


Figure 8.27

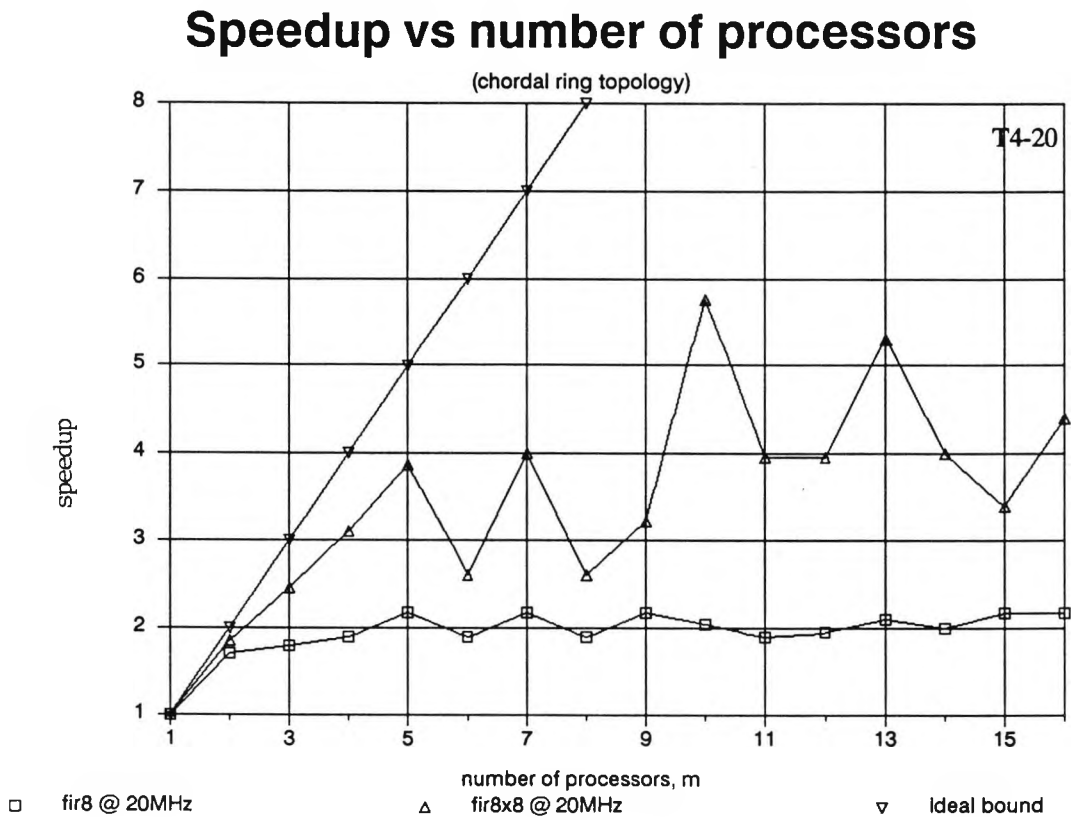


Figure 8.28

```
PROC fir8x8(CHAN OF INT32 in, out)
  ...  CHANNELS
  {{{  PROCEDURES
  {{{  fir8x801
  PROC fir801(CHAN OF ANY ch.1,
              CHAN OF INT32 ch.2,
              CHAN OF ANY ch.3,
              CHAN OF REAL32 ch.4)
    ...  VAL
    ...  VAR
    SEQ
      ...  INIT
      WHILE TRUE
        SEQ
          ...  Page - 1
          ...  Page - 2
          ...  Page - 3
          ...  Page - 4
          ...  Page - 5
          ...  Page - 6
          {{{  Page - 7
          aAe.real32 := aAf.real32 + aAe.real32
          ch.4 ! DELAY.8
          aAa.int32 := INT32 ROUND aAe.real32
          ch.1 ! aAa.int32
          aAe.real32 := coeff.7 * DELAY.12
          ch.1 ! DELAY.1
          value.5.4 := aAe.real32
          aAe.real32 := value.4.4 + value.5.4
          PAR
            ch.3 ! aAe.real32
            ch.1 ? aAf.real32
          ch.3 ! DELAY.8
          aAe.real32 := coeff.6 * DELAY.8
          value.7.5 := aAe.real32
          aAe.real32 := coeff.7 * DELAY.1
          ch.1 ! value.6.6
          value.6.5 := aAe.real32
          aAe.real32 := value.6.5 + value.7.5
          ch.1 ! aAe.real32
          DELAY.3 := aAf.real32
          }}}
          ...  Page - 8
  :
  }}}
  ...  fir8x802
  ...  fir8x803
  ...  fir8x804
  ...  fir8x805
  }}}
  PAR
    fir8x801(ch.any.01.0, in, ch.any.01.2, ch.real32.01.3)
    fir8x802(ch.real32.01.3, ch.real32.02.1, ch.any.02.2, ch.any.02.3)
    fir8x803(ch.any.02.3, ch.any.01.2, ch.real32.03.2, ch.any.03.3)
    fir8x804(ch.any.03.3, ch.any.02.2, out, ch.any.04.3)
    fir8x805(ch.any.04.3, ch.real32.03.2, ch.real32.02.1, ch.any.01.0)
:
```

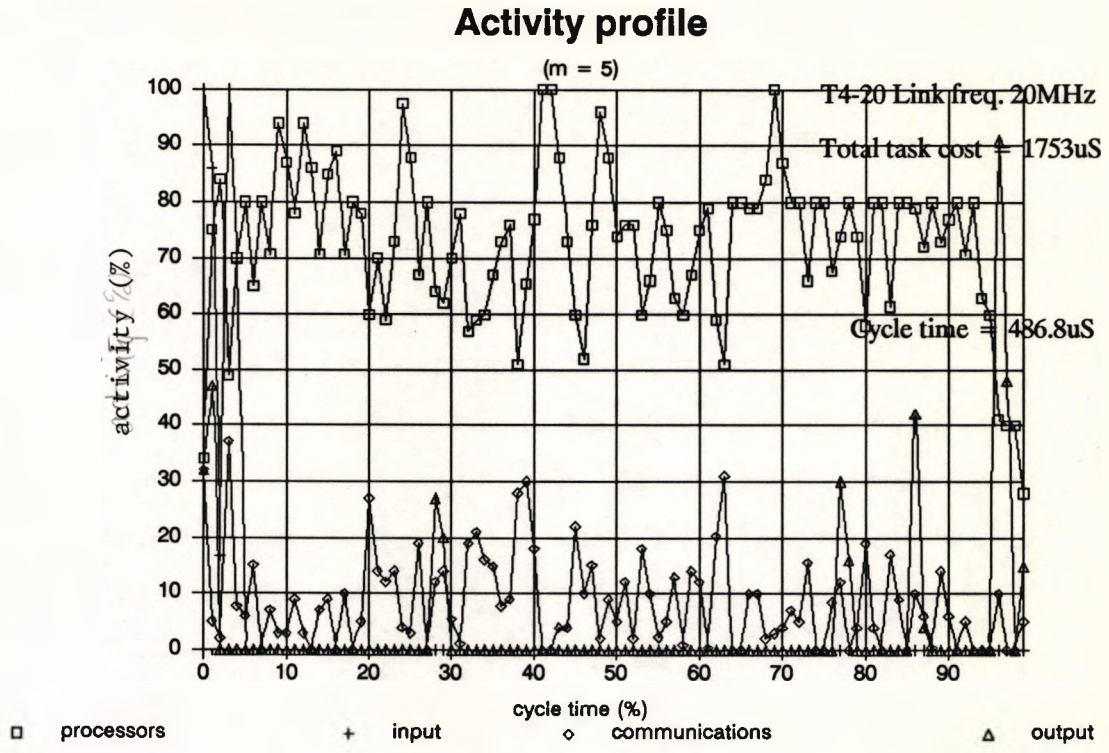


Figure 8.29

Figure 8.27 shows the impact on the algorithm's activity profile when 8 cycles are combined together. For example, the peak number of processors has risen from 15 processors (Figure 8.15) to 54 processors. This, along with the low increase in minimum cycle time, shows a significant increase in parallelism over the original algorithm.

The earliest and latest profiles (Figure 8.27) now tend to look similar. This indicates that there are many paths running through the algorithm that are critical, or lie close to the critical path, and suggests that the minimum cycle time will be difficult to attain.

Figure 8.28 compares the speedup for the single cycle algorithm (fir8) to the speedup for the multiple cycle algorithm (fir8x8). The multiple algorithm shows an increase in speedup from 2.2 to 5.8. Even when m is equal to 5, the speedup is 3.8 for the multiple case as opposed to 2.2 for the single case. The results from Figure 8.29 confirm the improvements in efficiency, by revealing an increased degree of processor utility.

One problem that may arise from using multiple cycle algorithms is that input and output events are not synchronised to regular time intervals, but operate on the principle of maintaining a sequential order. This may cause some difficulties and give rise to timing interference between I/O and scheduled algorithm. A solution to this problem would be to assign relative timings to all elements of input streams, which would specify the earliest and latest time that data could be inputted. Output timings would be a function of the input timings and the scheduled algorithm.

Chapter 9. Conclusions

Parallel processing, unlike sequential processing, introduces many viable, alternative routes to implementation. In the introduction, three distinct operations along the route to implementation were identified, these being parallelism definition, program partitioning and program-resource scheduling. These three operations are ordered and take place during distinct intervals; design-time, compile-time or run-time. The specific interval when an operation is performed characterises an implementation and so determines the implementation strategy.

An exclusively "compile-time" implementation strategy has been adopted and described by this thesis. Amongst the reasons outlined in the introduction, the justification for adopting this strategy was to circumvent the inefficiencies caused by poor process-processor allocation (design-time implementations) and scheduling overheads (run-time implementations).

This chapter summarises the implementation approach and concludes on its effectiveness. The final section suggests possible directions for future work.

9.1 Implementation summary

A brief summary of each step in the implementation strategy is now given. Most of the information contained in this section has been extracted from previous chapters.

9.1.1 Algorithm characteristics

The characteristics of discrete (cyclic) algorithms were reviewed in chapter 2, this illustrated the discreteness, complexity, memory, structure and composition of a discrete algorithm. The rules have been established for maintaining synchronisation between algorithm and input device, and algorithm and output device in a real-time system, which has shown that an algorithm must be deterministic if it is intended for a real-time system.

The algorithm has been expressed as comprising a set of disjoint tasks, whose complexity influences the granularity of the set. The cost (or execution time) of different task types was chosen to be small (i.e. low complexity tasks), hence an algorithm can be viewed as being composed of many medium-fine grain tasks. The alternative to this would have been a coarse grain structure, made up from a few complex tasks. A medium-fine grain structure was shown to have an advantage over coarse grain structures, in that a potentially high degree of parallelism can be represented. This approach does not preclude the advantage of low communication overhead offered by coarse grain structures from being utilised, since tasks can join together at some later time where it is advantageous to do so.

9.1.2 Algorithm representation

It has been shown that a discrete algorithm can be represented by a graph, **G** = **(T, C, B, E, A)**, which is transitive, irreflexive and asymmetric. These characteristics ensure the graph is both directed and acyclic (i.e. a directed acyclic graph, or DAG). A DAG describes a single cycle of a discrete, deterministic algorithm and is completely equivalent to the algorithm both in terms of function and structure, hence any parallelism is preserved.

A DAG has been shown to comprise nodes and arcs, these represent tasks and execution precedence respectively. Once costs have been assigned to those nodes representing tasks, the critical path method (CPM) can be applied to the graph **G**. CPM produces the earliest and latest start times for all tasks in **T** and gives the earliest overall cost of completion for an algorithm free from resource constraints. Results from the CPM are to be used for analysing the DAG, with a view to scheduling the tasks onto processors.

9.1.3 Language description

In order to facilitate algorithm description, in a form that is compatible to a DAG, a new programming language was developed. The design of the language, known as Digital Filter Description Language (DFDL) is based on the single-assignment rule, which conforms to the deterministic nature of a DAG. Single-assignment has been shown to have many effects, one of which is to preserve algorithmic structure and hence parallelism. Other effects have shown an influence on programming constructs, such as conditionals and repetitive constructs.

Program flow has been modelled on the repetitive input-process-output (or generate-output) cycle of a discrete algorithm. This has made DFDL suitable for describing deterministic, discrete processes (i.e. sampled systems) that have zero or more inputs and one or more outputs. More complex programs, which have different sample periods, can be described as separate parallel programs that are joined via their external input and output. The DFDL model of computation does not support nested programs.

External input and output have explicit data types (BYTE, INT16, INT32, REAL32.) and supports streams with up to two dimensions. A stream defines the number of inputs/outputs made over each program cycle. Internal variables (or more correctly, objects) are called nodes. Nodes have a fixed REAL32 data type. Type conversion is necessary wherever an input/output data type is non-REAL32, this is carried out implicitly. Constants are also type REAL32, and are declared as scalars, tables or expressions. DFDL's internal floating point data type facilitates the passing of run-time error messages, which provide a useful method of error tracing.

DFDL does not support procedures, but does include user-defined functions which provide a degree of abstraction. Within a function, all operands are passed as formal parameters or are declared within the function. Operands and objects declared within a function are only in scope within that function. Functions are used as operands and their result is a single value. A function can only be instanced after it has been declared, this prevents recursion, a feature which is not supported by DFDL.

A comprehensive range of arithmetic operators and standard functions are included in DFDL's syntax. Several special functions facilitate the addition, multiplication, mean, median, maximum and minimum of two or more operands. These functions are translated into maximally parallel structures.

DFDL restricts repetition to two cases. The first is a single program loop which constitutes the repetitive nature of the discrete algorithm, while the second facilitates operation upon multiple streams of data. The second form of repetition adheres to the single-assignment rule and is an efficient way to describe multiple operation on arrays.

The single-assignment rule requires that all declared objects and outputs are assigned once only. The conditional construct adheres to this rule by having a single object that is assigned from one of a number of expressions. Another result of single-assignment is the difficulty of assignment using operands from previous cycles. This difficulty was overcome by incorporating a Z operator into the language, this is appended to an operand to signify that data is from past program iterations. The number of past iterations is defined by indexing Z. A summary of DFDL's syntax is given in Appendix B.

9.1.4 Task graph

Chapter 5 described the data structures of those nodes comprising **G**. The low in/out-degree of these nodes (except B and E) is shown to produce a sparsely connected DAG, which can be realised using a doubly linked list whose length is proportional to $|\mathbf{T}| + |\mathbf{C}|$.

The different types of nodes (task primitives) have been described in terms of their in/out-degree, their worst case execution cost and their function (i.e. task primitive name). These nodes are used to create a connected graph structure (the DAG), as directed by a DFDL program.

Transformation from program to graph falls into three different categories of graph structure. The first includes external input/output, node and constant graph structures. The nodes which comprise these structures have been classified as named nodes, because they correspond to named elements in the program and form the skeleton of the DAG. The second category includes all graph structures that are primitive (simple nodes), which reflect simple or part transformations. Finally, more complex graph structures have been described, each using several primitive nodes.

9.1.5 Processor graph

A parallel processing model representing a loosely coupled Transputer based architecture has been described in chapter 6. The two resources important to scheduling were identified, namely processing and communications. The model has been shown to consist of two parts: (i) a processor graph $\mathbf{V} = (\mathbf{P}, \mathbf{I}, \mathbf{O}, \mathbf{L})$ (based mainly on the two resource types) and (ii) $|\mathbf{L}| + |\mathbf{P}|$ activity schedules, $\mathbf{S} = \{\mathbf{S}_1, ..., \mathbf{S}_q\}$. The processor graph gives a spatial representation of the machine (i.e. topology), while the activity schedules give a temporal representation (i.e. activity). The data structures for both parts of the model have been presented and illustrated using examples.

The latter part of chapter 6 has concentrated on the processor graph and the rules governing nodes and arcs that comprise the graph. Input and output have been included and the minimum system realisation has been defined which complies with that of DFDL. Finally, the problems accorded to building the graph have been discussed, this has been shown to create isolated nets unless precautions are taken. A solution to net isolation has been presented in the form of a three stage algorithm. The algorithm examines the graph whenever an arc is proposed, producing a boolean result which indicates whether or not net isolation would occur if the arc were established.

9.1.6 Compile-time scheduling

The merits and limitations of deterministic, compile-time scheduling have been discussed in chapter 7. Three different classes of scheduling have been illustrated; list scheduling, non pre-emptive scheduling and pre-emptive scheduling, and it was found that the non pre-emptive class is best suited to deterministic, compile-time scheduling.

Two major performance measures, schedule length and lateness, have been introduced and the relationship between these and the relationship between throughput and latency was discussed. The former measure is used in the definition of the scheduling problem, which is essentially a problem of schedule length minimisation.

The complexity of the scheduling problem has been investigated for different constraints on **T** and **P**, and the problem was found to have a number of solutions that is exponential of order n. To show the effect this has on inefficient algorithms that attempt to solve the problem, an exhaustive enumeration approach was illustrated. The problem was shown to be computationally intractable when using this type of algorithm. This result is reinforced by previous results which classify the problem as NP-complete.

Three basic heuristic search strategies, backtracking (BT), hill-climbing (HC) and best-first (BF) were introduced. The characteristics of each search strategy were discussed and it was considered necessary to introduce irrevocable scheduling decisions to ensure that time complexity became polynomial in n (the number of tasks). However, this step was shown to relinquish the guarantee of an optimal solution being found.

The final part of chapter 7 described the design of a hybrid HC BT-BF scheduling algorithm. At the heart of the algorithm is a heuristic, which guides the search strategy and so determines the scheduling decisions. Two heuristics have been compared, a minimum length heuristic and a minimum lateness heuristic. Each heuristic originates from its respective performance measure. The minimum lateness heuristic was shown to produce superior results. The reason for this being that the heuristic uses information from the CPM algorithm, which imparts "knowledge" of what lies ahead and so allows the algorithm to prioritise tasks accordingly. Chapter 7 concluded with a derivation for the scheduling algorithm's time complexity, which confirms it as polynomial in n, of order 2. This theoretical result has been confirmed by practical measurements made during the experimental results.

9.2 Concluding remarks

We now comment on both the expected performance and actual performance, and conclude on the effectiveness of the implementation strategy and the language DFDL.

9.2.1 Performance bounds

The CPM algorithm generates a value for the critical path cost. This cost defines the minimum time in which one cycle of a program can be executed. It also defines the maximum throughput (1 / critical path cost) and maximum possible speedup (total task cost / critical path cost) using parallel processing. Since this cost is treated as our goal, it is important to examine its origins.

Critical path cost is defined as the longest path through the graph G, from node B to node E. The duration of the critical path cost is dependent on how tasks are arranged (algorithm structure) and the cost of the tasks. Clearly, the algorithm's structure is dependent on the how the programmer describes the algorithm and the nature of the algorithm itself. Program language plays a large part in facilitating description, hence it is important to present a programmer with a language medium that allows him/her to concentrate on the application and not the implementation. Hopefully this has been achieved by DFDL.

Task complexity is determined by the particular partitioning strategy adopted. Chapter 2 described the partitioning strategy applied to DFDL programs when creating a task graph, G. Partitioning was shown to produce tasks with medium-fine grain complexities, which allows parallelism to be expressed between low cost tasks. To complement this, complex functions (SUM, PROD, etc.) are partitioned into maximally parallel structures.

Overall, the partitioning strategy aims to create a task graph that is maximally parallel and has minimal a cost critical path. The result of this approach is that potential speedup tends to be maximal. This would not be the case if, for example, a coarse grain partitioning strategy were to have been employed. It should be noted that when the value for maximum possible speedup is greater than the number of processors, m, for a particular scheduling problem, then the upper bound for speedup is m.

9.2.2 Factors affecting performance

Experimental results show that performance is often less than that defined by the performance bounds. Failure to attain the maximum potential speedup can be attributed to one, or more, of six effects, most of which have been described in chapter 8. The degree of speedup degradation due to some of these effects has been shown to be predictable.

9.2.2.1 Irrevocable scheduling

The first of these effects was discussed in chapter 7, and is due to the non-optimal nature of the scheduling algorithm. Non-optimality is caused by the need to take irrevocable scheduling decisions.

The noticeable effect of irrevocable scheduling is that the immediate predecessors to output tasks are often poorly scheduled. This only occurs when communication cost is non-zero. The reason for this poor scheduling is that an output task is pre-allocated a processor prior to scheduling and the scheduler places preceding tasks without consideration for this pre- allocation. The degree of degradation due to this effect is difficult to quantify, however, judging from the results it is probably small when compared to other effects. Suggestions for eliminating this source of inefficiency are given in section 9.3.3.

9.2.2.2 Integer effect

Integer effect occurs whenever n / m produces a remainder. The severity of this effect on speedup reduces as n becomes large in comparison to m. Integer effect has been shown to produce a maximum degradation in speedup of 50% when the value m approaches n, however, degradation will be small provided $n >> m$. When a schedule is severely affected by integer effect, one solution is to re-write the algorithm so it describes several cycles. This increases the value of n, while the number of processors can remain the same. An example of this technique was given in section 8.3.5, and was shown to improve speedup considerably.

9.2.2.3 Synchronisation and latent scope effects

Synchronisation and latent scope effects are products of the Occam language, which are produced by the scheduler in order that schedules can be translated into Occam. The first of these effects is caused by Occam's unbuffered, synchronised communication, which requires that both sender and receiver synchronise prior to passing data. This effect often introduces delay in communication.

Latent scope effect occurs when communication channels are held in scope while they are inactive. This is caused by Occam's inability to express non-nested, overlapping parallelism. This effect tends to reduce the available communication bandwidth.

Both these effects are proportional to the number of communications made and have been shown (section 8.2) to degrade maximum potential speedup by between 5% and 25%. Removal of these effects would only be possible if the Occam/Transputer model of computation were replaced by a true data-driven computational model.

9.2.2.4 Communication cost and topology effects

Communication cost effect can only occur when communication cost is non-zero and there exists an actual topology (i.e., $m > 1$). The degree of communication cost effect depends on the processor-communications cost ratio. An increase in the cost of communication relative to the cost of processing tends to impede the spread of data from one processor to another. This leads to a reluctance by the scheduler to exploit parallelism and so produces task coagulation. The effect is similar to using coarser grain tasks. Experimental results (section 8.2.4) have shown that there is a low speedup degradation (i.e., less than 5%) for ratios above 10:1, while degradation becomes severe (i.e., greater than 15%) when the ratio falls below 5:1.

The topology of the communications network has a definite effect on performance, this has been shown in section 8.2.5. The degree of topological effect has been shown to rely on m, mean degree and mean diameter of the topology. For example, a device such as the Transputer T4-20/20, which has a maximum mean degree of four, produces a topological degradation of 50% when the product of m and mean diameter is 88 (taken from Figure 8.14). Since mean diameter tends to increase with m, then m is probably limited to about 25 in this example.

9.2.3 Conclusions on performance

On average, the examples given in chapter 8, yield speedups of between 50% and 80% of their maximum bound. About half this loss is attributable to non-zero communication cost and topological effects. Variation in speedup from one value of m to the next can on occasions be considerable. Such large swings are thought to be caused by the more non-linear effects, namely integer, synchronisation and latent scope effect.

The results highlight the importance of communication bandwidth, which if inadequate will restrict the number of processors that can be usefully employed and hence, limit the degree of parallelism that can be exploited. This point is particularly relevant when using devices whose processing power vastly outweighs their ability to communicate (e.g., T8 Transputer). Perhaps, one can conclude that the current floating point Transputer is more suited to coarse grain parallelism, indeed this is the manufacturers intention (INMOS, 1986). For the future, it is hoped that manufacturers will realise the potential of fine grain parallelism and concentrate their efforts on increasing communication bandwidth. Suggestions include a move from serial to parallel communication and an increase in the number of links per device.

The Occam/Transputer model of computation has been shown to incur zero-cost communication penalties. To the best of the author's knowledge, neither synchronisation nor latent scope effects have been previously reported. The elimination of these effects may necessitate a move away from the parallel von Neumann machine to a static dataflow machine. However, an alternative solution lies in a re-design of the Transputer's architecture, which is typically von Neumann, and as such communicates across a single instruction/data bus.

9.2.4 Conclusions on DFDL

The use of DFDL has been illustrated by example, and descriptions have been given in chapters 3 and 4. Generally, a language's suitability to describe an algorithm depends on the language's facilities, its syntax, its computational model and the way a programmer interacts with the language. The deterministic nature of DFDL does deny a programmer much of the freedom he/she is used to, however, the author sees this as another step in the write direction, rather like the introduction of structured languages and the abolition of the GOTO statement.

The fact that DFDL is deterministic means that it is only possible to describe realisable systems. This feature has distinct advantages. For example, DFDL could be used as a high level interface to a VLSI automated process for the design of integrated circuits. Similarly, the deterministic nature also permits a simple interface to alternative mediums, such as graphics. Conversely, deterministic graphical descriptions may be described in a textural form by DFDL.

9.3 Future work

In common with most research work, there are outstanding problems, unanswered questions and unfinished work. This section discusses some of these items.

9.3.1 Reducing scheduling time

The current complexity of the scheduler is $\mathbf{O}\ (n^2)$. Reduction below this complexity is unlikely, unless the quality of scheduling is to be forfeit. However, when large task graphs are scheduled the scheduling time may be considered prohibitive. In such cases, provision should be made to divide the graph into several sub-graphs. The division between one sub-graph and another could be made through arcs that form simple cuts.

Another option, which is particularly relevant, is to employ parallel processing on the computationally intensive parts of the scheduling algorithm. This could be achieved reasonably easily, since the scheduler is a highly parallel algorithm.

9.3.2 Reducing scheduling memory

Currently, the scheduler loads all the data structures (e.g., task graph, processor graph and schedules) into RAM memory. This causes memory overflow when problems are large. An alternative approach would be to off-load areas of the task graph and schedules on to disk memory.

Other approaches could also be considered. One of these would be to interlace task graph construction, analysis, scheduling and translation. However, this would be difficult, since analysis (i.e., CPM) requires that the entire graph is available.

9.3.3 Improving the scheduler

Essentially, the scheduler works well and gives good results, however, there are areas that could be improved. One of these has been described in section 9.2.2., and concerns fixed location output tasks. One solution would be to allow output tasks to be scheduled on any processor. This solution is discounted, since an output relates to a physical link. An alternative solution would be to weight the scheduling heuristic so it accounted for the fixed output. Weighting would be confined to the portion of the task graph preceding the output task (i.e., lying close to and on a backwards arborescence from the output). The weighting applied to each task would decrease in proportion to its distance from the output task, and increase in proportion to the length of the shortest path between the task's processor and the output task's processor.

9.3.4 DFDL program linker

Individual DFDL programs can be connected together via their external input/output, provided data types are compatible. When an input and output from different programs are connected, then the number of cycles one program performs relative to the other is determined by the relative number of inputs/outputs made per cycle. Where more than one channel exists between programs, directly or indirectly, then the cyclic ratio must be consistent otherwise programs will deadlock. All these checks could be performed by a linker.

Normally, the linker would assume that programs are to run on separate devices. Where this is not the case, then programs would have to be linked prior to scheduling.

Finally, the task graph analysis assumes that I/O is free from external timing constraints. Clearly, this will not be the case when two or more separate DFDL programs are linked, since there will be timing interaction via their joint I/O. This would necessitate either some prefixed timing specification, composite scheduling or a progressive scheduling scheme that extracted timing information from adjoining scheduled programs. The consequence of neglecting this problem may cause a loss in throughput from one or more programs when joined together.

9.3.5 Inputs and outputs

The current implementation does not allow multiple input declarations, or multiple output declarations or mixed inputs and outputs to share the same physical link. This was done to simplify I/O. However, it is feasible that mixed I/O could be employed in practice.

9.3.6 Language model

The current implementation of DFDL does not include conditional statements, boolean operators, relational operators, user defined functions, the functions MED, MAX or MIN, user defined initialisation, boolean communication, or constant expressions. Although provision has been made for most of these items in the compiler, work is still required. Other items that could be included in the language are double length floating point arithmetic and possibly complex arithmetic. It is preferred to keep DFDL's internal data as some form of floating point, since this can convey run-time error information.

Other syntax additions may be needed for combining separate DFDL programs. Section 9.3.8 suggests some syntax additions to include a textural definition of the processor graph.

9.3.7 Deadlock avoidance

Deadlock avoidance has not been built into the current version of the scheduler. Consequently, it is possible that the Occam produced by the DFDL compiler will fail to run. On the occasions when this occurred, it was when there were multiple channels joining two processors, and communication had been scheduled in opposite directions at about the same time. Further investigation is needed into this problem, however, it is thought that the problem may be overcome if the scheduler first examines adjacent communication channels before establishing a communication.

The extent of deadlock may not be restricted to conflicts between two processors, but may involve any number of connected processors. Any comprehensive deadlock avoidance scheme would have to take this into account.

9.3.8 Processor graph definition

Currently, the processor graph is constructed from information entered from the keyboard by the user during compile-time. The user interface allows pre-selected topologies to be created easily, however, it may take several minutes to create a topology that is not included on the menu. It is suggested that an alternative source of information be provided. That is, parameters for the processor graph could be expressed as a program, for example:

```
{{{ file.name
... DFDL -- fold containing program
{{{ PROG
BEGIN
   PROC IS T4
   CLOCK FREQ IS 20MHz
   LINK FREQ IS 10MHz
   NUMBER IS 4

   COST OF TASKS IS STANDARD
   AUTOCONNECT IS OFF

   P1, L1 IS INPUT in
   P1, L3 IS P2, L1
   P1, L4 IS P4, L2
   P2, L4 IS P3, L2
   P2, L2 IS P4, L1
   P3, L3 IS OUT out
   P3, L1 IS P4, L3

END
}}}
}}}
```

Appendix A. Graph concepts and definitions

Some concepts and definitions of graphs are presented to avoid ambiguity over the use of these terms. The following references have been used to compile this appendix: (Minieka, 1978; Hetch, 1977; Glaser, Pyle and Illingworth, 1986).

A *graph* can be considered as consisting of points, or some other objects in a plane, connected together by a number of relationships. These relationships could be represented by lines, or arrows, connecting relevant points. Usually, points are called *nodes* (or *vertices*) and these are given labels, e.g., x_1, x_2, etc. The relationship that connects any two nodes characterises the graph and this relationship may be *directed* or *undirected*. It is usually convenient to represent directed relationships by a line with a single arrow head and undirected relationships by a line (or line with arrow head at either end).

Relationships between nodes are called *arcs* and are often written using the labels associated with the connecting nodes, e.g., (x_1, x_2). Where the relationship is directed, node x_1 is called the *tail* and x_2 the *head*, and the order in which they are written is important. If there is more than one arc connecting two nodes and the direction is the same then these may be distinguished by subscripting arcs, e.g., $(x_1, x_2)_1$, $(x_1, x_2)_2$.

When a graph represents a problem that does not need directed relationships between nodes then the graph is called an *undirected graph*. The undirected arcs that make up this type of graph are called *edges*. The two types of graph, directed and undirected, are distinguished from one another by their notation; **(X, E)** is used to denote an undirected graph with node set **X** and edge set **E**, and **(X, A)** denotes the *directed graph* with node set **X** and arc set **A**.

Often, the term *network* is used, here it is merely a graph with one or more numbers associated with each arc, or node. A network is not necessarily a directed graph, but does refer to graphs that are connected.

An arc that has both the same tail and head node is called a *loop*. A node and an arc are said to be *incident* to one another if the node makes up the tail, or head or the arc. Two arcs are said to be *incident* to one another if they are both incident to the same node. Two nodes are said to be *adjacent* to one another if there is an arc joining them.

Consider the sequence $x_1, x_2, x_3, ..., x_n, x_{n+1}$ of nodes. A *chain* is any sequence of arcs $a_1, a_2, ..., a_n$ such that the end points of a_i are x_i and x_{i+1} for $i = 1, 2, ..., n$. Thus, either $a_i = (x_i, x_{i+1})$ or $a_i = (x_{i+1}, x_i)$. Node x_1 is called the *initial node* of the chain and node x_{n+1} is called the *terminal node* of the chain. The chain is said to extend from the initial node to the terminal node. The length of the chain equals the number of arcs in the chain.

A *path* is a chain for which $a_i = (x_i, x_{i+1})$ for $i = 1, 2, 3, ..., n$. The length, initial node and terminal node can be defined similarly. A *cycle* is a chain whose initial and terminal node are the same. A *circuit* is a path whose initial and terminal nodes are the same. The length of a cycle or circuit is defined as the length of the corresponding chain. A chain, path, cycle or circuit is called *simple* if no node is incident to more than two of its arcs, i.e., if the chain, path, cycle or circuit properly contains no cycles.

A graph is called *connected* if there is a chain joining every pair of distinct nodes in the graph. A graph may be regarded as consisting of a set of connected graphs. Each of these connected graph is called a *component* or the original graph. A graph is called *strongly connected* if for any two nodes x and y in the graph, there is a path from x to y.

Let **X'** be any subset of **X**, the node set of graph $\mathbf{G} = (\mathbf{X}, \mathbf{A})$. The graph whose node set is **X'** and whose arc set consists entirely of arcs in **A** with both end points in **X'** is called the *subgraph generated by X'*.

Let **A'** be any subset of **A**, the arc set of graph $\mathbf{G} = (\mathbf{X}, \mathbf{A})$. The graph whose arc set is **A'** and whose node set consists entirely of nodes that are incident arcs in **A'** is called the *subgraph generated by A'*.

A set of arcs is called a *tree* if it satisfies two conditions:

(i) The arcs generate a connected subgraph

(ii) The arcs contain no cycles.

A *forest* is any set of arcs that contains no cycles. Thus, a forest contains one or more trees. A *spanning tree* of a graph is any tree formed from the arcs of the graph that includes every node in the graph. Clearly, no spanning tree can exist in a graph that contains more than one component, and every connected graph possesses a spanning tree. A tree with one arc contains two nodes, a tree with two arcs contains three nodes, etc. In general, a tree with n-1 arcs must contain n nodes. Hence, each spanning tree of a connected graph with n nodes consists of n-1 arcs.

A set of arcs whose removal from the graph increases the number of components in the graph is called a *cut*. A cut that contains no other cuts as a subset is called a *simple cut*.

An *arborescence* is defined as a tree in which no two arcs are directed into the same node. Note, several arcs in an arborescence can share a common tail node. An arborescence can be thought of as a directed tree that can be used as a grapevine. The *root* of an arborescence is the unique node included in the arborescence that has no arcs directed into it.

Appendix B. EBNF description of DFDL

There are two extended Backus-Naur form descriptions in this appendix. The first describes the lexical part of DFDL, while the second part describes the syntax of DFDL. All lines preceded by an asterisk are not implemented in this version of DFDL; DFDL(mod.state 1/90).

B.1 EBNF description of DFDL lexical analyser

non.alphanumeric.char :: = | ! | | | " | £ | $ | % | ^ | & | * | (|) | _ | +
| = | { | [| } |] | : | ; | @ | ' | ~ | # | , | . | > | < | ? | / | \ | | -
digit :: = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
lc.letter :: = a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | z
uc.letter :: = A | B | C | D | E | F | G | H | I | J | K | L | M
| N | O | P | Q | R | S | T | U | V | W | X | W | Z
letter :: = *uc.letter* | *lc.letter*
return :: = *c
comment :: = % { 0 *(digit* | *letter* | *non.alphanumeric.char)*} *(return* | % *)*
exponent :: = E *monadic.op* { 1 *digit*}
real :: = [monadic.op] { 1 *digit*} . { 1 *digit*} *[exponent]*
integer :: = *[monadic.op]* { 1 *digit*}
monadic.op :: = + | -
ident :: = *lc.letter* { 0 *lc.letter* | *digit* | . }
reserved.word :: = { 1 *uc.letter*}
delimiters :: = *delim.assign* | *delim.plus* | *delim.minus* | *delim.mult*
| *delim.divide* | *delim.power* | *delim.rem* | *delim.lt* | *delim.not.equal*
| *delim.gt* | *delim.lt.eq* | *delim.gt.eq* | *delim.equal* | *delim.continued*
| *delim.lh.bracket* | *delim.rh.bracket* | *delim.lh.square* | *delim.rh.square*
| *delim.comma* | *delim.semi.colon*

delim.assign ::= :=
delim.plus ::= +
delim.minus ::= -
delim.mult ::= *
delim.divide ::= /
delim.power ::= **
delim.rem ::= \
delim.lt ::= <
delim.not.equal ::= < >
delim.gt ::= >
delim.lt.eq ::= < =
delim.gt.eq ::= > =
delim.equal ::= =
delim.continued ::= { 2 . }
delim.lh.bracket ::= (
delim.rh.bracket ::=)
delim.lh.square ::= [
delim.rh.square ::=]
delim.comma ::= ,
delim.semi.colon ::= ;

reserved.word.types ::= *delim.abs* | *delim.acos* | *delim.alog* | *delim.and*
| *delim.asin* | *delim.atan* | *delim.begin* | *delim.byte* | *delim.cos*
| *delim.end* | *delim.else* | *delim.elseif* | *delim.every* | *delim.exp*
| *delim.expression* | *delim.for* | *delim.forever* | *delim.from*
| *delim.funct* | *delim.init* | *delim.int* | *delim.if* | *delim.is* | *delim.log*
| *delim.ln* | *delim.mod* | *delim.max* | *delim.mean* | *delim.med*
| *delim.min* | *delim.not* | *delim.or* | *delim.prod* | *delim.program*
| *delim.repeat* | *delim.result* | *delim.real* | *delim.sin* | *delim.sqrt*
| *delim.sgn* | *delim.sum* | *delim.tan* | *delim.table* | *delim.then*
| *delim.value* | *delim.z*

delim.abs ::= ABS
delim.acos ::= ACOS
delim.alog ::= ALOG
delim.and ::= AND
delim.asin ::= ASIN
delim.atan ::= ATAN
delim.begin ::= BEGIN
delim.byte ::= BYTE
delim.cos ::= COS
delim.end ::= END
delim.else ::= ELSE
delim.elseif ::= ELSEIF
delim.exp ::= EXP

delim.expression ::= EXPRESSION
delim.every ::= EVERY
delim.for ::= FOR
delim.forever ::= FOREVER
delim.from ::= FROM
delim.funct ::= FUNCTION
delim.init ::= INIT
delim.int ::= INT
delim.if ::= IF
delim.is ::= IS
delim.log ::= LOG
delim.ln ::= LN
delim.max ::= MAX
delim.mean ::= MEAN
delim.med ::= MED
delim.min ::= MIN
delim.mod ::= MOD
delim.not ::= NOT
delim.or ::= OR
delim.prod ::= PROD
delim.program ::= PROG
delim.result ::= RESULT
delim.repeat ::= REPEAT
delim.real ::= REAL
delim.sin ::= SIN
delim.sqrt ::= SQRT
delim.sgn ::= SGN
delim.sum ::= SUM
delim.tan ::= TAN
delim.table ::= TABLE
delim.then ::= THEN
delim.value ::= VALUE
delim.z ::= Z

B.2 EBNF description of DFDL syntax

B.2.1 Program

program :: = PROG *program.name*
(*ext.declaration*) *int.declaration*
BEGIN *assignment.section* END
program.name :: = ident

B.2.2 External and Internal declarations

ext.declaration :: = {*1* { *0 input.declaration*}
{ *1 output.declaration*} }
int.declaration :: = {*0* { *0 node.declaration*}
{ *0 constant.declaration*}
{ *0 function.declaration*} }

B.2.3 Input, output and node declaration

input.declaration :: = INPUT(*data.type*) *input [sub.size]* { *0, input [sub.size]*}
output.declaration :: = OUTPUT(*data.type*) *output [sub.size]* { *0, output [sub.size]*}
node.declaration :: = NODE *node [sub.size]* { *0, node [sub.size]*}
input :: = ident
node :: = ident
output :: = ident
data.type :: = BYTE | INT16 | INT32 | REAL32
sub.size :: = *column.size [row.size]*
column.size :: = [pos.integer]
row.size :: = [pos.integer]

B.2.4 Constant declaration

constant.declaration :: = VALUE *(constant [sub.size]* IS *constant.expression*
| *undefined.constant [sub.size]* IS EXPRESSION
| TABLE *constant [sub.size]* IS *table)*
constant :: = ident
undefined.constant :: = ident
sub.size :: = *column.size [row.size]*
column.size :: = [pos.integer]
row.size :: = [pos.integer]
table :: = *real.string* { *0* ; *real.string*}
real.string :: = [real { *0* , real}]

B.2.5 User defined function declaration

*function.declaration :: = FUNCTION function.name ([formal.parameters])
function.body
*function.name :: = ident
*formal.parameters :: = formal [sub.size] { 0 , formal [sub.size]}
*formal :: = ident
*function.body :: = {0 { 0 node.declaration}
*{ 0 constant.declaration}
*{ 0 function.declaration} }
*BEGIN { 0 assignment}
*RESULT : = (conditional.expression | expression)

B.2.6 Assignments

assignment.section :: = [INIT init.section]
REPEAT (FOREVER | FOR non.neg.integer) repeat.section

B.2.7 Initialise assignment

*initialise.section :: = { 1 value.constant | init.assignment}
*value.constant :: = undefined.constant IS constant.expression
*init.assignment :: = init.object : = constant.expression
*init.object :: = input { 0 spatial.sub} Z temporal.sub
*| output { 0 spatial.sub} Z temporal.sub
*| node { 0 spatial.sub} Z temporal.sub

B.2.8 Constant expressions

*constant.expression :: = constant.operand
*| monadic.function constant.operand
*| multi.function constant.operand { 0 , constant.operand}
*| conv.function spatial.sub
*| monadic.op constant.operand
*| constant.operand dyadic.op constant.operand

B.2.9 Constant operands

*constant.operand :: = real
*| constant { 0 spatial.sub}
*| (constant.expression)
spatial.sub :: = fixed.sub | repetition.sub

B.2.10 Subscripts

spatial.sub :: = *fixed.sub* | *repetitive.sub*
temporal.sub :: = *fixed.sub* | *repetitive.sub*
fixed.sub :: = [*non.neg.integer.expression*]
repetition.sub :: = [*repetition.ident*
FROM *non.neg.integer.expression*
FOR *pos.integer.expression*
[EVERY *integer.expression*]]
repetition.ident :: = ident

where *non.neg.integer.expression* > = 0
and *pos.integer.expression* > 0

integer.expression :: = *integer.operand*
| *monadic.integer.op integer.operand*
| *integer.operand dyadic.integer.op integer.operand*
integer.operand :: = integer
*| (*integer.expression*)
monadic.integer.op :: = + | -
dyadic.integer.op :: = + | - | * | MOD

B.2.11 Real arithmetic operators

monadic.op :: = + | -
dyadic.op :: = + | - | * | / | \ | **

B.2.12 Functions

monadic.function :: = ABS | ACOS | ASIN
| ATAN | COS | EXP | LOG | LN | SGN
| SIN | SQRT | TAN | ALOG
multi.function :: = MEAN | SUM | PROD
*MAX | MIN | MED
conv.function :: = REAL

B.2.13 Repeat assignment

repeat.section :: = { *1 assignment*}
assignment :: = *object* := *(expression* | *conditional.expression)*

B.2.14 Objects

object :: = node { o spatial.sub} | output { o spatial.sub}

B.2.15 Expressions

expression :: = operand
| monadic.function operand
| multi.function operand { o , operand}
| conv.function spatial.sub
| monadic.op operand
| operand dyadic.op operand

B.2.16 Operands

operand :: = real
| input { o spatial.sub} [Z temporal.sub]
| output { o spatial.sub} [Z temporal.sub]
| node { o spatial.sub} [Z temporal.sub]
| constant { o spatial.sub}
**| function.instance*
| (expression)
**function.instance :: = function.name ([passed.parameters])*
**function.name :: =* ident
**passed.parameters :: = operand* { 0 , *operand*}

B.2.17 Conditionals

**conditional :: =* IF *boolean.expression*
*THEN *(expression | (conditional.expression))*
*{ o ELSEIF *boolean.expression* THEN *(expression | (conditional.expression))*}
*ELSE *(expression | (conditional.expression))*

B.2.18 Boolean expressions

**boolean.expression :: = relational.expression*
**| monadic.boolean.op relational.expression*
**| relational.expression dyadic.boolean.op relational.expression*

B.2.19 Relational expressions

*relational.expression ::= operand relational.op operand
*| (boolean.expression)

B.2.20 Relational operators

*relational.op ::= < | < = | = | > = | > | < >

B.2.21 Boolean operators

*monadic.boolean.op ::= NOT
*dyadic.boolean.op ::= AND | OR

Appendix C. User's guide

C.1 Installation

The DFDL compiler and examples are contained on four low-density disks. The compiler is used inconjunction with the D700 TDS, this must be installed prior to running DFDL.

In order to install the DFDL compiler and examples you will need at least 2Mbytes of hard disk space. To install DFDL place disk #1 in drive A: and type

```
A:INSTALL
```

The batch file INSTALL.BAT creates the directory C:\DFDL and transfers the contents of disk #1 into that directory. A prompt will request the remaining disks #2, #3 etc. The compiled code is executable on the T4 Transputer. If DFDL is to be run on other types of Transputer then the code will have to be re-compiled. Consult the DFDL information fold for any pre-compilation settings.

C.2 Getting Started

The user should be familiar with TDS and its folding editor before running DFDL. Assuming this is so, DFDL is started by running TDS2.BAT in C:\DFDL. This will result in the four top folds:

```
...F UTILITY.TOP
...F DFDL.TOP
...F EXAMPLE.TOP
...F USER.TOP
```

appearing on the screen. All folded structures are denoted by their leading dot dot dot notation. A description of the TDS fold structure is given in the TDS literature and a tutorial is provided for those users unfamiliar with the TDS folding editor.

The contents of the four top folds are:

- UTILITY.TOP TDS/OCCAM utilities for compiling, configuring, file utilities, etc.
- DFDL.TOP DFDL compiler, DFDL library and DFDL information
- EXAMPLE.TOP examples of DFDL source programs
- USER.TOP user program area

C.3 Running the DFDL compiler

The DFDL compiler is used by getting the compiler's executable code, this is covered in steps 1 to 3:

1 Enter the top fold ... DFDL.TOP
2 Place cursor on ... DFDL Compiler
3 Press [get code] (key [F5])
4 Move to source file and place cursor on fold line
5 Press [run code] (key [F6])

Step 5 executes the DFDL compiler, which takes as its input the source file selected in step 4.

C.4 Exiting

At almost every point in the compilation, the user can abandon the process by pressing [Alt][F2]. If necessary, TDS can be reset by pressing [Ctrl][Break] or [Ctrl][Y], then [Space]. To exit TDS normally, move to the top fold structure using [PgUp], then press [Alt][F2].

C.5 Making a DFDL source file

DFDL source files are made using the TDS folding editor. Several examples are given in the top fold EXAMPLE.TOP. At the outer level a source program looks like:

...F *filename*

The *filename* can comprise letters and numbers, but must be free from full stops and non-alphanumeric characters. All source files are automatically given the extension .tsr by the TDS folding editor when first made. TDS (D700 implementation) only accepts the first six characters of a *filename* . Duplicate filenames are catered for by the TDS editor.

Inside the outer fold of a DFDL source file (fold 0) is the source fold (fold 1):

```
{{{F filename
... DFDL
}}}
```

This fold is always identified by the name DFDL and is not filed. Within this fold resides the source program written in DFDL, this fold (fold 1) can contain other folds as stipulated by the TDS editor. Folds within fold 1 are ignored by the DFDL compiler, and hence, not displayed in DFDL error reports.

```
{{{ DFDL
... fold
... fold
... fold

... fold
}}}
```

Additional folds may be created by the DFDL compiler, these are located within the outer fold (fold 0), immediately after the DFDL fold:

```
{{{F filename
... DFDL
... fold
... fold

... fold
}}}
```

These folds contain syntax error reports and object code; occam source folds. The contents of all folds located after fold 1 (denoted DFDL) are ignored by the DFDL compiler. Error folds and object code folds should be removed by the user when not needed.

C.6 Flow

A flow diagram of the compiler options is given by Figure C.1, this is complemented by descriptions of each section, these are given below:

C.6.1 Processor type

The user chooses the type of processor the object code is required to run on, this may be different from the compiling processor. Options are T4 (T414 transputer) or T8 (T800 transputer).

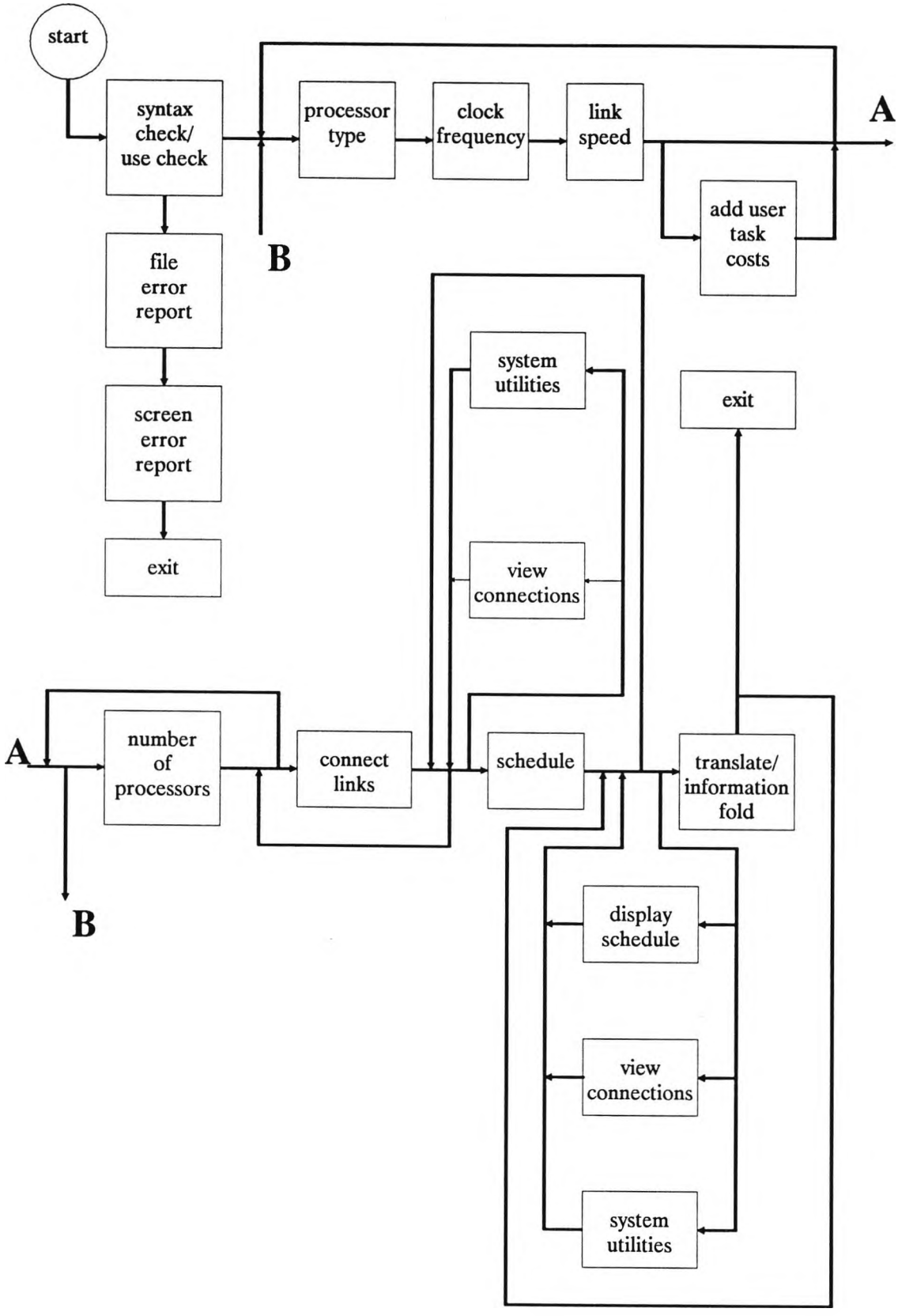


Figure C.1 Compiler flow diagram

C.6.2 Clock frequency

The user chooses the clock frequency of the target processor, this affects the time (or cost) operations take to run on the object processor, which in turn, affects the analysis performed by the compiler. Options range from 1MHz to 99MHz in 1MHz steps.

C.6.3 Link speed

The user selects the link speed of the target processor, this affects the communication time between object processors, which in turn, affects the analysis performed by the compiler. Options range from 1Mbits/sec to 99Mbits/sec in 1Mbits/sec steps.

C.6.4 Number of processors

The number of object processors is chosen by the user, the number may be in the range 1 to 99 inclusive. A suggested number is provided by the compiler at this stage. This number is calculated as a by-product of critical path analysis and is the truncated integer result of:

$$(\text{total task cost} / \text{critical path cost}) + 0.5$$

This estimate neglects communication costs, which are only available once scheduling has been performed. This figure may deviate, somewhat, from the best number of processors for implementation and should, therefore, only be used as a guide. In addition to this estimate are two pre-schedule program profiles that can be displayed by pressing the [V] key. These display the results of the critical path algorithm as time vs. number of useable processors.

C.6.5 Connect links

The user has to inform the compiler of all the object processor links that are dedicated to inputting and outputting data. Defining inter-processor link connections are optional, because the compiler incorporates an auto-router. Messages will alert the user to the following:

- When connecting links:
  (i) Connection of a link isolates an unconnected processor(s) from future connection.
  (ii) A link is already connected.
  (iii) A link cannot be connected to itself.
  (iv) A link cannot be connected to the same processor as it is connected.

- When requesting an input connection:
  (i) The source program does not contain inputs.
  (ii) All inputs have been connected.

- When requesting an output connection:
  (i) All outputs have been connected.

- When exiting the connecting process:
  (i) An input or output is not been connected.
  (ii) Warning, there is more than one processor net.
  (iii) There are isolated processors.

A multiple net condition is only a warning, since auto-routing will connect any processors required during the scheduling process.

C.6.6 View Connections

While there is a valid processor graph (or net), the user can look at the interconnections between processors and input and output connections without re-entering the connection process.

C.6.7 Schedule

There are several options the user can take before the scheduling process begins:

- Auto-route toggles the auto-route option ON/OFF. Auto-route ON is mandatory if there are multiple processor nets. Conversely, auto-route OFF is mandatory when there are no valid unconnected links.

- Alarm toggles the alarm option ON/OFF. The alarm alerts the user that the scheduling process has completed. This may be advantageous when scheduling is expected to take a long time.

- Display toggles the schedule display ON/OFF. The schedule display shows the user
  (i) Number of tasks to be scheduled.
  (ii) Number of tasks available.
  (iii) Number of processor nets.
  (iv) Number of tasks scheduled.

- Root allows the user to select the processor from which the scheduling process begins. This may affect the outcome of the scheduling process, especially when the topology of the processor net is irregular.

- Accept begins the scheduling process.

C.6.8 Schedule display

The schedule display presents the user with view of the processor and link schedules. These are arranged across the screen in the form of a Gantt chart. The screen displays up to 3 sets of processor/link schedules at a time, each comprising a processor schedule and four link schedules.

The schedule display flow is shown by Figure C.2.

- [F1] Help menu.

- [F2] Timing display: see timing display flow.

- [F3] Change view: allows the user to select different processor schedules for display.

- [F4] Show the menu at the bottom of the screen.

- [F5] Decrease step size by one: the step is used by other functions.

- [F6] Increase step size by one: the step is used by other functions.

- [F7] Begin schedule: moves the screen to the beginning (time zero) of the schedules.

- [F8] End schedule: moves the screen to the end of the longest schedule.

- [F9] Down schedule: moves the screen towards the beginning of the schedule by the step size.

- [F10] Up schedule: moves the screen towards the end of the schedule by the step size.

- [+] Zoom in: decrease the scale of the schedule display by the step size.

- [-] Zoom out: increase the scale of the schedule display by the step size.

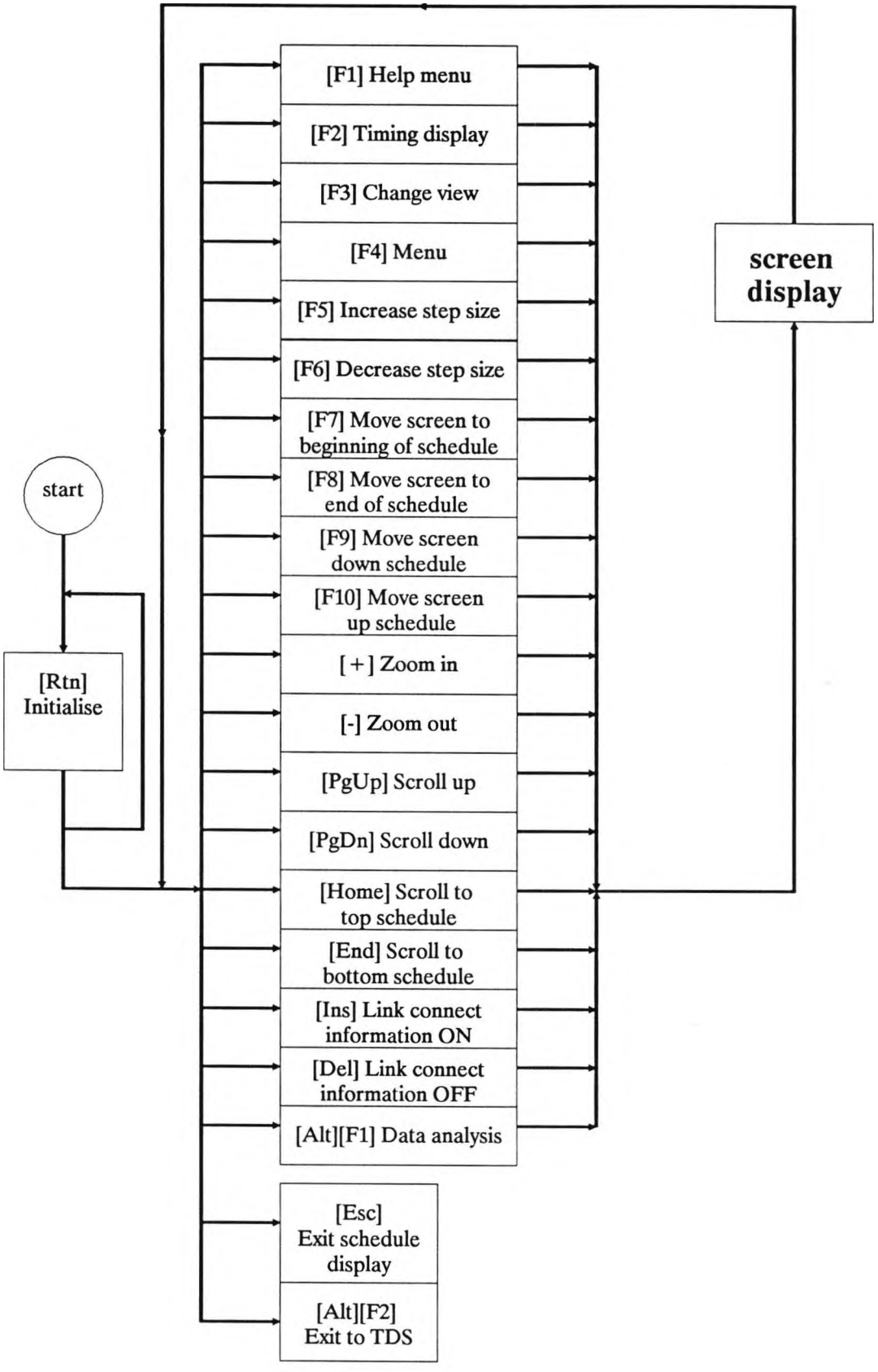


Figure C.2 Schedule display

- [PgUp] Scroll up: move the screen up, to display the processor schedule above that already at the top of the screen.

- [PgDn] Scroll down: move the screen down, to display the processor schedule below that already at the bottom of the screen.

- [Home] Top: move the screen to display the first processor schedule at the top of the screen.

- [End] Bottom: move screen to display the last processor schedule at the bottom of the screen.

- [Ins] Connect information ON: display link connection information. This is displayed on the right hand side of the screen and obscures part of the schedules.

- [Del] Connect information OFF: replace link connection information by link schedules.

- [Rtn] Initialise display

- [Alt][F1] Data analysis: see data analysis.

- [Esc] Exit display.

- [Alt][F2] Exit to TDS.

C.6.9 Timing display

The timing display can only be used when the displayed scale is 1:1. It allows the cursor to be moved around the screen so further information can be extracted from the display. When the cursor is placed on a schedule the task beneath the cursor is highlighted. The type of task and its starting and finishing times are displayed at the top of the screen. The timing display flow is shown by Figure C.3.

- [F1] Help menu.

- [F3] Change view: allows the user to select different processor schedules for display.

- [F4] Show the menu.

- [F5] Decrease step size by one: the step is used by other functions.

- [F6] Increase step size by one: the step is used by other functions.

- [F7] Begin schedule: moves the screen to the beginning (time zero) of the schedule beneath the cursor.

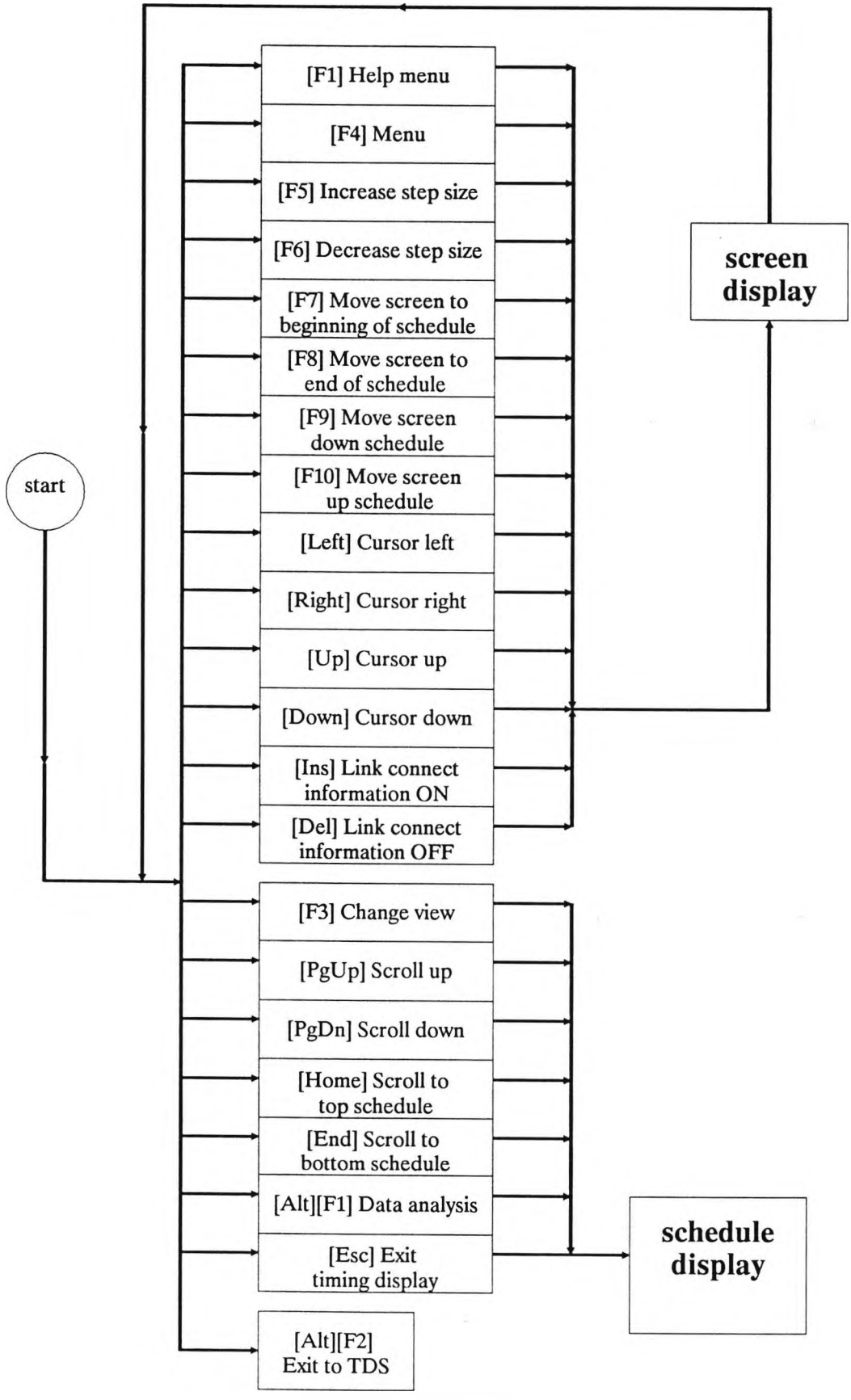


Figure C.3 Timing display

- [F8] End schedule: moves the screen to the end of the schedule beneath the cursor.
- [F9] Down schedule: move the screen towards the beginning of the schedule by the step size.
- [F10] Up schedule: moves the screen towards the end of the schedule by the step size.
- [PgUp] Scroll up: moves the screen up, to display the processor schedule above that already at the top of the screen.
- [Pgdn] Scroll down: moves the screen down, to display the processor schedule below that already at the bottom of the screen.
- [Home] Top: moves the screen to display the first processor schedule at the top of the screen.
- [End] Bottom: moves the screen to display the last processor schedule at the bottom of the screen.
- [Ins] Connect information ON: display link connection information. This is displayed on the right hand side of the screen and obscures part of the schedules.
- [Del] Connect information OFF: replace link connection information by link schedules.
- [Rtn] Initialise display
- [Alt][F1] Data analysis: see data analysis.
- [Esc] Exit timing display and return to the schedule display.
- [Alt][F2] Exit to TDS.
- [Tab left] Move cursor 10 places left
- [Tab right] Move cursor 10 places right.

C.6.10 Data analysis

The data analysis selection provides the user with data from the schedules. The format of the data is given below:

- Total task cost: this is the sum of all tasks in the task graph and is equivalent to the cost (or time) expended by a single processor when executing all the tasks. This figure excludes communication costs.
- Critical path cost: the cost of the longest path through the task graph. This excludes communication costs.
- (Total task cost / critical path cost): gives the maximum limit of available processor speed up, ignoring communication costs.
- Number of processors selected.
- Maximum processor cycle: the cost (or time) of the longest processor schedule.
- Maximum link cycle: the cost (or time) of the longest communications link schedule.
- (Total task cost / maximum cycle): gives speed up. The maximum cycle is the greatest of the two cycle costs; maximum processor cycle and maximum communications link cycle.
- Maximum latency: the greatest time from when a signal enters an input to when it exits at an output.
- Maximum overlap: the maximum possible overlap cost between successive schedules.

During analysis, the latency results for up to 400 paths can be stored, these can be displayed using the [Left] and [Right] cursor keys.

For each processor, the processor and link cycle cost, utility, percentage cycle cost and percentage utility are displayed towards the bottom of the screen, one processor at a time. Other processor's cycle cost, utility, etc. are available for display using the [Up] and [Dn] cursor keys.

C.6.11 Translate

The translate option converts the schedules into object code (Occam). There are two translate options:

- the first is labelled TDS, this creates object code for execution within the TDS environment.
- the second option, T4 or T8, generates object code suitable for execution on multiple processors.

The translate option is not completely implemented in this version of DFDL, DFDL(mod.state 1/90), consult the DFDL information fold for details.

C.6.12 System utilities

The [?] key calls the system utilities option, these include: schedule inspection, graph inspection, compiler array use, timer and symbol table inspection.

C.6.13 Add user task costs

The [!] key calls the option to adjust the cost assigned to tasks.

C.6.14 Information fold

A summary of the pre-shedule and post-schedule profiles, predicted and actual results are written into a single fold at the end of the fold bundle within the outer source fold.

Appendix D. Programmer's guide

D.1 Reading the source file

It is the purpose of the process &Read to open, read and close a DFDL source file. In doing so, &Read adds a communication protocol to the source file data which it has read, and sends it to &Lex in the form of an omni-directional data/control stream. Termination of an uncompleted data stream is not possible from outside &Read, because data/control only flows out of &Read. Termination occurs when the source file has been read completely, or there is an error associated with opening, reading or closing the source file.

&Read mainly consists of two TDS library routines. These are **read.fold.string**() and **keystream.from.file**(). Details of these two library routines can be found in the TDS Programming interface literature or under TDS in directory C:\TDSIOLIB.

D.1.1 File name

The library routine **read.fold.string**(), reads the character string associated with the first folded structure (fold 1) found at the top of the outermost fold (fold 0). Fold 0 is that fold which lies beneath the screen cursor when the routine is executed. The character string attributed to fold 1 is checked to be equal to "DFDL", this indicates that the fold contains a source program. An erroneous string or a file read error will cause &Read to send the message:

read.error / ft.terminated

and then terminate, whereas a successful read of the DFDL label causes &Read to read the contents of fold 1.

D.1.2 File contents

Provided the initial opening and reading of the file was successful, the transfer of its contents then commences. The library routine **keystream.from.file**() reads the contents of fold 1 and writes them to &Lex, a process running in parallel with &Read. In the course of reading and writing, **keystream.from.file**() removes all characters associated with the Occam fold structure, this allows the user to employ a folded source program structure without corrupting the program. At the end of each line, **keystream.from.file**() inserts a *return* character, and at the end of the file it inserts a termination tag, *ft.terminated*. For example a typical data/control stream would look like:

F / i / r / s / t / / l / i / n / e / *return*
S / e / c / o / n / d / / l / i / n / e / *return*
...
...
...
L / a / s / t / / l / i / n / e / *return*
ft.terminated / *errornum*

An error value is appended to the termination tag, to indicate the outcome of closing the source file. Successful closure is represented by the value *fi.ok*, otherwise a suitable error value is appended according to the error; see

D.2 Lexical analysis

Under non-error conditions, &Lex reads the data/control stream from &Read. First the file name, followed by program character streams divided into lines by the *return* character, *c. The final string &Lex reads is the terminate tag, *ft.terminated*, followed by a value indicating the state of file closure. Once the terminate string is read, &Lex terminates.

While &Lex reads the data/control streams from &Read, it conditions the input data and sends the results to &Syntax in a handshaken form. The input data is conditioned by lexical parsing, which identifies and tags numbers, identifiers, keywords and delimiters. Other inputs such as comments and spaces are removed at this stage. Numbers are accepted in the format of real or integer, and are converted from their character representation to a 32bit real or integer form depending on their type. An EBNF description of the lexical analyser can be found in Appendix B.

D.2.1 Protocol &Lex-&Syntax

The protocol between &Lex and &Syntax, a parallel process to &Read and &Lex, takes the form of a bi-directional data/control stream. From &Lex to &Syntax the stream is a fixed 4 word string followed by a variable size string, length of zero or more. In reply, the stream from &Syntax to &Lex is a single word. The contents of the outgoing stream is determined by the lexical item being sent from &Lex to &Syntax and the status of the lexical analyser. There are 7 different cases, these are illustrated below:

1 Identifier: *lex.ok* / *ident.tag* / line.no / SIZE(str) / str[0] / ... / str[SIZE(str) - 1]
2 Keyword: *lex.ok* / *keywrd.tag* / line.no / SIZE(str) / str[0] / ... /
str[SIZE(str) - 1]
3 Delimiter: *lex.ok* / *delimiter* / line.no / *X*
4 Integer: *lex.ok* / *integer.tag* / line.no / *integer*
5 Real: *lex.ok* / *real.tag* / line.no / *real*
6 Error: *lex.error* / *error.type* / line.no / *error.value*
7 Terminate: *lex.end* / *X* / *X* / *X*

Cases 1 and 2 have variable length strings, because they have to send a character string of unknown size. All other cases are a fixed string of 4 words (32 bits each). The line number (line.no) is determined within the lexical analyser by counting the number of *return* characters which have been read from &Read.

The incoming stream from &Syntax to &Lex is a fixed length of one word. This word is an acknowledgement from &Syntax. The single word can be in one of two states:

1 Syntax OK: *syntax.ok*
2 Syntax error: NOT *syntax.ok*

If an error is detected, either from within &Lex, or from &Read or &Syntax (case 2 above), the remaining data/control stream from &Read is read continuously until &Read terminates. The error string (case 6) is then sent to &Syntax, after which, &Lex terminates.

During non-error conditions, a data/control stream will be received from &Read, conditioned by lexical analysis and sent to &Syntax. Provided a *syntax.ok* acknowledgement is received from &Syntax, &Lex will read another stream of inputs from &Read. This will continue, provided there are no errors, until the terminate string is received. Thereupon, the end string (case 7) is sent to &Syntax, after which, &Lex terminates.

D.3 Syntax analysis

The process &Syntax comprises 4 phases. Each phase has a different function and only the first two phases are strictly concerned with syntax analysis, these two phases run in parallel with &Read, &Lex and &Graph. The second phase is ended by the termination of &Read, which causes &Lex to terminate. Phase 3 terminates &Graph, and runs in parallel with the consecutive processes &Schedule and &Translate. Phase 4 terminates &Translate and initiates, then terminates &Error where necessary. The screen and keyboard processes, &Screen and &Key respectively, run in parallel with &Syntax throughout and are the last two processes to terminate. Table D.1 illustrates the concurrent operation of the compiler.

Phase 1: Initialise
Phase 2: Syntax analysis
Phase 3: User
Phase 4: End

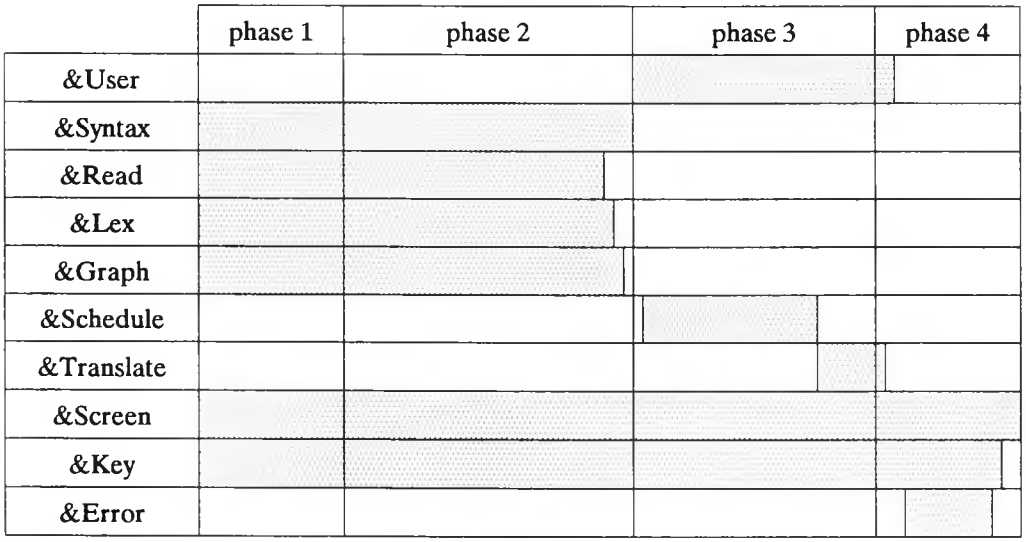


Table D.1 Concurrent compiler operation

Briefly, phase 1 initialises the hash table and symbol table. Phase 2 performs the syntax analysis on the data received from &Lex and controls the construction of the DAG (directed acyclic graph) via &Graph. The DAG represents the DFDL source program. Phase 3 is a user interactive stage. From here, the processor graph construction is controlled via &Schedule and the scheduling of the DAG onto the processor graph is initiated. Phase 3 also controls other functions like schedule display (within &Schedule), processor graph display (within &Schedule) and translation (within &Translate). The final phase, phase 4, displays any source program error that has been detected during read/syntax/lexical analysis and terminates all concurrent processes. &Screen, &Key and &Error then terminate jointly.

D.3.1 Initialise

D.3.1.1 Variables

&Syntax and &User are the two central modules (&User takes over from &Syntax), and directly or indirectly, have control over all other processes. For this reason, they maintain a record of the state of all directly connected modules (except &Screen and &Key). In the initialise phase these are all set to an active state:

state.of.lex := *lex.ok*
state.of.syntax := *syntax.ok*
state.of.graph := *graph.ok*
state.of.schedule := *schedule.ok*
state.of.translate := *translate.ok*

These states are used to prevent communication to other modules once they have terminated, and so avoid deadlock. The syntax state is used to control the program flow for &Syntax, &User and &Error.

Other variables set at this stage are associated with the symbol table and the error routine.

D.3.1.2 Reserved word initialisation

The main purpose of initialise is to set the hash and symbol tables in a ready state for syntax analysis. This is done by pre-entering the keywords into the symbol table, via the hash table:

reserved.word :: = ABS | ACOS | ALOG | AND | ASIN | ATAN | BEGIN
| BYTE | COS | END | ELSE | ELSEIF | EXP | EXPRESSION | EVERY
| FOR | FOREVER | FROM | FUNCTION | INIT | INT | IF | INPUT
| IS | LOG | LN | MAX | MEAN | MED | MIN | MOD | NODE
| NOT | OR | OUTPUT | PROD | PROG | RESULT | REPEAT
| REAL | SIN | SQRT | SGN | SUM | TAN | TABLE | THEN
| VALUE | Z

To begin, all locations in the hash table (SIZE = 200) are set to an empty value (empty = 0), and the start of the next free location in the symbol table is set to 1 (set during initialise phase). An empty value, in a hash table location, signifies there has not been a character string entered via the hash table that maps onto that location. Hence, if all locations are empty, the hash table, and consequently the symbol table, are empty. The reserved words are then entered into the symbol table via the hash table using a hash function.

D.3.1.3 Hash function

The input to the hashing routine is a character string ([]str) of one or more characters. This is processed by the hash function to give a deterministic value of good distribution. The character string []str is read into []buf, []buf = (SIZE str)/str[0]/.../str[(SIZE str) - 1], the hash function produces a value which is the EX-OR of successive character products (except the first product which includes the string size). The hash function is shown below:

```
hash.value := 0
SEQ i = 0 FOR (SIZE str)
  hash.value := hash.value EXOR (buf[i] * buf[i + 1])
```

The result hash.value, is then divided by (SIZE hash.table) and the remainder taken. The remainder lies between 0 and (SIZE hash.table) - 1. The remainder value is used as an address to the hash table.

```
address := hash.value REM (SIZE hash.table)
```

The contents of the location hash.table[address] is either equal to zero (empty value), or lies within the range 1 to (SIZE symtab) - 1. If equal to zero, the character string does not already exist in the symbol table. If non-zero, the character string may exist in the symbol table.

D.3.1.4 Appending the symbol table

Whenever a character string has been mapped onto an empty location in the hash table, the character string may be directly appended to the symbol table. New entries of this type are either new identifiers (*new.ident*) or reserved word initialisations (*new.reserved.wrd*). The character string is entered into the symbol table, preceded by its size. The first symbol table address of the entry is placed in the hash table at the location hash.table[address].

In the second case, where the hash table location is non-empty (symbol table address), the routine examines the symbol table contents at the location beginning symtab[hash.table[address]]. If the character string in the symbol table matches the accessing string, then the character string already exists in the symbol table. This represents a repeated identifier (*ident*) or repeated reserved word (*reserved.wrd*), and would be an erroneous case for the initialisation of reserved words.

A third situation occurs when the character strings do not match. This is called a collision; different character strings map onto the same hash table location. Collisions are overcome either by re-hashing or chaining. In this case we use chaining.

D.3.1.5 Chaining

Chaining caters for hash collisions, and unlike other methods prevents bunching in the hash table or the re-building of the symbol table. Different character strings which map onto the same hash table address are linked together in a chain (single linked list). At each symbol table entry, a word is reserved for the chain address. In effect, the chain location serves as an extension to the hash table. Chain locations are initially set to empty.

D.3.1.6 Symbol table format

The first four fields of a symbol table entry are identical for both reserved words and identifiers. Identifiers entries have an additional three fields to hold dimension and data structure information.

(a) Reserved words

Field 1: Size.field holds the size of the character string.
Field 2: Char.field is a variable size (size defined in Size.field)
Field 3: Chain.field, links entries with the same hash table address.
Field 4: Type.field holds the type of entry, *reserved.word.types* (400 - 499).

Reserved word symbol table format is illustrated below:

size.field	char.field	chain.field	type.field

As an example, the reserved word NOT would be entered into the symbol table as shown below:

3	N	O	T	empty	delim.not

(b) Identifiers

Identifier symbol table entries are:

Field 1: Size.field holds the size of the character string.
Field 2: Char.field is a variable size (size defined in Size.field).
Field 3: Chain.field, links entries with the same hash table address.
Field 4: Type.field holds the type of entry, *ident.type*. (500 - 599).
Field 5: Col.field, holds column size.
Field 6: Row.field, holds row size.
Field 7: Graph.field, holds pointer to the data flow graph.

The format of the *ident.type* entry is illustrated below:

size.field	char.field	chain.field	type.field	col.field	row.field	graph.field

ident.type :: = input.type | node.type | output.type
| const.type | function.type | subscript.type

The three additional fields, col.field, row.field and graph.field, are used to hold the spatial dimensions (column and row) and a pointer to the DAG. The pointer links the symbol table entry to the DAG, pointing to the vertex (or node) which has an index of zero. For example an input "read" (dimension [2][4]), whose vertex corresponding to element read[0][0] is at location 605 in the graph structure, would be entered into the symbol table as shown below:

4	r	e	a	d	empty	input.type	2	4	605

Where column, or column and row, dimensions are not specified, a default value of 1 is assigned.

D.3.1.7 Symbol table limits

The symbol table is of a finite size (2000). Should overflow occur it will result in the error message:

"Implementation limit; symtab[ ] is full"

D.3.2 Syntax analysis

Syntax analysis is achieved using a top-down, syntax analyser. The grammar of DFDL is free from left recursion and can be parsed without resorting to backtracking. Parenthesised expressions do cause problems however, as these lead to self embedding which is difficult to handle using top-down analysis. This is overcome by storing complete assignment statements, whereupon they are decomposed into simple expressions (operator, left operand, right operand) prior to parsing. This process allows us to remove all parentheses, where upon top-down analysis can be performed on each simple expression/assignment.

The syntax of DFDL does not employ any operator-precedence, and relies entirely on the use of parentheses to determine the order of execution in multiple operator expressions. Even identical operators in the same expression have to be ordered. This results in an unambiguous structure (precedence relationships) and allows the user to experiment with different structures for the same program.

The complete syntax for DFDL is expressed in Appendix B. Certain operations and functions are not implemented by the current version of the syntax analyser, which implements a restricted form of DFDL referred to as DFDL(mod.state 1/90). Differences between DFDL(mod.state 1/90) and the complete DFDL syntax are in the following areas:

1 User defined functions: not available
2 Value initialisation: not available but delay variables are set to 0.0.
3 Conditional assignment: not available
4 Integer expressions: simple expressions only
5 Functions: restricted set

D.4 Syntax analyser description

To supplement the BNF description of the syntax (Appendix B), flow diagrams are used the describe the DFDL(mod.state 1/90) implementation. A top-down approach is employed, using rectangular boxes to express terminal items, and round-edged rectangular boxes to express non-terminal items. Where necessary, diagrams are supplemented by control information.

The syntax program flow is divided into 2 main non-terminal blocks, declaration and assignment , these are illustrated by Figure D.1.

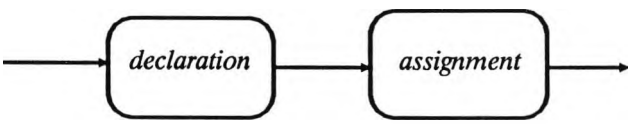


Figure D.1 Program flow

D.4.1 Declaration flow

The declaration block is described first. This block is expanded as illustrated by Figure D.2. All the non-terminal blocks after PROG are optional except output. There must be at least one output in a source program.

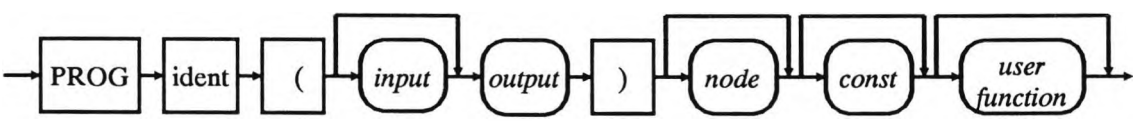


Figure D.2 Declaration flow

D.4.1.1 Input, node and output flow

Within the input, node and output blocks, the parsing action is similar. For each identifier, the product of column size and row size is checked for a maximum limit, which is defined as max.array.size (65536 elements). The minimum limit is 1. If max.array.size is exceeded, an error message is given:

"Implementation limit; array size too big"

The maximum array size is dictated by the size of the index storage in the graph (DAG) data structure. The zeroth index element (index = 0) of an array, has its attributes sent to &Graph. &Graph builds a vertex in the graph data structure which represents this element. The format of the attribute string is as follows:

mv.dec.type / *data.type* / *label* / *index*

mv.dec.type ::= mv.input.dec | mv.node.dec | mv.output.dec
data.type ::= BYTE | INT16 | INT32 | REAL32
label ::= 1 | ... | (SIZE symtab) - 1
index ::= 0 | ... | max.array.size

Note: When the identifier represents a node the data type is REAL32 only, because DFDL's internal arithmetic is real throughout. Other data types (BYTE, INT16, INT32) are only allowed for external inputs and outputs. The *label* is the address in the symbol table where the identifier resides, this is used by &Graph as a back reference to the identifier.

Once the zeroth index element of an array is built, &Graph replies with the address in the data structure where the vertex is located. This address is stored in the symbol table at the location graph.field (section 3.1.6b). Subsequent elements (index > 0), if any, are built in a similar fashion, adjacent to their preceding vertex. When required, all vertices can be located via the zeroth index element.

Figures D.3 through to D.5 illustrate the input, node and output flows respectively. Note, the flow for node does not include a *data.type* block.

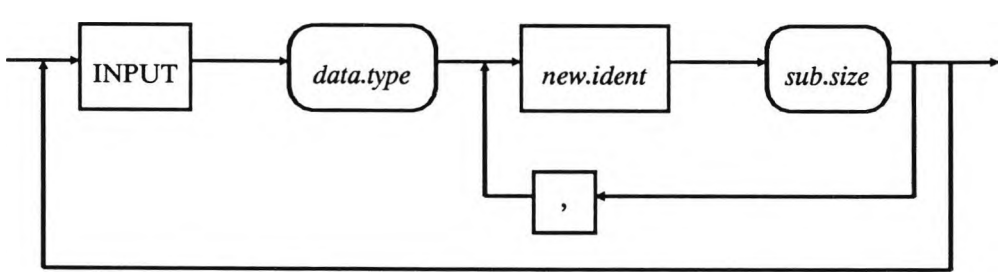


Figure D.3 Input flow

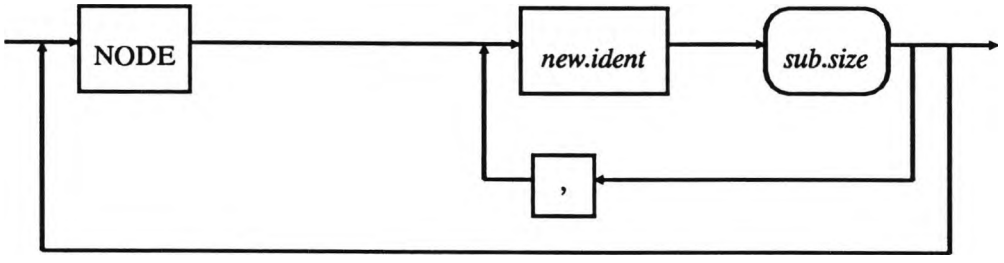


Figure D.4 Node flow

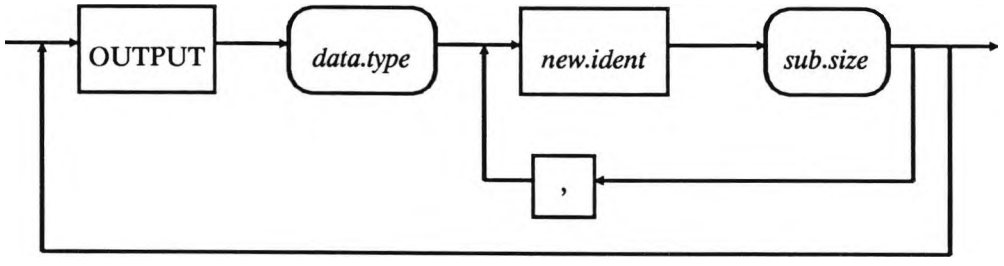


Figure D.5 Output flow

D.4.1.2 Data type flow

The flow for *data.type* is given below. Data types INT16, INT32 and REAL32 are formed by reading the reserved words INT or REAL, followed by the integer 16 or 32. Only the correct combinations of reserved word/integer are accepted.

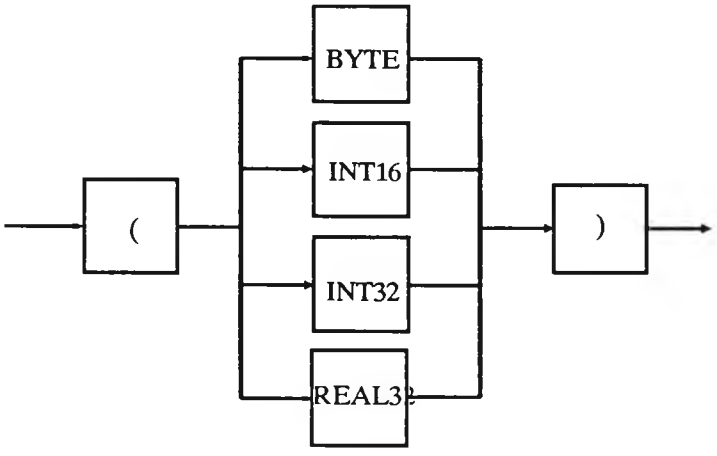


Figure D.6 Data type flow

D.4.1.3 Subscript size flow

The flow for *sub.size*, the array subscript size, is given below. The column size precedes the row size. The terms column and row represent the two planar dimensions, where column is the first subscript and row the second subscript. Subscript sizes, and the product of the subscript sizes, have a valid range from 1 to max.array.size (equal to 65536).

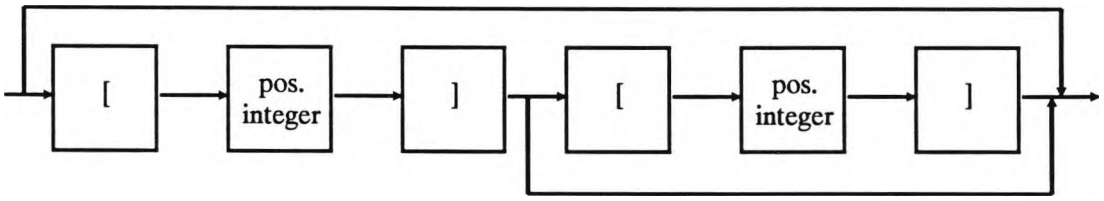


Figure D.7 Subscript size flow

D.4.1.4 Constant flow

Constants are constructed in a similar fashion to inputs, nodes and outputs. Constant declaration can take any one of three forms:

The first of these forms is a table of real values. A table comprises 'row size' of real strings, where each real string comprises 'column size' of real values. The numbers of real strings, and real values within the strings, must correspond to their respective *sub.size* declaration.

The second form, is that of an expression. Values are assigned to the constant identifier during the initialise section, after which, they cannot be altered. An constant declaration of this form is labelled by assigning the keyword EXPRESSION to the identifier.

The final form allows a single real value to be assigned to an identifier. The identifier may be a scalar or an array. If the constant identifier is an array, every element in that array will be assigned the single real value.

In all three cases the subscript size is checked for a valid range. If the product of the column and row subscripts exceeds max.array.size (65536) the following message is given:

"Implementation limit; array size too big"

The maximum array size is dictated by the size of the index storage in the graph data structure (see DFDL task model). The zeroth index element (index = 0) of an array, has its attributes sent to &Graph. &Graph builds a vertex in the graph data structure (data flow graph) representing this element. The format of the attribute string is as follows:

mv.dec.type / real / label / index

mv.dec.type ::= mv.const.dec | mv.const.exp
label ::= 1 | ... | (SIZE symtab) - 1
index ::= 0 | ... | max.array.size

Note: Where the declaration type is *mv.const.exp*, the second element of the string, normally used to convey the real value, is empty.

Note: Constant identifier values have a data type of REAL32 only, because DFDL's internal arithmetic is real throughout. Other data types (BYTE, INT16, INT32) are only allowed at inputs or outputs. The *label* is the address in the symbol table where the identifier resides, this is used by &Graph as a back reference to the identifier.

Once the zeroth index element of an array is built, &Graph replies with the address in the data structure where the vertex is located. This address is stored in the symbol table at the location graph.field (see Initialise(phase 1)). Subsequent elements (index > 0), if any, are built in a similar fashion, adjacent to their preceding vertex. When required, all vertices can be located via the zeroth index

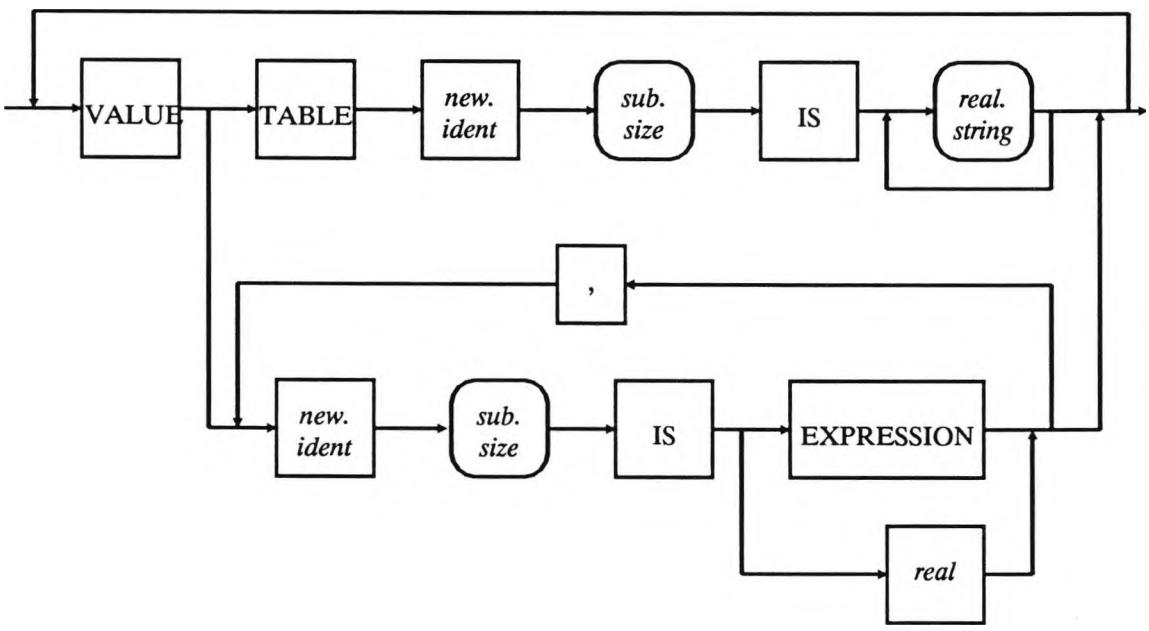


Figure D.8 Constant flow

element. A detailed description of the vertex and graph data structure is to be found in the chapter on the DFDL task graph.

D.4.1.5 Real string flow

Real values are enclosed by square brackets and separated by commas.

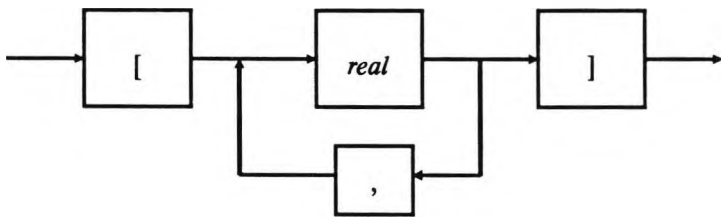


Figure D.9 Real string flow

D.4.1.6 User defined function flow

User functions are not implemented in DFDL(mod.state 1/90). Attempts to use user functions will be met with the message:

"Function not installed yet!"

D.4.2 Assignment flow

The second major block within program flow is called assignment flow. This block comprises the initialise block (optional) and the repeat block.

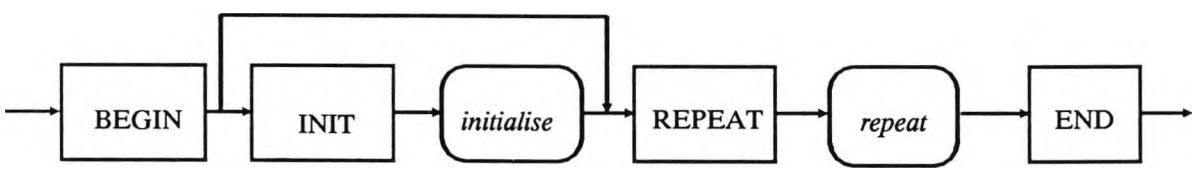


Figure D.10 Assignment flow

D.4.2.1 Initialise flow

The initialise block specifies the source program code which, when called, is executed once only. The current implementation, DFDL(mod.state 1/90), does not include initialise. Attempts to use initialise will be met with the message:

"Initialise not installed yet!"

D.4.2.2 Repeat flow

The repeat block specifies the source program code which is repeated, and only stops when the program is terminated. The syntax of the repeat block has three main stages: parse, check and build. These are described individually.

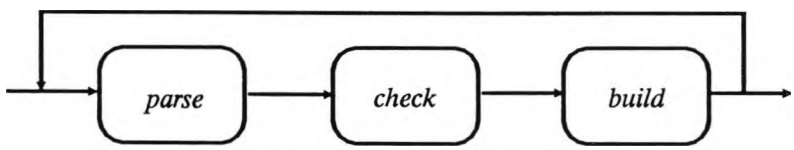


Figure D.11 Repeat flow

D.5 Parse flow

D.5.1 Stack construction

Parse is one of the main syntax routines. It reads complete assignment statements from &Lex, checking the syntax as it does so. Assignment statements are sub-divided into simple expressions (one/two operands and one operator) and each simple expression is placed on a different level of an expression stack. The levels are doubly linked to each other to allow the routine to travel up and down the stack.

The depth of the stack is currently 20 levels, this allows up to 19 pairs of parentheses to be used in an assignment statement. If this is exceeded, the following error message is given:

"Implementation limit; expression too deeply nested"

The lowest level (level 0) contains the assignment statement's object and assignment operator, while the 19 upper levels accommodate simple expressions. The width of each level is 30 words. Each level is divided into 4 main fields:

Each of these fields begins with their respective reference location; rsp.loc, op.loc, left.loc and right.loc. The widths of the fields are as shown below:

rsp.field - 1 word
op.field - 1 word
left.field - 14 words
right.field - 14 words

D.5.1.1 RSP field

The rsp (return stack pointer) field contains a pointer which links a simple subordinate expression to its parent expression. The pointer is always directed down the stack, towards the lowest level. At the lowest level (level 0) the rsp field is empty. All active levels, except level 0, have an rsp field which contains a valid pointer. The valid range of the pointer is between 0 and (tos - 1). Where tos (top of stack) indicates the highest level in use.

D.5.1.2 Operator field

The op field contains an operator which belongs to the simple expression. The operator is either an assignment operator (treated as a dyadic operator), monadic operator, function, dyadic operator or no-op. The operator describes how, in the case of a dyadic operator, the left and right operands interact, and how, in the case of a monadic operator, function or no-op, the left operand is processed. The set of valid operators for DFDL(mod.state 1/90) is given below:

op ::= *assignment* | *monadic.op* | *dyadic.op* | *nop*
assignment ::= :=
monadic.op ::= -
dyadic.op ::= + | - | * | / | \ | **
function ::= ABS | ACOS | ASIN | ATAN | COS | EXP | LN | LOG
| SGN | SIN | SQRT | TAN| MEAN | SUM | PROD | ALOG

Note: nop is used as an operator in simple expressions, where the operand was preceded by the monadic operator + , or the simple expression stems from a single, parenthesised operand.

D.5.1.3 Left/Right operand field

The left and right fields are almost identical, any differences will be commented on, as and when they occur. The fields are used to hold the operands belonging to a simple expression. The types of valid operand are:

1 real literal
2 input
3 node
4 output
5 constant
6 fsp (forward stack pointer)
7 empty (right field only)

Input, node, output and constant types are all treated the same in the Parse routine, and are classed as identifier types.

left.operand.type ::= *real* | *ident* | fsp
right.operand.type ::= *real* | *ident* | fsp | empty

D.5.1.4 Operand is a real literal

When the operand is a real literal only the first two sub-fields of the operand are used to hold information. These fields are shown below:

ab.type	ab.value	col.field	row.field	del.field

ab.type holds the tag which identifies the operand as a real literal, and ab.value holds its real value. The remaining fields are set to dont.care (shown as X), this allows real literals to be used in replicated expressions.

real.type	*real*	X	X	X

For example a real literal of 0.453 would be stored as:

real.type	0.453	X	X	X

D.5.1.5 Operand is an identifier

When the operand is an input, node, output or constant, it is represented by its identifier. This type of operand uses all five sub-fields. The sub-fields are shown below:

ab.type	ab.label	col.field	row.field	del.field

ab.type holds the tag which identifies the operand as an identifier, and ab.label holds the identifier's label (symbol table reference). The three remaining sub-fields contain the column, row and delay subscript attributes. Each attribute sub-field is 4 words long, and has the following format:

ab.ident	ab.start	ab.size	ab.step

ab.ident holds the label (symbol table reference) of the subscript identifier when the subscript is repetitive. For fixed subscripts this is set to dont.care. Location ab.start holds the beginning index of the subscript, ab.size holds the number of iterations, and ab.step holds the step size per iteration.

A default setting for the subscript fields (col.field, row.field, del.field) is made when no subscript is given, this is shown below:

X	0	1	1

The default setting signifies a fixed subscript, beginning at index 0, for 1 iteration, and a step size of 1.

When the subscript to an identifier is defined, the appropriate values are entered into the requisite locations. The examples below illustrate the correspondence between a DFDL description of an operand's subscript (fixed and repetitive), and the operand's subscript as held in the operand field. The label values are for example only.

(i) the operand 'in[2][10]' is a single element of an array called "in", symbol table label = 255, which exists at column 2, row 10

ident	255	X	2	1	1	X	10	1	1	X	0	1	1

(ii) the operand 'out' represents a scalar called "out", symbol table label = 153

ident	153	X	0	1	1	X	0	1	1	X	0	1	1

(iii) the operand 'old[k FROM 4 FOR 2]Z[j FROM 8 FOR 4 EVERY -2]' is an iteration (2 * 4 times) of an array called "old", symbol table label = 236. The label for k = 425, and the label for j = 432.

ident	236	425	4	2	1	X	0	1	1	432	8	4	-2

D.5.1.6 Operand is a forward stack pointer

Both the left and right operands are used to hold a forward stack pointer. A fsp (forward stack pointer) is directed from a parent expression (simple expression) to a subordinate expression (simple expression). A fsp is always directed up the stack towards the tos (top of stack). All levels that are in use contain fsps, except leaf expressions; those which have no subordinate expressions. The valid range of a fsp is from 1 to tos. This type of operand only uses the first three words of the left/right operand field, as illustrated below:

ab.type	ab.sp	ab.addr	X ...

ab.type holds the tag which identifies the operand as an forward stack pointer, and ab.sp holds the fsp's pointer. Location ab.addr is used as a temporary location for the operator's graph data structure address. This is initially set to don't care (shown as X). The remaining fields are also set to dont.care.

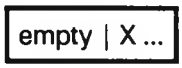
#.type	*integer*	X	X

For example an fsp pointing to level 7 is represented as:

#.type	7	X	X

D.5.1.7 Operand is empty (right only)

An empty operand is denoted by an empty tag in the ab.type location. All other fields are don't care (shown as X).



D.5.2 Stack operation

The stacking of simple expressions is controlled by five different delimiters; (,), ..., := and eol. The final delimiter, eol (end of line), is created from detecting an increase in line number from &Lex.

The left hand parenthesis signifies the start of a new simple expression. Its occurrence causes the current operand location to be formatted as an fsp. The fsp's pointer is set to the number of the next free level, which is the current top of stack (tos). The top of stack level (subordinate level), is back linked to the parent level, by entering the parental level in the subordinate's rsp (return stack pointer) location. The tos is then incremented.

The assignment operator has the same effect as the left hand parenthesis, but only occurs when moving from level 0 to 1.

The right hand parenthesis signifies the end of a simple expression. Its occurrence causes a move down the stack, to the level pointed to by the current level's rsp.

A record is kept of the difference between the number left hand parentheses and right hand parentheses. A counter (initially set to zero) is incremented for left hand parentheses, and decremented for right hand parentheses. An error message is given if the parentheses count goes negative or is non-zero at the end of the assignment statement.

"Expression has an excessive number of ')'s"

"Expression has an excessive number of '('s"

The continuation delimiter (...) allows the assignment statement to be spread across more than one line, and prevents the eol delimiter from signifying the end of the assignment statement.

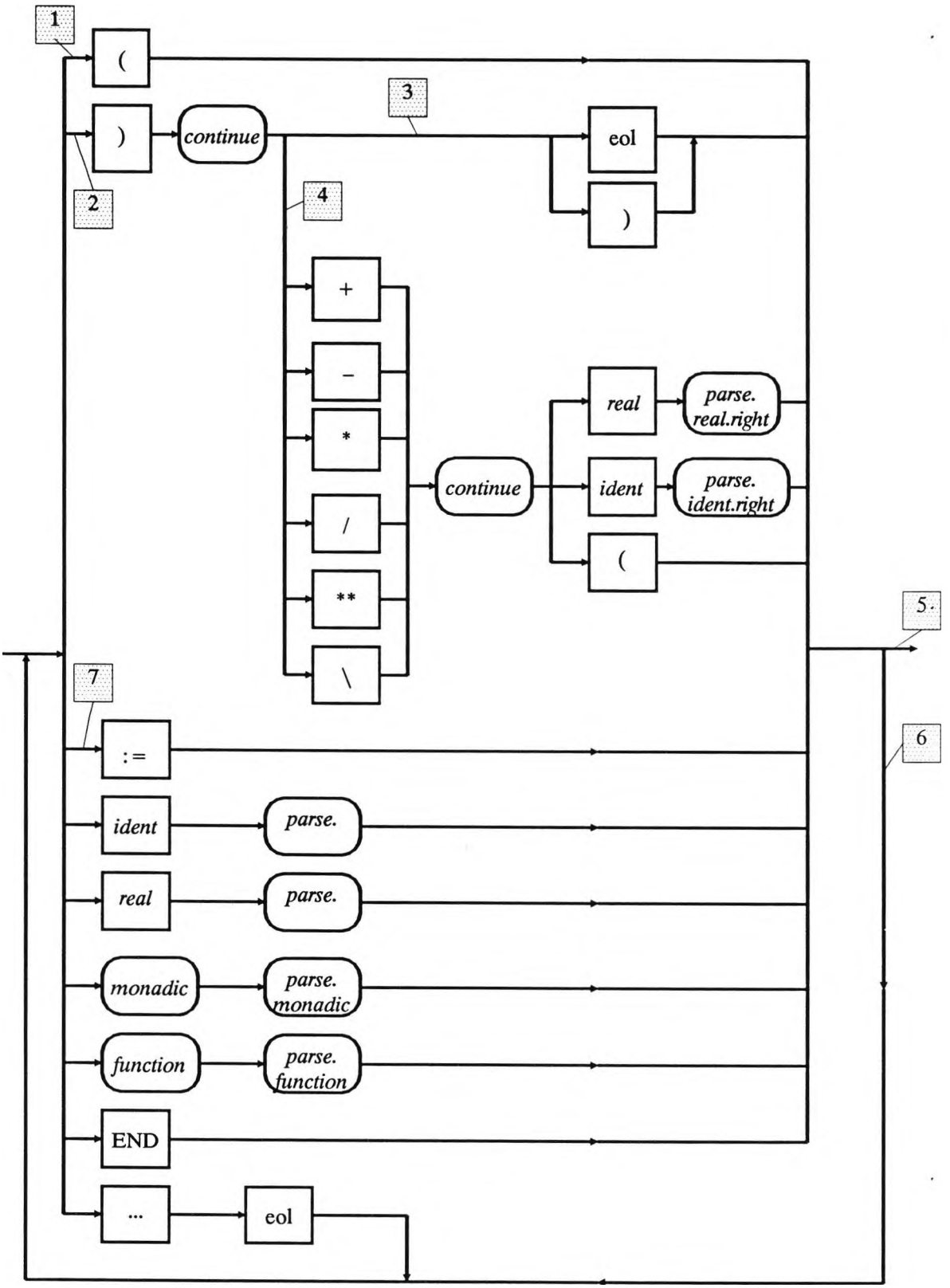


Figure D.12 Parse flow

D.5.3 Parse flow

The parse flow is described using several diagrams. These are supplemented by control flow information regarding the stack. The parse flow is illustrated by Figure D.12.

- 1. Left hand operand location empty, if not, then right hand operand location empty.
  0 < stack pointer < stack depth.
  Move up stack.
  Increment parentheses count.

- 2. Left hand operand location not empty, operator location not empty.
  1 < stack pointer < = stack depth.
  Move down stack.
  Decrement parentheses count.

- 3. If operator location empty, then right hand operand location empty also.

- 4. Operator location empty.

- 5. End of assignment statement or END
  stack pointer = 0

- 6. Not end of assignment and not END.

- 7. Left hand operator is identifier, right hand operator location empty.
  stack pointer = 0
  Move up stack.

D.5.3.1 Continue flow

The continue delimiter is valid between most syntactically complete items.

- 1. old line number : = line number

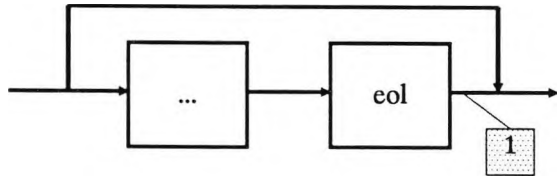


Figure D.13 Continue flow

D.5.3.2 Parse left identifier flow

Simple expressions which begin with an input, node, output or constant, all start with an identifier tag, followed by their identifier label (originating from the symbol table). The tag and label are placed in the left hand operand location. The expression is parsed as shown below in Figure D.14.

- 1. Left hand operand location not empty.
  Right hand operand location empty.
- 2. Left hand operand location empty.
- 3. No dyadic operator, operator becomes a no-op.
- 4. Operator location is empty.

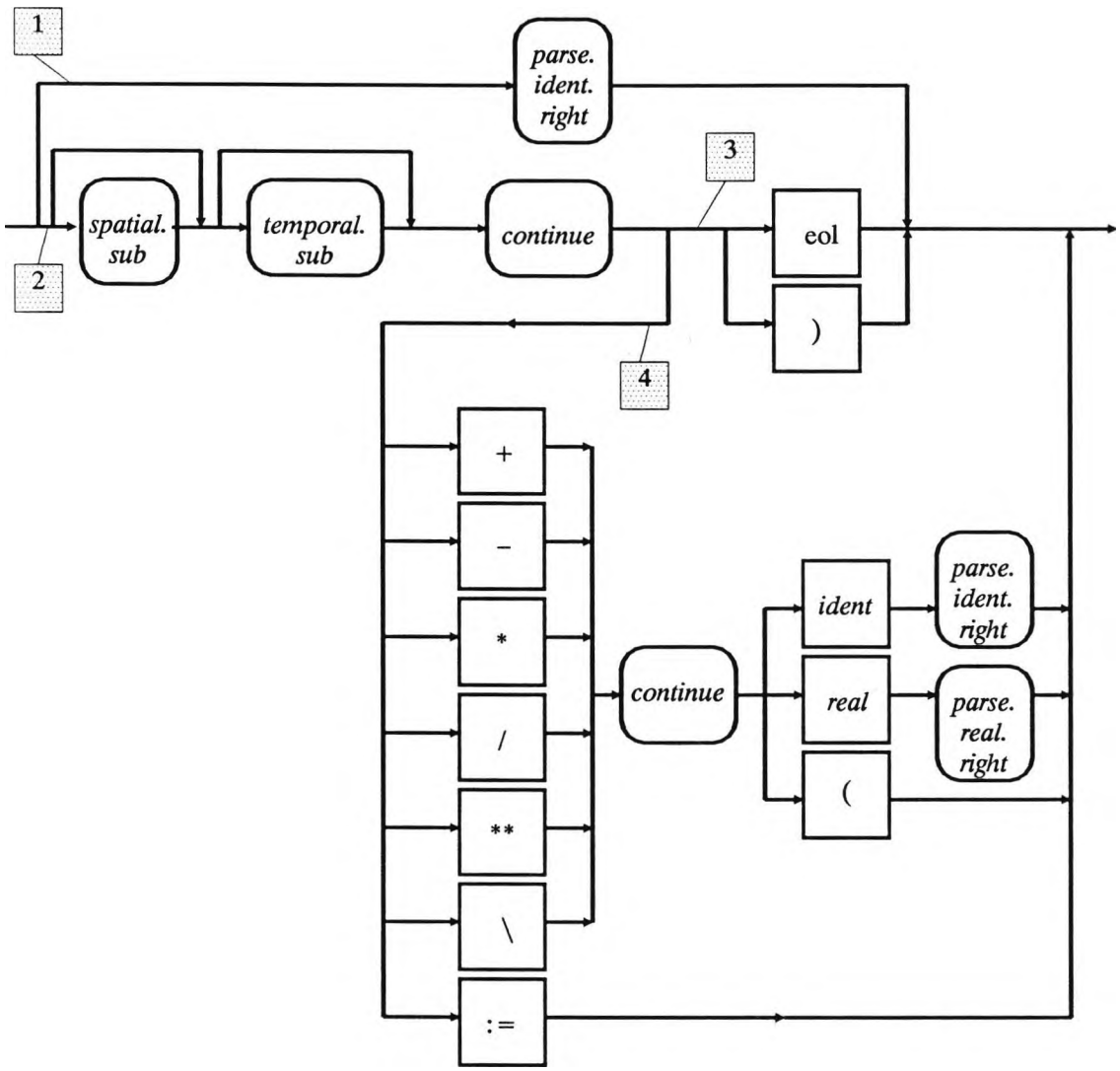


Figure D.14 Parse left identifier flow

D.5.3.3 Parse left real flow

Simple expressions which begin with a real literal, start with a real tag, followed by a real value. Tag and value are placed in the left hand operand location. The expression is parsed as shown below.

- 1. Left hand operand location not empty.
  Right hand operand location empty.
- 2. Left hand operand location empty.
- 3. No dyadic operator, operator becomes a no-op.
- 4. Operator location is empty.

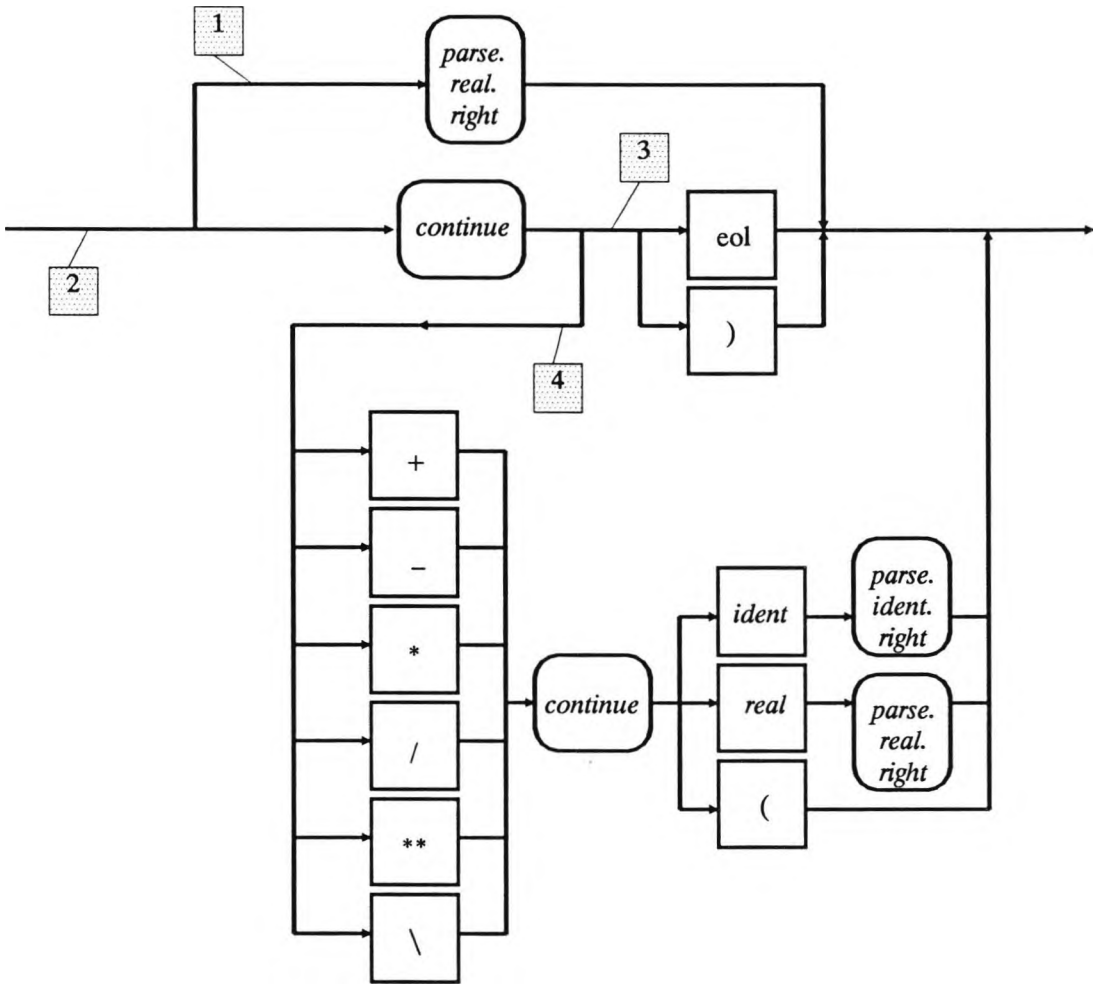


Figure D.15 Parse left real flow

D.5.3.4 Parse right identifier flow

An operand which is an input, node, output or constant, and which occurs immediately after a dyadic operator (or continue), has its identifier tag and identifier label placed in the right operand location. The remainder of the expression is parsed as shown below in Figure D.16.

- 1. Left hand operand location not empty.
  Right hand operand location empty.

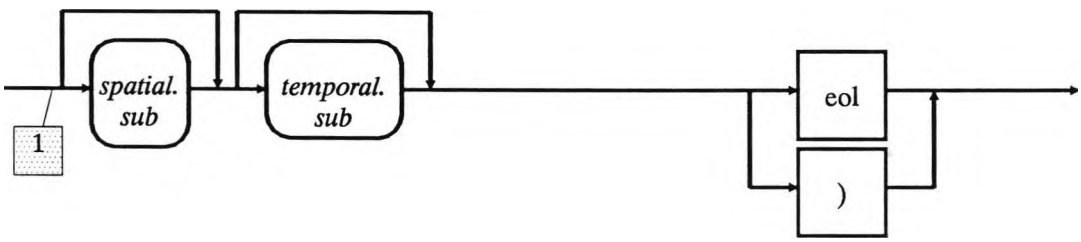


Figure D.16 Parse right identifier flow

D.5.3.5 Parse right real flow

An operand which is a real literal, and which occurs immediately after a dyadic operator (or continue), has its tag and value placed in the right hand operand location. The remainder of the expression is parsed as shown below in Figure D.17.

- 1. Left hand operand location not empty.
  Right hand operand location empty

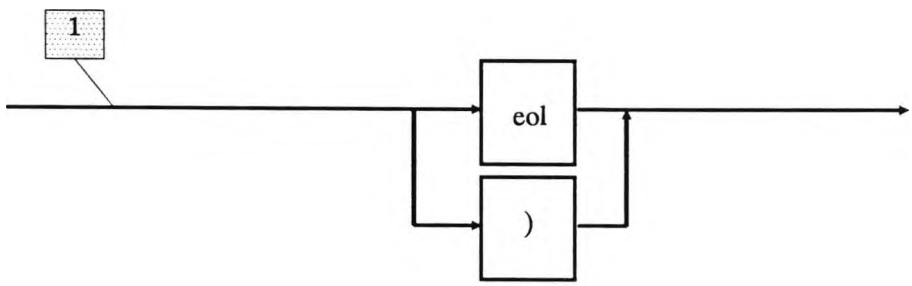


Figure D.17 Parse right real flow

D.5.3.6 Functions and parse function flow

Figure D.18, below, combines the flow for functions and the operand/expression immediately following the function.

- 1. Left hand operand location is empty.
  Right hand operand location is empty.
  Operator location is empty.

- 2. Function not implemented by
  DFDL(mod. state 1/90).

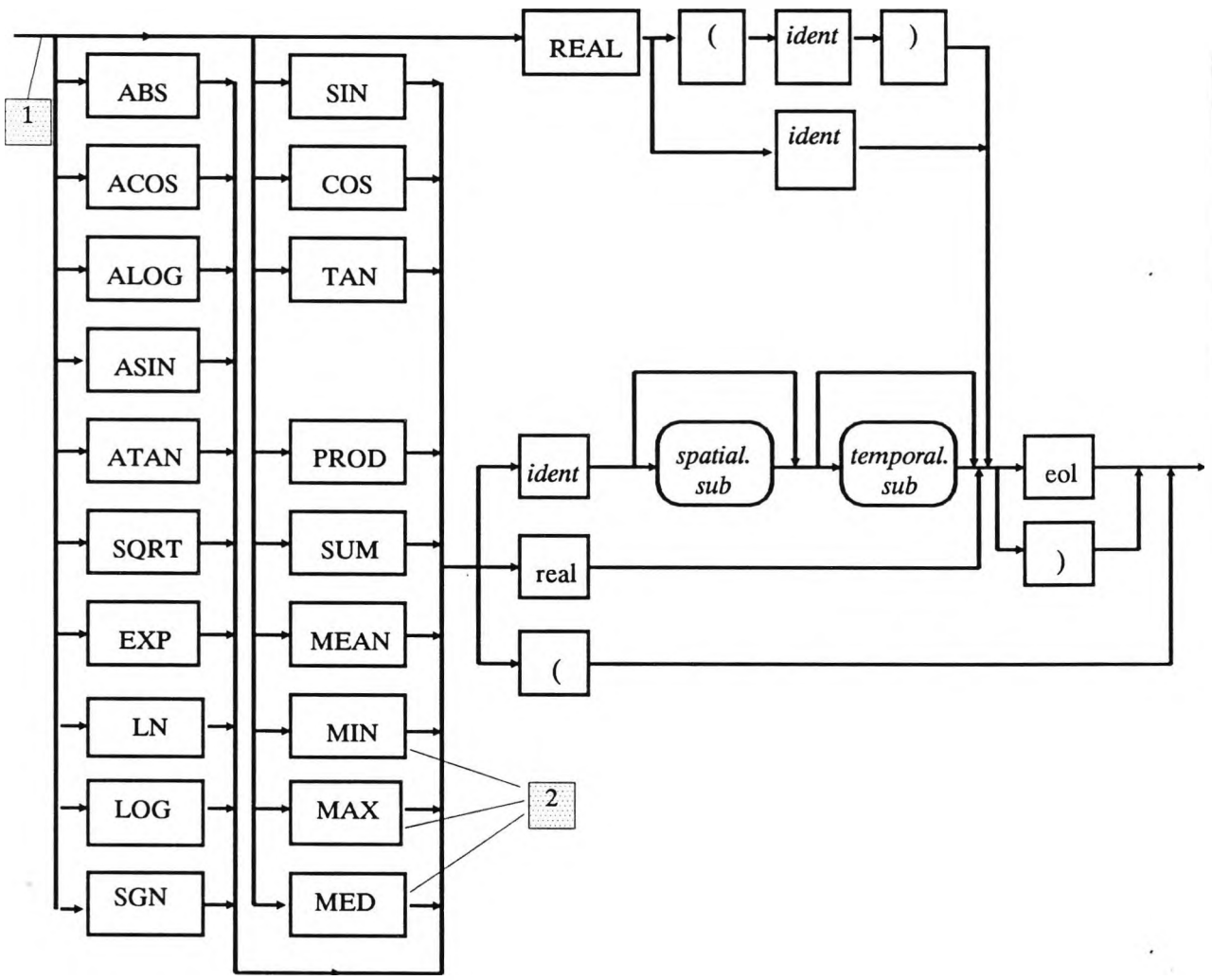


Figure D.18 Function flow

D.5.3.7 Monadic operator and monadic operator parse

Figure D.19, below, combines the flow for monadic operators and the operand/expression immediately following the monadic operator.

- 1. Left hand operand location is empty.
  Right hand operand location is empty.
  Operator location is empty.

- 2. Operator becomes a no-op.

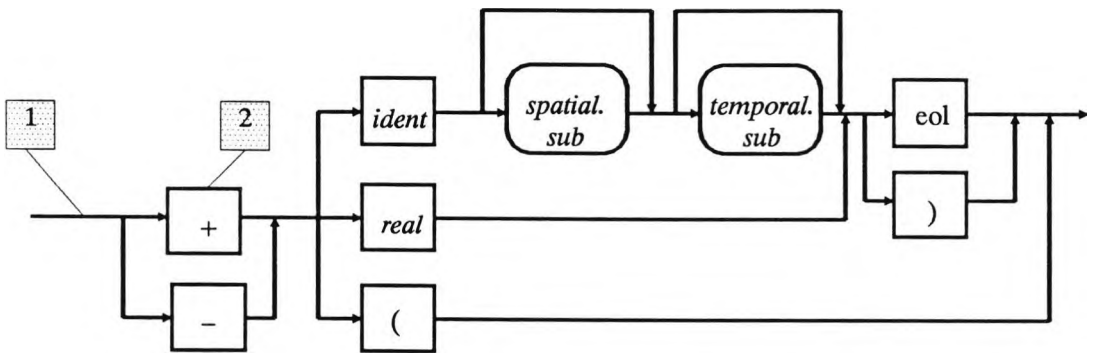


Figure D.19 Monadic flow

D.5.3.8 Spatial subscript flow

A spatial subscript describes the column and row indices. Valid column/row indices are non-negative integer values less than the column/row dimension size.

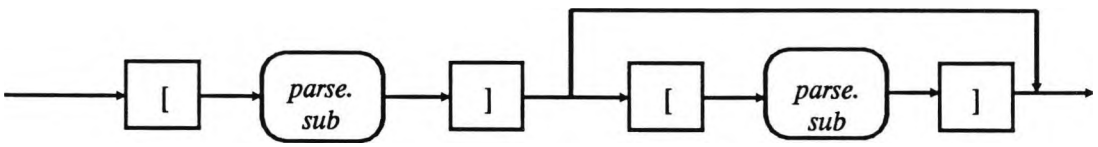


Figure D.20 Spatial subscript flow

D.5.3.9 Temporal subscript flow

A temporal subscript begins with the keyword Z. It defines the unit delay applied to the preceding operand. Valid delays are non- negative integer values not greater than the maximum delay (currently 1000).

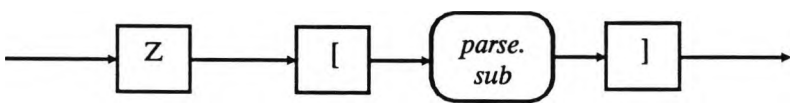


Figure D.21 Temporal subscript flow

D.5.3.10 Parse subscript flow

Subscripts are either fixed or repetitive. Fixed types evaluate to a single non-negative integer value. Repetitive types evaluate to a series of non-negative integer values. The series starts FROM the *start,* goes on FOR the *size,* and advances EVERY *step*. Where the step is not declared, a default value (+ 1) is inserted.

- 1. Fixed subscript.
- 2. Repetitive subscript.
- 3. Non-negative integer.
- 4. Positive integer.
- 5. Default step is + 1.
- 6. Non-zero integer.

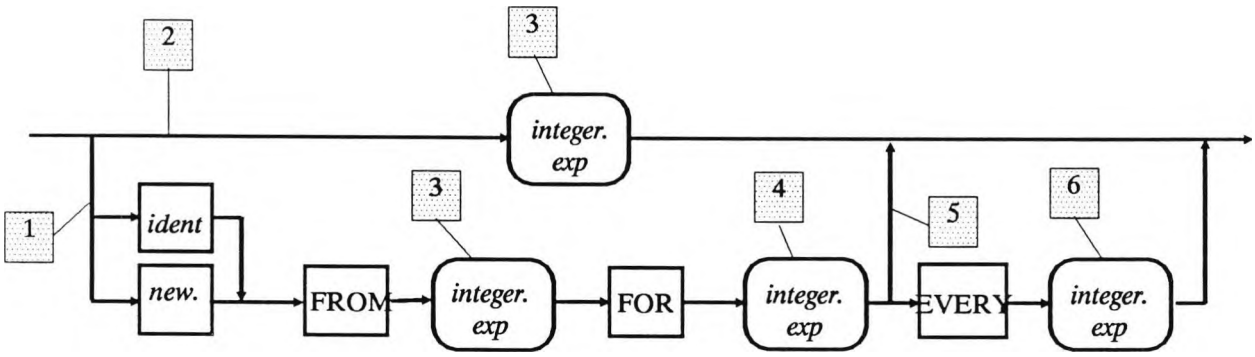


Figure D.22 Parse subscript flow

D.5.3.11 Integer expression flow

Integers in the DFDL(mod. state 1/90) implementation are evaluated by simple expression. The flow of the integer expression is illustrated by Figure D.23. All the arithmetic operators are integer operators.

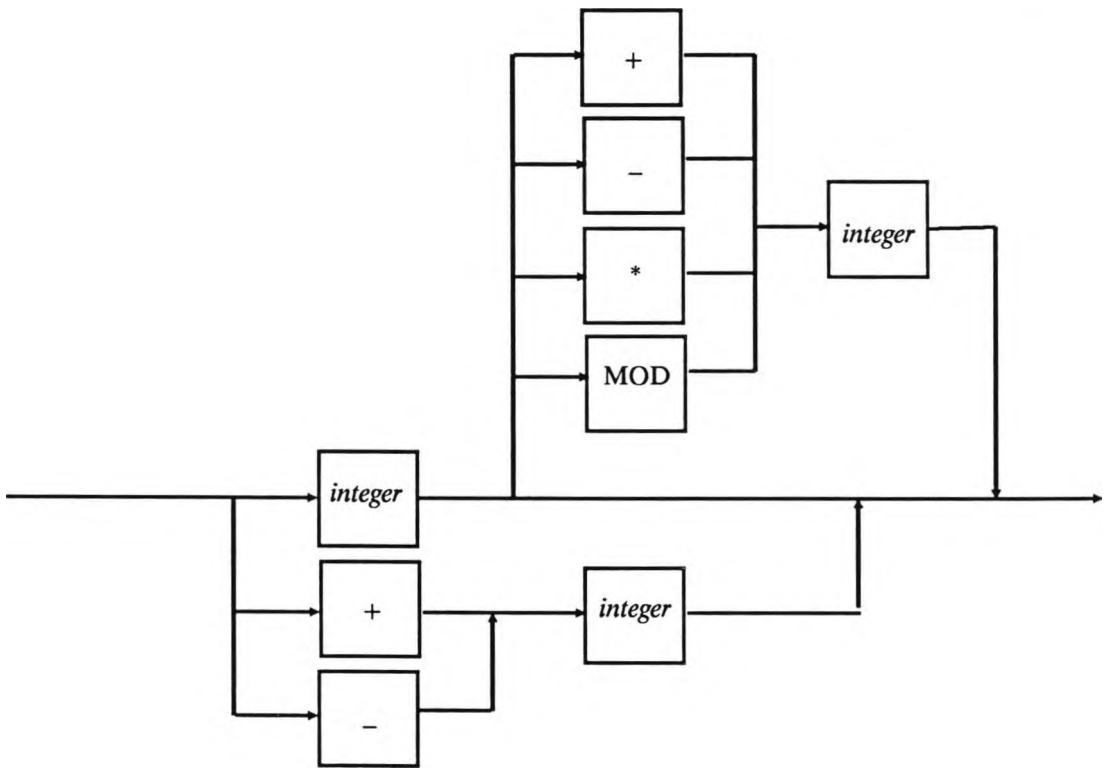


Figure D.23 Integer expression flow

References

- Abramsky S. and Bornat R. (1982).
  In: Pascal-M: A language for distributed systems, QMC CSL 326, Queen Mary Collage Computer Systems Laboratory.

- Ackerman W.B. (1982).
  Dataflow languages. In: IEEE Computer, vol.15, no.2, pp. 15-25.

- Ackerman W.B. and Dennis J.B. (1979).
  In: VAL-A value- oriented algorithmic language: preliminary reference manual, MIT Laboratory for Computer Science Technical Report, TR-218, MIT, Cambridge, Mass.

- Adam T.L., Chandy K.M. and Dickson J.R. (1974).
  A comparison of list schedules for parallel processing systems. In: Commun. ACM, vol.17, no.12, pp.685-690.

- Aho A.V., Hopcroft J.E. and Ullman J.D. (1974).
  In: The design and analysis of computer algorithms. (Reading, Mass: Addison-Wesley).

- Allen J. (1985).
  Computer architectures for digital signal processing. In: Proc. IEEE, vol.73, no.5, pp. 852-873.

- Andrews G.R. and Schneider F. (1983).
  Concepts and notation for concurrent programming. In: ACM Computing Surveys, vol.15, no.1, pp3-44.

- Annaratone M., Arnould E., Kung H.T. and Menzilcioglu O. (1986).
  Using Warp as a supercomputer in signal processing. In: IEEE Proc. International Conference on Acoustics, Speech and Signal Processing ICASSP-86, pp.2895-2898.

- Annaratone M., Arnould E., Gross T., Kung H.T., Lam M., Menzilcioglu O. and Webb J.A. (1987).
  The Warp computer: architecture, implementation and performance. In: IEEE Trans. Comput., vol.36, no.12, pp.1523-1537.

- Arvind, Gostelow K.P. and Plouffe W. (1978).
  In: An asynchronous programming language and computing machine. Technical Report TR114a, Department of Information and Computer Science, University of California at Irvine.

References

- Arvind and Kathail V. (1981).
  A multiple processor dataflow machine that supports generalised procedures. In: Proc. 8th Annual Symposium on Computer Architecture.

- Ashcroft E.A. and Wadge W.W. (1977).
  LUCID, A non-procedural language with iteration. In: Commun. ACM, vol.20, no.7, pp. 519-526.

- Barnes G.H. (1968).
  The ILLIAC IV computer. In: IEEE Trans. Comput., vol.17, no.8, p.746.

- Backus J. (1978).
  Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. In: Commun. ACM, vol.21, no.8, pp.613-641.

- Bertsekas D.P. and Tsitsiklis J.N. (1989).
  In: Parallel and distributed computation. (New Jersey: Prentice-Hall).

- Bhuyan L.N., Yang Q. and Agrawal D.P. (1989).
  Performance of multi-processor interconnection networks. In: IEEE Computer, vol.22, no.2, pp.25-37.

- Bokhari S.H. (1988).
  Partitioning problems in parallel, pipelined and distributed computing. In: IEEE Trans. Comput., vol.37, no.1, pp.48-57.

- Bornat R. (1979).
  In: Understanding and writing compilers: A do-it-yourself guide. (Basingstoke: Macmillan).

- Brinch-Hansen P. (1972).
  Structured multiprogramming. In: Commun. ACM, vol.15, no.7, pp.574-578.

- Brinch-Hansen P. (1975).
  The programming language Concurrent Pascal. In: IEEE Trans. Software Eng., vol.1, no.2, pp.199-207.

- Burstall R.M., MacQueen D.B. and Sandella D.T. (1980).
  HOPE: An experimental applicative language. In: Internal report CSR-62-80, Department of Computer Science, University of Edinburgh.

References

- Chamberlin D.D. (1971).
  Parallel implementation of a single assignment language. Ph.D. thesis: Stanford University, Computer Science Dept.

- Chambers F.B., Duce D.A. and Jones G.P. (1984).
  In: Distributed Computing. (London: Academic Press).

- Clark W. (1952).
  In: The Gantt chart. 3rd edition (London: Pitman).

- Coffman E.G. (1976).
  In: Computer and job-shop scheduling theory. (New York: John Wiley & sons).

- Coffman E.G. and Graham R.L. (1972).
  Optimal scheduling for two-processor systems. In: Acta Informata., vol.1, pp.200-213.

- Cohen D.A. (1976).
  Basic techniques of combinatorial theory. (New York: John Wiley & Sons).

- Conway R.W., Maxwell W.L. and Miller L.W. (1967).
  Theory of scheduling. (Reading, Mass: Addison-Wesley).

- Cook S. (1971).
  The complexity of theorem proving. In: Proc. 3rd Annual ACM Symposium on Theory of Computing, 1971, pp.151-158.

- Cvetanovic Z. (1987).
  The effects of problem partitioning, allocation and granularity on the performance of multi-processor systems. In: IEEE Trans. Comput., vol.36, no.4, pp.421-432.

- Darlington J. and Reeve M. (1981).
  Alice: A multi-processor reduction machine for the parallel evaluation of applicative languages. In: Proc. ACM Conference for Functional Programming Languages and Computer Architecture, pp.65-75.

- Day J.E. and Hottenstein M.P. (1971).
  Review of sequencing research. In: Naval Research Logistics Quarterly, vol.18, pp.11- 39.

- Dennis J.B. (1980).
  Data-flow supercomputers. In: IEEE Computer, vol.13, no.11, pp.48-56.

References

- Dennis J.B. and Misunas D.P. (1975).
  A preliminary architecture for a basic dataflow processor. In: Proc. 2nd IEEE Symposium on Computer Architecture, p.126.

- Dijkstra E.W. (1959).
  A note on two problems in connection with graphs. In: Numer. Math., vol.1, pp.269-271.

- Dijkstra E.W. (1968).
  Co-operating sequential processes. In: Programming languages. Genuys F. (ed.), (New York: Academic Press).

- Elmaghraby S.E. (1968).
  The machine sequencing problem - review and extensions. In: Naval Research Logistics Quarterly, vol.15, no.2, pp.205-232.

- Fernandez E.B. and Bussell B. (1973).
  Bounds on the number of processors and time for multi-processor optimal schedules. In: IEEE Trans. Comput., vol.22, no.8, pp.745-751.

- Flynn M.J. (1972).
  Some computer organisations and their effectiveness. In: IEEE Trans. Computers, vol.21, no.9, pp.948-960.

- Gajski D.D. and Peir J. (1985).
  Essential issues in multi-processor systems. In: IEEE Computer, vol.18, no.6, pp.9-27.

- Garey M.R. and Johnson D.S. (1979).
  In: Computers and intractability: A guide to the theory of NP-completeness. (San Fransisco: W.H.Freeman).

- Gaudiot J.L. (1987).
  Data-driven multi-computers in digital signal processing. In: Proceedings of the IEEE, vol.75, no.9, pp.1220-1234.

- Gaudiot J.L., Dubois M., Lee L.T. and Tohme N. (1986).
  The TX16: A highly programmable multi-processor architecture. In: IEEE Micro, vol.6, no.10, pp.18-31.

- Glaser E.L., Pyle I.C. and Illingworth V. (1986).
  In: Dictionary of computing (second edition). (New York: Oxford University Press).

References

- Goddard A.J. (1987).
  In: DFDL; definition, specification and examples. CIE internal report AJG/DFDL/1, City University, London, August 1987.

- Goddard A.J. (1989).
  In: DFDL language definition. CIE internal report AJG/DFDL/2, City University, London, February 1989.

- Goddard A.J. and Lawson S.S. (1988a).
  Mapping signal processing algorithms onto a multi-processor network. In: Signal Processing IV: Theory and Applications, pp.1229-1232, (Amsterdam: North-Holland).

- Goddard A.J. and Lawson S.S. (1988b).
  An automated approach to mapping DSP algorithms onto Transputer arrays. In: IEE Symposium on Digital Signal Processing for VLSI, Savoy Place, London. (Digest No: 1988/137).

- Gordon M.J., Milner A.J. and Wandsworth C.P. (1979).
  Edinburgh LCF. In: Lecture notes in Computer Science (Springer-Verlag).

- Goyal D.K. (1976).
  Scheduling processor bound systems. In: Report no. CS-76-036, Computer Science Dept., Washington State Univ., Pullman, WA.

- Graham R.L. (1972).
  Bounds on multi-processing anomalies and related packing algorithms. In: Spring Joint Computer Conference, 1972, pp.205-217.

- Guernic P., Benveniste A., Bournai P. and Gautier T. (1986).
  SIGNAL - A dataflow oriented language for signal processing. In: IEEE Trans. Acoustics, Speech and Signal Processing, vol.34, no.2, pp.362-374.

- Gurd J. and Watson I. (1980).
  Data driven system for high speed parallel computing. In: Computer design vol.9, no.6, p.91 and no.7, p.97.

- Harary F. (1969).
  In: Graph theory. (Reading, Mass: Addison-Wesley).

- Hartimo I., Kronlof K., Simula O. and Skytta J. (1986).
  DFSP: A data flow signal processor. In: IEEE Trans. Comput., vol.35, no.1, pp.23-33.

References

- Hetch M.S. (1977).
  In: Flow analysis of computer programs. (New York: North Holland).

- Hillis W.D. (1985).
  In: The Connection Machine. (Cambridge, Mass: MIT Press).

- Hoare C.A.R. (1974).
  Monitors: An operating system structuring concept. In: Commun. ACM, vol.17, no.10, pp.549-557.

- Hoare C.A.R. (1978).
  Communicating sequential processes. In: Commun. ACM, vol.21, no.8, pp.666-677.

- Hockney R.W. (1983).
  Characterisation of parallel computers. In: Parallel and large- scale computers: performance, architecture, applications. ed. M.Ruschitzka. (Oxford: North- Holland) pp.201-206.

- Hockney R.W. and Jesshope C.R. (1981).
  In: Parallel computers: architecture, programming and algorithms. (Bristol, England: Adam Hilgar).

- Hu T.C. (1961).
  Parallel sequencing and assembly line problems. In: Operations Research, vol.9, no.6, pp.841-848.

- Hwang K. and Briggs F.A. (1984).
  In: Computer architecture and parallel processing. (New York: M^{c}Graw-Hill).

- Hyvarinen O., Hartimo I. and Simula O. (1987).
  Methods to improve the computing efficiency of an arbitrary digital signal processing flow graph with implementation constraints. In: Proc. IEEE International Symposium on Circuits and Systems ISCAS-87, Philadelphia, pp.915-918.

- IBM (1988).
  In: Parallel Fortran language and library reference. International Business Machines, Pub. No. SC23-0431-0.

- IEEE (1981).
  Interconnection networks. In: IEEE Computer, vol.14, no.12.

References

- IEEE (1982).
  Special issue on dataflow systems. In: IEEE Computer, vol.15, no.2.

- IEEE (1987).
  Interconnection networks. In: IEEE Computer, vol.20, no.6.

- INMOS (1986).
  In: Transputer reference manual. (Bristol: INMOS Ltd.).

- Jesshope C. (1987).
  In: Major advances in parallel processing. (Aldershot: Gower Technical Press).

- Karp R.M. and Miller R.E. (1966).
  Properties of a model for parallel computations: determinacy, termination, queuing. In: SIAM Journal of Applied Mathematics, vol.14, no.6, pp.1390-1411.

- Karp R.M. and Miller R.E. (1969).
  Parallel program Schemata. In: Journal of Computer and System Sciences, vol.3, pp.147-195.

- Karp R.M. (1986).
  Combinatorics, complexity and randomness. In: Commun. ACM, vol.29, no.2, pp.98-111.

- Knuth D.E. (1972).
  In: The art of computer programming: vol.3, sorting and searching. (Reading, Mass: Addison-Wesley).

- Kronolf K., Hartimo I. and Simila O. (1983).
  The compatibility of computing algorithms to parallel processing architectures. In: Proc. IEEE International Symposium on Circuits and Systems ISCAS-83, Newport Beach, California, pp.48-51.

- Kruatrachue B. and Lewis T. (1988).
  Grain size determination for parallel processing. In: IEEE Software, no.1, pp.23-32.

- Kuck D.J. and Stokes R.A. (1982).
  The Burrough's Scientific Processor. In: IEEE Trans. Computers, vol.31, no.5, pp.363-376.

- Kung H.T. (1982).
  Why Systolic architectures? In: IEEE Computer, vol.15, pp. 37-46.

References

- Kung S.Y. (1984).
  On supercomputing with systolic/wavefront array processors. In: Proc. IEEE, vol.72, no.7, pp. 867-884.

- Kung S.Y., Arun K.S., Gal-Ezer R.J. and Bhaskar Rao D.V. (1982).
  Wavefront array processor: Language, architecture and applications. In: IEEE Trans. Comput., vol.31, no.11, pp.1054-1066.

- Ledgard H. (1981).
  In: ADA - An introduction and Ada reference manual. (Springer-Verlag).

- Lee S.Y. and Aggarwal J.K. (1987).
  A mapping strategy for parallel processing. In: IEEE Trans. on Comput., vol.36, no.4, pp.433-441.

- Lee E.A. and Messerschmitt D.G. (1987).
  Synchronous data flow. In: Proc. IEEE, vol.75, no.9, pp.1235-1245.

- Lenstra J.K. and Rinnooy Kan A.H.G. (1978).
  Complexity of scheduling under precedence constraints. In: Operations Research, vol.26, pp.22-35.

- Loeffler C., Lightenberg A. and Moschytz (1988).
  Algorithm-architecture mapping for custom DSP chips. In: Proc. IEEE International Symposium on Circuits and Systems ISCAS-88, Helsinki, Finland.

- Mason S.J. (1953).
  Feedback theory-Some properties of signal flow graphs. In: Proc. IRE, vol.41, pp. 920-926.

- May D. (1983).
  Occam. In: SIGPLAN notices, vol.18, no.4, pp.69-79.

- May D. (1987).
  In: Occam2 language definition. (Bristol: INMOS Ltd.).

- M^{c}Carthy J. (1960).
  Recursive functions of symbolic expressions and their computation by machine. In: Commun. ACM, vol.3, no.4, pp.185-195.

- M^{c}Entire P.L., O'Reily J.G. and Larson R.E. (1984).
  In: Distributed computing: concepts and implementations. (New York: IEEE press).

References

- M^cGraw J., Skedzielewski S., Allan S., Grit D., Oldehoeft R., Glauert J.R.W., Dobes I. and Hohensee P. (1983).
  In: SISAL-Streams and iterations in a single-assignment language. Language reference manual version 1.0., Lawrence Livermore National Laboratory.

- M^cNaughton R. (1959).
  Scheduling with deadlines and loss functions. In: Management Science, vol.6, no.1, pp.1-12.

- Mead C. and Conway L. (1980).
  In: Introduction to VLSI systems. (Reading, Mass: Addison-Wesley).

- Millstein R. (1973).
  Control structures in Illiac IV Fortran. In: Commun. ACM, vol.16, pp.622-627.

- Minieka E. (1978).
  In: Optimisation algorithms for networks and graphs. (New York: Marcel Dekker).

- Mitchell J.G., Maybury W. and Sweet R. (1979).
  In: Mesa language manual, version 5.0. CSL-79-3, Palo Alto Research Centre, Xerox.

- Moder J.J. and Phillips C.R. (1964).
  In: Project management with CPM and PERT. (New York: Reinhold).

- More W., M^cCabe A. and Urquhart R. (1987).
  In: Systolic Arrays. (Bristol: Adam Hilgar).

- Mundie D.A. and Fisher D.A. (1986).
  Parallel processing in Ada. In: IEEE Computer, vol.19, no.8, pp.20-25.

- Nilsson N. (1971).
  In: Problem solving methods in artificial intelligence. (New York: M^c-Graw-Hill).

- Nilsson N. (1980).
  In: Principles of artificial intelligence. (Palo Alto, Calif: Tioga).

- Padua D.A. and Wolfe M.J. (1986).
  Advanced compiler optimisations for supercomputers. In: Commun. ACM, vol.29, no.12, pp.1184-1201.

References

- Papadimitriou C.H. and Yannakakis M. (1978).
  Scheduling interval-ordered tasks. In: Report no. TR-11-78, Centre for Research in Computer Technology, Harvard Univ., Cambridge, Mass.

- Pearl J. (1984).
  In: Heuristics: Intelligent search strategies for computer problem solving. (Reading, Mass: Addison-Wesley).

- Polychronopoulos C.D. and Banerjee U. (1987).
  Processor allocation for horizontal and vertical parallelism and related speedup bounds. In: IEEE Trans. Comput., vol.36, no.4, pp.410-420.

- Price C.C. and Pooch U.W. (1982).
  Search techniques for a non-linear multi-processor scheduling problem. In: Naval Research Logistics Quarterly, vol.29, no.2, pp.213-233.

- Reddaway S.F. (1973).
  DAP-A distributed array processor. In: Proc. 1st ACM Symposium on Computer Architecture.

- Reed D.A and Grunwald D.C. (1987).
  The performance of multi-computer interconnection networks. In: IEEE Computer, vol.20, no.6, pp.63-73.

- Richmond G. (1982).
  A dataflow implementation of SASL. In: M.Sc. Thesis, Department of Computer Science, University of Manchester.

- Sarkar V. (1989).
  In: Partitioning and scheduling parallel programs for execution on multi-processors. (London: Pitman).

- Skillicorn D.B. (1988).
  A taxonomy for computer architectures. In: IEEE Computer, vol.21, no.11, pp.46-57.

- Steinmetz R., Gemballa R., Lenzer J. and Roth H. (1983).
  Realisation of digital filter algorithms by the use of a high speed parallel processing architecture. In: Proc. IEEE International Conference on Acoustics, Speech and Signal Processing, pp.1188-1191.

- Stevens K. (1975).
  CFD-A Fortran like language for the ILLIAC IV. In: SIGPLAN notices, vol.10, pp.72-80.

References

- Su W., Faucette R. and Seitz C. (1985).
  In: C programmer's guide to the Cosmic Cube. Technical report 5203:TR:85, Computer Science Department, California Institute of Technology.

- Thaler M., Loeffler C. and Moschytz (1987).
  Programming, analysis and synthesis of parallel signal processors. In: Proc. IEEE International Symposium on Circuits and Systems ISCAS-87, Philadelphia, pp.358-361.

- Turing A. (1936).
  On computable numbers, with an application to the Entscheidungsproblem. In: Proc. London Math. Soc., Ser 2, no.42, pp.230-265, and no.43, pp.544-546.

- Turner D.A. (1976).
  In: SASL language manual. Computer Laboratory, University of Kent.

- Ullman J.D. (1975).
  NP-complete scheduling problems. In: Journal of Computer System Science, vol.10, pp.384-393.

- Watson I. (1988).
  Flagship: A parallel architecture for declarative programming. In: Proceedings of the 15th Annual International Symposium on Computer Architecture, May 1988, pp.124-130.

- Writh N. (1977).
  Modula: A language for modular multiprogramming. In: Software Practice & Experience, vol.7, no.1, p.3.

- Writh N. (1978).
  In: Modula-2. Nr.36, Institut fur Informatik, ETH.

- Young S.J. (1982).
  In: Real time languages: Design and development. (Ellis Horwood).

- Zakharov V. (1984).
  Parallelism and array processing. In: IEEE Trans. Comput., vol.33, no.1, pp.45-78.