JOINT ROYAL MELBOURNE INSTITUTE OF TECHNOLOGY AND COMMONWEALTH SCIENTIFIC AND INDUSTRIAL RESEARCH ORGANISATION PARALLEL SYSTEMS ARCHITECTURE PROJECT

Implementation and Analysis of a Hybrid Dataflow System

TR 118 094 R

Mark W. Rawling †

December 1989

† CSIRO Division of Information Technology
c/o Department of Communication and Electrical Engineering
Royal Melbourne Institute of Technology
124 La Trobe St., Melbourne, 3000

This technical report is a reproduction of the author's thesis submitted in March 1988 to the Royal Melbourne Institute of Technology in partial fulfillment of the requirements for the degree of Master of Engineering in the Faculty of Engineering.

ABSTRACT

A new, high performance dataflow computer architecture is being developed in the Department of Communication and Electronic Engineering, Royal Melbourne Institute of Technology (RMIT), Victoria, Australia. This thesis describes the emulation and simulation of a unique 'hybrid' dataflow system that will form the basis of this new architecture.

A dataflow processing element emulator is described, along with modifications to the RMIT dataflow model to suit its implementation on this machine. Particular advantages and disadvantages of the unique architecture of the emulator are discussed.

An implementation and critical analysis of the hybrid dataflow model is also presented. Of special interest is the dataflow language DL1, which has been greatly enhanced. Problems with the language syntax and code generation templates of the old compiler, which reflect deficiencies in the original dataflow model used at RMIT, are discussed and corrected. Where appropriate, changes have been made to the dataflow model itself, especially in the areas of the node set and matching unit operation. The benefits of these changes are analysed by simulation and comparisons are made with approaches taken by other dataflow researchers.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Gregory Egan for his guidance throughout the course of this research.

I would also like to thank Chris Baharis for discussions on dataflow principles and compiler testing, and Alan Young and Dr. David Abramson for their contributions to the project.

Lastly, thanks to the Commonwealth Department of Education which provided financial support in the form of a Commonwealth Postgraduate Research Award.

DECLARATION

No portion of the work referred to in this thesis has been submitted to this or any other university, college of advanced education, or other institute of learning in support of any other degree or qualification.

CONTENTS

Chapter 1 **Introduction** 1

1.1 Parallel Computing 1
1.2 The Dataflow Approach 1
1.3 Dataflow Machines 2
1.4 Dataflow Programming Languages 2
1.5 Research Aim 4
1.6 Thesis Outline 4

Chapter 2 **Dataflow at RMIT** 5

2.1 A Brief History 5
2.2 Static and Dynamic Architectures 7
2.3 The RMIT Hybrid Architecture 7
2.4 Characteristics of the RMIT Dataflow System 8
2.5 Execution of Dataflow Graphs 9
2.6 The Prototype Emulator 9
2.7 Token Format 10
2.8 Node Format 10
2.9 The Simulation Environment, DFSIM 10
2.9.1 Simulation Details 11
2.9.2 An Example 12
2.9.3 Explanation of Simulation Results 15

Chapter 3 **Matching Functions** 17

3.1 Hybrid Matching Functions 17
3.2 Monadic 17
3.3 Diadic 18
3.4 Storage 18
3.5 First 19
3.6 Prime 19
3.7 Protect 20
3.8 Complex Protect 21
3.9 Stream 21
3.10 Head and Tail 23
3.11 Cons 24

Chapter 4 **Dataflow Languages and Code Generation** 27

4.1 Introduction 27
4.2 Properties of Dataflow Languages 27
4.2.1 Implicit Control 27
4.2.2 Single Assignment 27
4.2.3 Functionality 28
4.2.4 Non-strict Evaluation 28
4.2.5 Well Formed and Clean Graphs 28
4.2.6 Reentrancy 28
4.2.7 Graphical Languages 28
4.3 Dataflow Language One (DL1) 29
4.3.1 Expressions and Expression Lists 29
4.3.2 Determinacy 30
4.3.3 Conditional Expressions 31
4.3.4 Eager and Lazy Evaluation 35
4.3.4.1 Eager Evaluation in Conditionals 35
4.3.4.2 Lazy Evaluation in Conditionals 36
4.3.4.3 Eager, Lazy and Hybrid Code Templates 37
4.3.4.4 Eager, Lazy and Hybrid Merge Expansions 39

4.3.5 Subgraphs 41
4.3.5.1 The Copy Number Mechanism 42
4.3.5.2 The Unique Colour Mechanism 45
4.3.6 Iteration and Tail Recursion 47
4.3.6.1 Static Iteration 47
4.3.6.2 Dynamic Tail Recursion and Loop Unfolding 49
4.3.6.3 Tail Optimisation 50
4.3.7 Dynamic Graph Unravelling 51
4.3.8 Sequences and Streams 53
4.3.8.1 Open Sequences (Queues) 55
4.3.8.2 Streams 55
4.3.8.3 Stream Functions 56
4.3.8.4 Code Templates for Stream Functions 57

Chapter 5 Simulation, Results and Analysis 61

5.1 Introduction 61
5.2 Test Programs 61
5.3 Duplicate vs Replicate 61
5.4 Influence of Node Set and Matching Functions 63
5.4.1 Effect on Graph Size 69
5.4.2 Effect on Concurrency 71
5.4.3 Effect on Execution Times 71
5.5 Unravelling 72
5.6 Overheads of Tagging 78
5.7 Machine Utilisation 80
5.8 Distribution of Machine Activity 81
5.9 Other Effects 82
5.9.1 Effect of Sequential Code Segments 82
5.9.2 Operations on Open Sequences 83
5.9.3 Unravelling of Queues 83

Chapter 6 Summary and Conclusions 87

6.1 Simulation and the Design Process 87
6.2 Recommended Modifications to DL1 87
6.3 Suggestions for Further Research 88
6.4 Conclusions 89

Appendix A Test Programs 91

A.1 DL1 Source Listings 91

Appendix B DL1 Reference 99

B.1 Introduction 99
B.2 Using DL1 99
B.3 Syntax Diagrams 99
B.4 Compiler Options 107
B.5 Reserved Words 107
B.6 Predefined Functions 107

Appendix C Advanced Node Set Definition 109

C.1 Introduction 109
C.2 Multiple Output Destinations 109
C.3 Transparent Transmission of End of Stream Tokens 109
C.4 Exceptions 109
C.5 Node Definitions 109
C.5.1 Arithmetic 110
C.5.2 Logical and Set 111
C.5.3 Relational 111
C.5.4 Sequence 111
C.5.5 Stream 111
C.5.6 Type Coercion 112

C.5.7 Storage 112
C.5.8 Replicate and Identity 112
C.5.9 Path Control 112
C.5.10 Token Structure 113
C.5.11 Priming 113
C.5.12 Destination 114
C.5.13 Colour 114
C.5.14 Shared Subgraphs (1) 114
C.5.15 Shared Subgraphs (2) 115
C.5.16 Predefined Nodes 115
C.5.17 System Nodes 115
C.5.18 Input and Output 116

Appendix D Emulation Details 117

D.1 Emulator Hardware 117
D.2 Token Format 117
D.3 Data Types 118
D.4 Exceptions 118
D.5 Nodes 119
D.6 ITL Format 119

References 121

Chapter 1
INTRODUCTION

1.1 Parallel Computing

Advances in electronic technology have traditionally been relied upon to supply the ever increasing computing power required by today's applications. However, relatively conservative commercial equipment manufacturers have a need to produce products that will be readily accepted in a highly competitive market place. It has thus been left to research oriented bodies such as universities and some élite manufacturers to provide the facilities and manpower for research into new, perhaps novel ways of improving performance.

Such a research program is under way at the Royal Melbourne Institute of Technology (RMIT), Victoria, Australia. Parallel computer architectures are being investigated as a means of improving performance over the more conventional von Neumann uniprocessors.

State of the art micros, minis and super computers have reached a sophisticated level of optimisation in an attempt to overcome the constraints of sequential, fetch and execute type of operation. However, while they may remain purely sequential from a programmer's point of view, much of their high performance is due to various forms of parallel processing, e.g. instruction pipelining, multiple logic units (for concurrent address/data calculations, etc.) and vector processing.

An alternative to optimising the performance of von Neumann machines is to design new architectures with fewer operational constraints. The optimisations mentioned above have been successfully applied at a higher (inter-processor) level, giving rise to parallel computers which can execute many statements concurrently and operate on more than one set of data at a time. These machines can be programmed to take advantage of the concurrency in an algorithm, giving the programmer much greater scope for algorithmic optimisation.

Parallel machines have their drawbacks however. More complicated, explicit control over the run time environment is often required in both the expression and execution of an algorithm, making these machines more difficult to program than their sequential counterparts. Even though much excellent work has been done in the design of special compilers and translators to adapt conventional languages for execution on parallel machines, there are usually specific hardware constraints that can not be fully overcome, e.g. fixed vector sizes, non-uniform machine architecture, etc.. Also, conventional programming languages can hinder the expression of concurrency due to their reliance on sequential control constructs. The result is that the user can become just as involved in designing an algorithm to suit the computer, as in designing one that suits the problem. New computer architectures and programming languages should therefore not only perform adequately in terms of raw processing speed, but must also take into account the factors of hardware control complexity and ease of software development.

1.2 The Dataflow Approach

The dataflow approach aims to provide a simple and elegant solution to the problems of control and program development in a multiprocessor environment. This is achieved by avoiding most of the problem areas of conventional multiprocessors such as shared memory conflicts (multiple assignment), side effect based computation, etc.. Much of the burden of program development for efficient multiprocessor based execution is removed by changing the emphasis from explicit to implicit specification of parallelism and communication [10].

In the dataflow model, an algorithm is expressed through the use of specially designed high level, textual dataflow languages. It is the task of the compiler to convert this program into a graphical, intermediate form known as the *program graph*, the program graph bears a strong relationship to the compiler's parse tree. The compiler then expands the program graph, through the use of code generation templates, into a low level executable form called the *machine graph*. This is a finite directed graph of low level operators (*nodes*) which communicate by passing results (*tokens*) between them, along both static and dynamic data paths (*arcs*).

At run time, a node in the machine graph becomes executable when a valid firing condition has been established by tokens on its input arcs. The dataflow model places no inherent restrictions on the firing rules allowable (the *matching functions*), nor on the granularity of the node set itself, but most systems use nodes with at most two input arcs for hardware simplicity, and of generally comparable power to the machine instructions of conventional microprocessors. It is common to have a literal data token associated with a node description, so that three inputs are possible if one of them is a constant; this is true of the RMIT system.

The node set is made up of mostly functional operators which have a constant, single valued mapping of input data onto output data. Machine graphs built from these nodes are also functional, but have the added properties of non-strictness (so that not all inputs must be present before execution can begin) and inherent concurrency (so that many nodes can execute in parallel, spread out over the dataflow machine).

Despite its inherently decentralised, determinate nature, dataflow does not reject the concepts of random access/shared memory, nondeterminism, etc.. In fact much of the current research into dataflow architectures is directed towards the inclusion of these very features. Special memory units, variously called *I-Structures*, *structure-stores*, etc., provide for efficient, random memory access; while the asynchronous nature of dataflow architectures make them suitable for the execution of nondeterministic programs. The RMIT dataflow system, in common with most others, features a nondeterministic *merge* operation which takes advantage of the asynchronous dataflow communications channels. Other nondeterministic operators are present in the RMIT node set, but as this thesis is mainly concerned with determinate programming, these operators will be used in special combinations to build graphs that are themselves determinate.

1.3 Dataflow Machines

Dataflow machines are MIMD (multiple instruction stream, multiple data set) processors, designed to provide an efficient architecture for the parallel execution of dataflow machine graphs. A typical machine architecture is shown in figure 1.1.

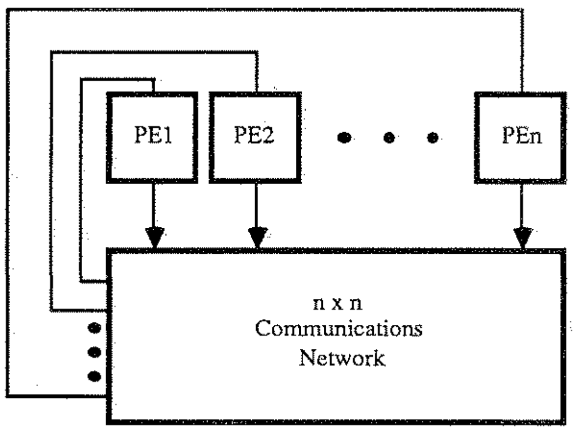


FIGURE 1.1 A basic dataflow machine architecture

This particular arrangement forms the basis of most of the dataflow machines designed to date. Extensions to this basic architecture include *structure stores* for holding complex data items [11, 43], *fault tolerance* [46, 30], etc.. However, due to a current lack of suitable operating systems, most dataflow machines don't operate in a stand alone manner, and in the interim, a practical dataflow machine will almost certainly have an interface to a conventional host computer which provides a basis for program development, file operations, etc.. In figure 1.1, the host interface would be one of the PEs or at least be attached to one of the PEs, together with i/o streams and other external interfaces.

1.4 Dataflow Programming Languages

Dataflow languages can be similar to conventional imperative/procedural ones (e.g. Whitelock's P5, which is a dataflow subset of PASCAL [51]), but more usually combine the features of *functional* and *single assignment* languages, to provide a powerful and efficient interface between high level algorithms and the low level node, arc and token, graphical format.

```
subgraph Quadratic(a, b, c: real) -> (root1, root2: real);
        begin
                b^2 - 4*a*c -> discrim;
                (-b + sqrt(discrim)) / (2*a) -> root1;
                (-b - sqrt(discrim)) / (2*a) -> root2;
        end (* Quadratic *);
```

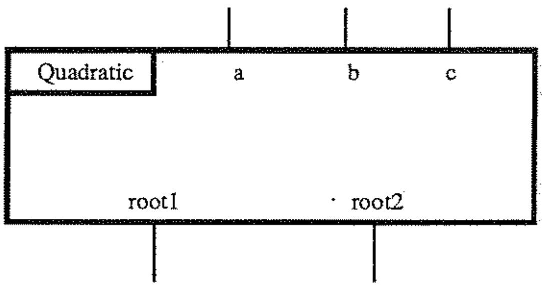


FIGURE 1.2 A dataflow function definition

Figure 1.2 shows a simple function as it might be coded in a typical dataflow programming language (in this case DL1), together with a simple graphical representation which is used throughout this thesis. Note that no explicit statements of control or connectivity exist in this code, apart from those implied by pure data dependencies. In particular, sequential execution of statements is not assumed; a function can execute at any time that all (or possibly just some) of its arguments are present. *Priming tokens* initiate the execution, while result tokens, together with any other side effects, are the results of the execution.

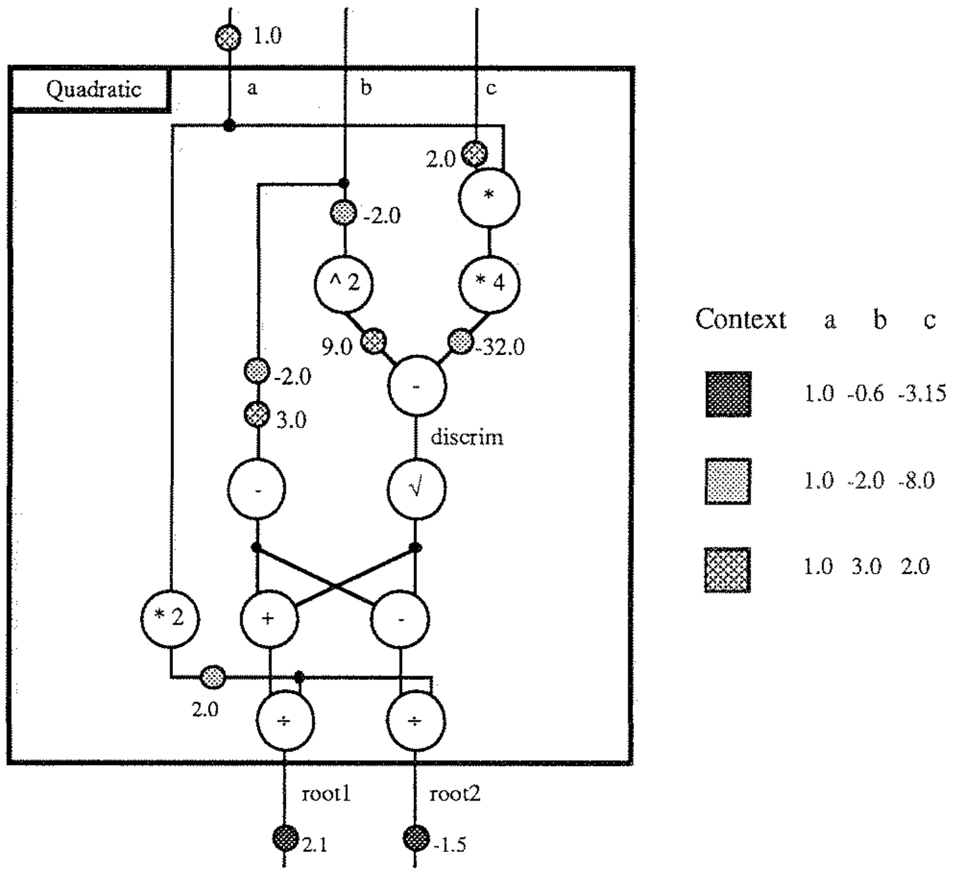


FIGURE 1.3 A snapshot of the function 'Quadratic'

Figure 1.3 shows a *snapshot* of the execution of the function `Quadratic`. Such a snapshot shows the current state of a graph's execution and includes all of the intermediate data tokens together with their static positions in the graph. In the case of dynamic architectures, the tokens' *colours*, which separate tokens from different contexts sharing the same code, are also shown. In this snapshot, `Quadratic` has been called from three separate contexts as indicated by tokens of three different colours. Note the presence of literal data on some of the nodes in this graph. An optimisation is shown in this graph in that common subexpressions of `-b`, `sqrt(discrim)` and `2*a` have been found. In fact, many of the optimisation techniques used in the compilation of conventional programming languages can be applied to dataflow languages as well [48, 33, 34].

A program written in a dataflow language will often be more 'readable' than one written in a conventional language like Pascal. Coding techniques which take advantage of the complex execution characteristics of uniprocessors (e.g. reusing variables to save space, because sequential execution guarantees unambiguous multiple assignment) are avoided. While some of these techniques may be space/time efficient, or textually concise, the added intricacy of the resulting program often makes mathematical analysis, code readability, etc., difficult [4, 10].

Some of the dataflow languages designed to date are: TDFL (Textual Data Flow Language), developed by K.S. Weng in his work on streams and static dataflow machines [50]; DL1 (Dataflow Language 1), designed by C.P. Richardson at Manchester University and now used extensively at RMIT and throughout this thesis [42]; ID/ID-NOUVEAU (Irvine Dataflow), used by Professor Arvind's dataflow group at MIT [8, 48, 37]; and SISAL (Streams and Iteration in a Single Assignment Language), a collaborative venture of the Colorado State University, Digital Equipment Corporation, Lawrence Livermore National Laboratory and Manchester University [35]. Of course there are numerous other dataflow languages in use and many traditional languages are suitable for data driven execution, e.g., 'functional' languages like pure LISP, and parallel logic programming languages like GHC (Guarded Horn Clauses), Concurrent Prolog, Oc, etc. [47].

1.5 Research Aim

A collaborative research program between RMIT and the Commonwealth Scientific and Industrial Research Organisation (CSIRO) Department of Information Technology (DIT), is investigating the issues briefly raised above with regards to the RMIT dataflow architecture. This thesis contributes to this research by describing the work done by the author in implementing a prototype emulation facility for the RMIT architecture and in designing an enhanced version of the DL1 dataflow compiler originally written by C.P. Richardson as part of his Ph.D. research at the University of Manchester [42]. Both Richardson's work and the RMIT dataflow architecture are based on Egan's 'FLO' dataflow model [24].

1.6 Thesis Outline

Chapter 2 describes the dataflow system in use at RMIT and the author's involvement in the project. The prototype emulator is described along with modifications to the FLO dataflow system to adapt it for use on this emulator.

Chapter 3 describes the matching functions introduced by the author to overcome many of the limitations of the current system and to help investigate the role of the matching unit in efficient graph execution.

Chapter 4 describes the program development environment at RMIT, in particular the language DL1 is critically examined and enhanced.

Chapter 5 presents the results of simulation studies which outline the features of, and show the effects of the changes made to, the RMIT dataflow system. The significant features of these results are highlighted and discussed in this chapter.

Chapter 6 presents conclusions and some possible areas for future research.

Chapter 2
DATAFLOW AT RMIT

2.1 A Brief History

Software:-

Dataflow research began at RMIT in 1981 when the original simulator, translator, compiler, etc., for the FLO dataflow system [23] were installed on the RMIT computer centre CDC Cyber 180-835. In following years, the software was ported to Unix based engineering work stations in the Department of Communication and Electronic Engineering. Some improvements, especially to the simulator were also made at this stage.

Having completed hardware testing and the assembly code for the prototype emulator board, the author began working with the DL1 compiler to write test graphs. A significant rewrite of much of the compiler was undertaken, including new code templates, improved syntax, new types, improved type checking, etc.. New features have been added to other system software as needed, e.g., the matching functions described in chapter 3 have been added to the simulator, and new node and binary formats have been added to the compiler, simulator DFSIM, and intermediate textual language (ITL) to machine binary translator FLOITL. Also, a disassembler to convert from binary back to ITL was written.

An important development was the author's design and implementation of an interactive control and debugging interface for the prototype processing element. This interface allows graphs to be loaded, run, debugged, etc.. Also, it provides a debugging package for the emulator code itself, which has proven invaluable in the commissioning of the prototype hardware.

The latest development in the project has been the involvement of the CSIRO Division of Information Technology (DIT). With DIT's support, the software side of the project will see the development of an ID-Nouveau compiler [37], an IF1 to RMIT ITL translator (for SISAL and related compilers [35, 45]), and a GHC compiler for Ueda's 'Guarded Horn Clauses' logic programming language [49]. This is part of an effort to concentrate on applications programming and analysis of dataflow machines. Several CSIRO divisions and other interested bodies (e.g. Melbourne University's and RMIT's computer science departments) are already developing large applications systems based on these languages, which include:

- Using simulated annealing algorithms for optimal building layout
- Robot trajectory planning algorithms
- Timetable computation algorithms
- Some experimental expert systems
- High speed digital logic simulation
- Real time computer generated imagery

Hardware:-

Hardware development has been undertaken at RMIT since 1982. In the original documentation [24], was a proposal for a fast dataflow processing element (PE) design for the FLO system; a very similar layout to that used in other dataflow architectures [6, 29, 14, etc.]. It is based on a pipelined ring structure, with a communications network interface through which PEs transmit and receive tokens to/from each other. A local queue is included in an attempt to optimise performance when the output of one node is directed to the input of another node in the same element, but if the network has a path from the output of a PE back to the input of that PE, then that path is made redundant by the local queue. In this case, the local queue itself might be unnecessary and whether to include one or not will depend largely on the allocation strategy used to distribute the nodes across the dataflow machine.

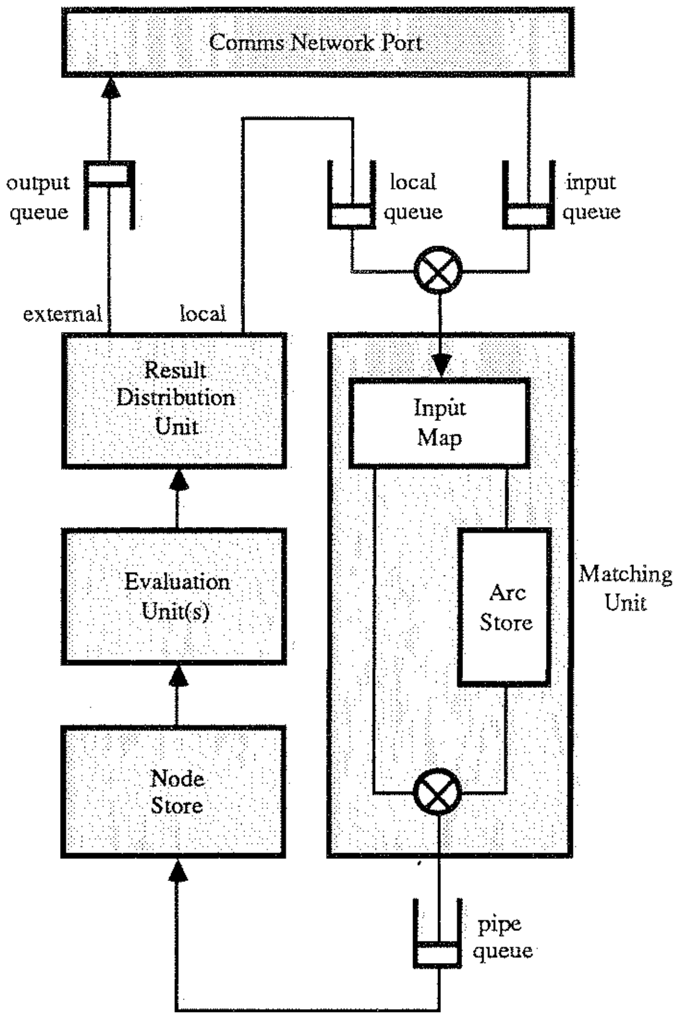


FIGURE 2.1 Fast dataflow processing element layout

Figure 2.1 shows a revised layout of the fast processing element design, which features a separate unit for result distribution (the *result distribution unit*) and an extra buffer (the *pipe queue*) at the output of the *matching unit*. The pipe queue is included to help separate the tasks performed by the CPU modules in the PE emulator (§2.6 and [40]), but its value has also been demonstrated for discrete, high speed PE designs [29].

The design of a single board prototype emulator, based on this PE structure, and using dual MC68000s with interconnecting FIFOs and an external host interface, was completed by the author and another fourth year B.Eng. student in 1982 (appendix D and [40]). Test graphs, including a 3000 node simulator for a laser range finder based object recognition system, [41], were running on the emulator by 1984.

More recently, a re-engineering of the prototype board has adapted it for use in a sixteen element multiprocessor emulator [52]. This emulator features a sixteen way buffered delta network [22] for token transmission between PEs and has the capability of using either one or two single 68000/68020 processor boards per PE slot (§2.6). It has been running test graphs since autumn 1987, controlled by a Unix based host computer interfaced to one of the PE slots, but the work described in this thesis with regards to PE design and programming refers to the author's original dual 68000 board.

The CSIRO project involvement will have a large impact on hardware development, since it is proposed to commission a very high performance, many PE (≥32) hybrid dataflow machine (§2.3). A 'Sun 3/260' Unix work station has been installed as the host for the new dataflow machine and software development. Currently, a high speed interface between the Sun and the 16 PE emulator is nearing completion. The new dataflow machine will include a floating point node set with high speed hardware support; a hybrid architecture, combining static/queued and dynamic computation models (see below); and distributed *object storage* for efficient random access of stored data structures [1, 3].

2.2 Static and Dynamic Architectures

There are currently two main classifications for dataflow architectures, *static* and *dynamic* . The static scheme was first proposed by Dennis [19, 20, 21], and has been used by various research architectures such as Texas Instruments' DDP [15] and the French LAU system [38]. The dynamic scheme is used by Arvind's research group at MIT [6]; by Gurd *et al* at Manchester University [29]; by Davis' DDM architecture at the University of Utah [17]; and by the Japanese SIGMA-1 system [44, 53].

In the static model only one token is allowed on an arc at a time, whereas in the dynamic model, many tokens are allowed on an arc, their order or context being determined by special *tag* or *colour* fields.

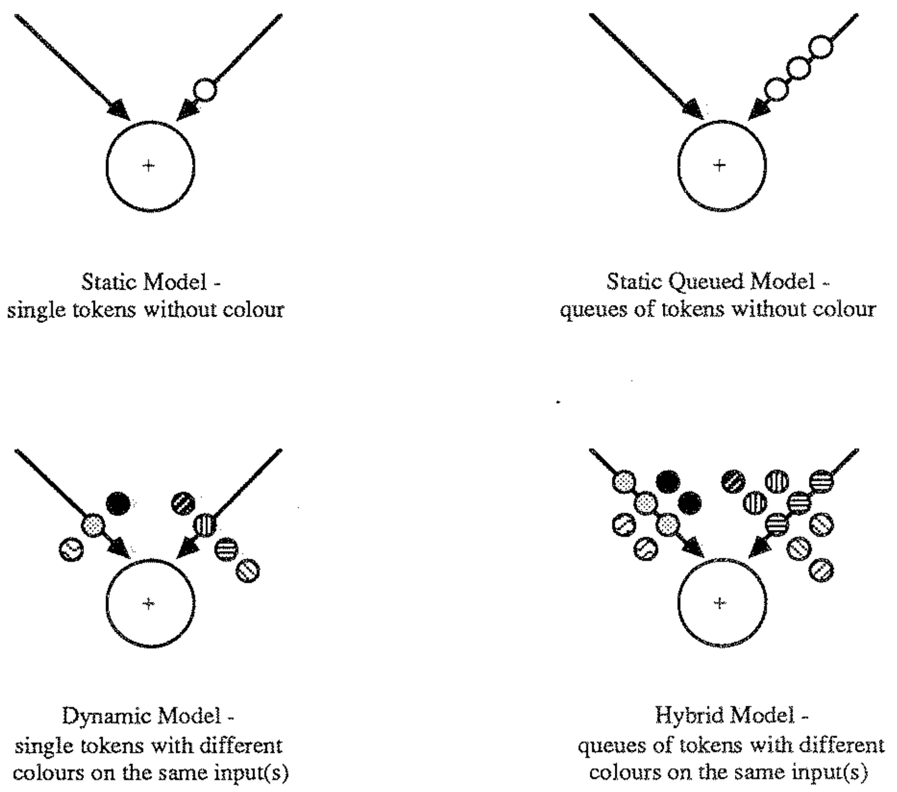
Static architectures are easier to implement than dynamic ones because the matching of operands can be performed with a simple table lookup. A two input node has at most one token waiting on one of its input arcs. When a second token arrives, on the opposite arc, the node *fires* and the tokens are consumed. The main disadvantage of the static model is that the concurrency of a graph is limited by its static width and the amount of data pipelining that can be achieved. The static width of a graph can be a particularly poor measure of a graph's potential concurrency as it hides the effects of loop unfolding and other sources of dynamic parallelism, e.g. recursion. Pipelining can be restricted by static architectures which do not allow queueing on arcs, since dummy nodes are required to equalise the lengths of the data paths to prevent deadlocking [36]. Never the less, static architectures can provide very good performance for 'wide' algorithms which process large sets or streams of data in a filter like fashion.

In a dynamic architecture many tokens with different tags, or colours, may be pending on both inputs of a node, and they are only consumed when tokens with matching tags arrive at the opposite input. The matching process is more complex than for a static architecture and can be further complicated by more complex matching functions such as those discussed in chapter 3 and [46]. Network traffic is higher in a dynamic machine because the tags must be carried with the tokens. Also, special tagging nodes must be placed in graphs, which expands the code size. The main advantage of the dynamic model is that a particular node may consume more than one token pair simultaneously (on different PEs), thus increasing the concurrency in the dataflow graph. Tagged token dataflow machines take full advantage of all forms of dynamic concurrency by *unravelling* the graph at run time, §4.3.7 and [7]. A disadvantage is that algorithms which cannot use this feature must inherit the cost of tagging the tokens.

2.3 The RMIT Hybrid Architecture

The RMIT architecture uses a hybrid scheme in which a modified form of the static model coexists with the tagged dynamic scheme. Tokens are of variable length and may or may not be tagged; tokens without tags may be regarded as an optimisation of tokens with the special tag 'zero'. If a token arrives at a node where there is already a token with the same tag present on the same input, then the tokens are queued until partner tokens arrive, this is opposed to the more usual case where it is an error for such a queue to form. A separate queue is maintained for each different tag and for those tokens with no tag at all; the matching unit can be optimised for tokens with no tag. When a partner arrives, a token is removed from the head of the appropriate queue. If many colours are present, then many instances of the node may execute in parallel because the hardware is capable of unravelling, or distributing the graph according to colour. Alternatively, if there is a queue of tokens with the same colour, then the program may take advantage of pipelining in the hardware to still achieve high concurrency, although not potentially as high as in the unravelled case.

The advantage of the hybrid arrangement is that the cost of tagging the tokens is not present when tagging is not required. Because the queued static mode of operation does not demand that each arc only hold one token, the potential concurrency is higher than in Dennis' static model. Figure 2.2 shows how the hybrid model combines the queued static and dynamic models.


Static Model -
single tokens without colour

Static Queued Model -
queues of tokens without colour

Dynamic Model -
single tokens with different
colours on the same input(s)

Hybrid Model -
queues of tokens with different
colours on the same input(s)

FIGURE 2.2 Static, dynamic and hybrid models

2.4 Characteristics of the RMIT Dataflow System

The dataflow variant used at RMIT is based on Egan's original FLO model [23, 26] and is characterised by the following attributes:

- The hybrid architecture combines a queued static execution model with an unravelled dynamic model.
- Node functions are weakly typed.
- Tokens are strongly typed and of variable length.
- The system supports shared subgraphs which facilitate multiple recursion and reduced code size.
- Graphs are both statically and dynamically partitioned to achieve a high machine utilisation. Note that dynamic unravelling requires multiple copies of the graph to be stored in the machine at run time.
- Storage nodes are provided to allow the graph to retain 'semi-permanent' information.
- Exceptions are handled using a special error token type.
- Open and closed streams are supported.
- I/O is accomplished using predefined nodes.
- Nodes may send tokens to many destinations either by using a tree of duplicate nodes, or by stepping through a destination list.

The RMIT system takes a 'low level' approach to dataflow whereby the granularity of the underlying node set is quite fine and high rates of achieved concurrency are required to produce efficient execution. This has come about because of a desire for flexibility in the node set, thus specialised and possibly complex node functions have been avoided. In such an environment, the design of the communications network is of paramount importance since the dynamic token load is usually very high. Simulations have shown that even the simplest of graphs which do any kind of looping or recursion can generate very large token traffic indeed (see ch. 5). For these reasons, together with an emphasis on 'eager, data driven' evaluation [5, 39], the RMIT system calls for a high degree of buffering within processing elements and communications network modules, hence the inclusion of the input, pipe, local and output buffers.

Nodes are currently limited to two inputs and two outputs (in general), which greatly simplifies the design of the matching unit, as with more than two inputs, the node firing conditions can become very complex and expensive to implement. The simplest firing condition possible for a node with more than two inputs is based on the presence of all of its inputs, and designs which feature more than two inputs usually allow even more complex firing conditions to exist [32, 46]. These firing conditions could be easily programmed into the interpretive emulator, but this approach has been avoided since it would not form a solid basis for the eventual transition to a dedicated discrete design, implementing the matching unit as a simple state machine. Some nodes (e.g. `replicate`) have more than two outputs, and the inclusion of a specialised result distribution unit will allow all nodes to have multiple outputs in future implementations.

The complexity of token queueing in the RMIT system can be seen in the structure of the matching unit, which rather than simply reserving a slot for one token for each two input node, must maintain separate pointers to the heads and tails of token queues. The situation is made even more complicated by dynamic token tagging. As it is very difficult to generate predictable tag sequences the associative search to find the queue with the correct colour is non-trivial [18]. A hashing strategy to access the token queues held in the arc store has been proposed and appears to be the best compromise for the RMIT system [1]. Currently, the emulator conducts a simple linear search for a queue with the correct colour, as in the original FLO specifications (see also §3.1).

2.5 Execution of Dataflow Graphs

At their lowest human readable level, dataflow graphs are represented in a textual form known as the intermediate target language (ITL) (appendix D and [23]). The ITL consists of a list of node descriptions and priming tokens which represents the initial *live* state of a machine graph. For a graph to be executed, these node descriptions and priming tokens must be translated from ITL to machine binary format and then loaded into the dataflow machine. This loading is done by representing the graph as a collection of node type tokens which are directed to a predefined node store node in each element which requires a copy of that node. Priming tokens are just data tokens sent to the previously loaded nodes, clearly, the graph must be loaded before the priming tokens. Once the graph and priming tokens have been loaded, execution may commence by the detection of *live* or *executable* nodes by the matching unit(s). Execution proceeds as live nodes *fire* and generate result tokens which are sent on to successor nodes. The process terminates when no more live nodes can be found.

2.6 The Prototype Emulator

The prototype emulator is a medium performance, single board implementation of the fast PE layout of figure 2.1. Its structure is shown in figure 2.3.

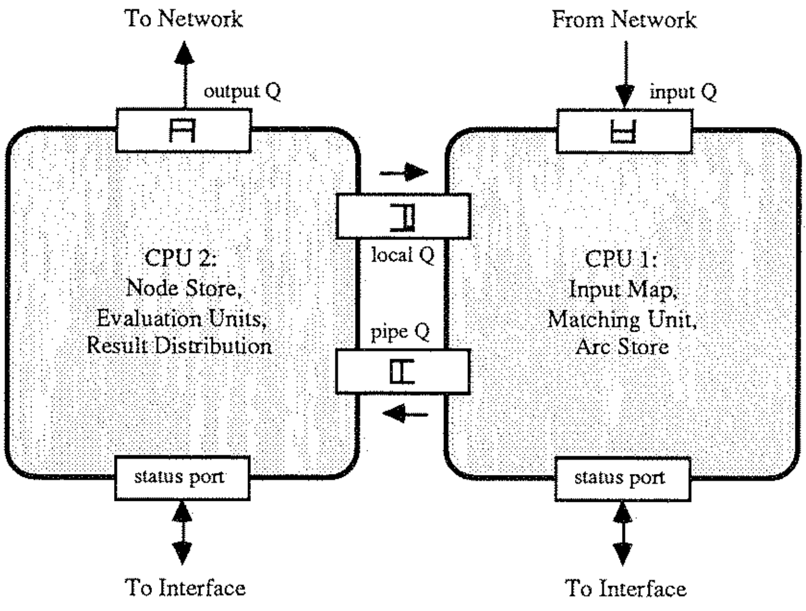


FIGURE 2.3 The prototype emulator layout

Detailed characteristics of the prototype emulator are described in appendix D, while timing information for the execution of nodes on this hardware is given in the node set description of appendix C. This machine forms the basis of the new multi-element emulator being constructed at RMIT, which will be capable of running much larger and more practical applications then the single board prototype, [52].

2.7 Token Format

The token formats used in the RMIT dataflow system are based on the FLO formats described and justified in [24], but have been modified to reflect the change from eight bit to sixteen bit CPU emulation. Appendix D gives a more precise description than that presented here, while reference [25] describes new binary formats to be used in the next generation architecture. Figure 2.4 shows the basic token format used in the 16 bit emulator and the simulator DFSIM.

Context	Element #	Node #	Colour (if present)	Type	Length	Value
<8>	<8>	<16>	<32>	<8>	<8>	<variable>

FIGURE 2.4 The 16 bit oriented token format

Figure 2.4 is a slight simplification of the actual format used, since some hidden bits have special meanings, e.g., colour present, node input point, etc.. The token fields as shown have the following meanings:-

Context	This field is part of a virtual addressing scheme to be used in the next generation machine, it separates the graphs of different users.
Element #	The destination element number used to route tokens through the communications network. This field is ignored internally.
Node #	The graph address of this token's destination node. The node space uses a linear addressing scheme with node numbers of 1 and upwards.
Colour	The dynamic tag or label. This field is only present if an appropriate bit is set in the *Node #* field. Token's without a colour behave as if their colour was zero, the matching unit being optimised for this case.
Type	The type of the token's data, see appendix D.
Length	The length of the token's data fields. The *end-of-stream* token has zero length. *Type* and *Length* together make up the so called *data header.*

2.8 Node Format

Once a graph has been loaded, the node store (CPU 2 in the emulator) has access to node descriptors of the format shown in figure 2.5. These descriptors are loaded into the machine by the *node* type tokens of the ITL machine graph description.

Function #	Flags	Literal	Dest1	Dest2	Link
<8>	<8>	<16 or 32>	<32>	<32>	<16>

FIGURE 2.5 The 16 bit oriented node descriptor format

The node descriptor fields have the following meanings:-

Function #	An 8 bit vector number into a table of node function routines.	
Flags	A set of flags with the following meanings:-	
	I	one input node
	D	literal data present
	T	this node is to be traced
	E	an extension bit used by the emulator as a link into the node extension store for node descriptors > 5 words
Literal	A literal data field ≤ 2 words in size, larger literals are stored in the extension store.	
Dest1, *Dest2*	32 bit destination node addresses (*Dest2* may be held in extension store).	
Link	A pointer into the extension store for long descriptors.	

2.9 The Simulation Environment, DFSIM

Although the emulator has proven useful in the development of PE hardware design criteria, it has not been so useful in general dataflow programming development. This is largely due to the limitation of only having one processing element to study, with a limited range of statistics being returned (this situation will change in the near future with the commissioning of the 16 element emulator). On the other hand, the program development and simulation environment that exists entirely on the Unix workstations is not only extremely convenient to use, but also returns much more meaningful results in terms of multiprocessor performance characteristics. For these reasons, most of the language development work carried out under this research concentrates on the simulation environment.

The dataflow machine simulator, DFSIM, was developed for use with the original FLO system, but has been adapted to keep pace with alterations to node set and matching function specifications. It provides a discrete event based simulation of a multiprocessor consisting of any number of simplified, non-pipelined processing elements. Communications network delays are modelled (in the form of destination clashes) but delays within the ring structure of a single PE are not, e.g. it is assumed that node function evaluation may proceed immediately upon the detection of a live node by the matching unit. In the real pipelined PE however, node evaluation will not proceed until an idle evaluation unit has been found.

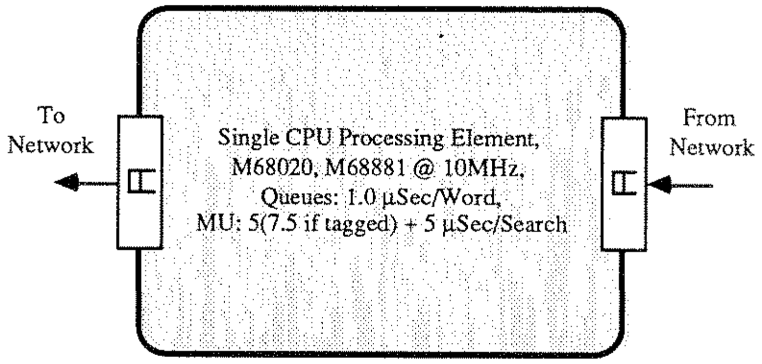


FIGURE 2.6 The processing element modelled by DFSIM

Figure 2.6 shows the format of the simulated processing element. The timing figures are based on a system with the hardware characteristics shown. As a result of this simplified PE structure, figures of concurrency due to the pipelining within each PE are not available and absolute timing figures are therefore approximate but never the less 'reasonable'. In any event, most of the important simulation results come about from a comparative analysis between separate simulations and are not influenced by actual time values. Most importantly, simulation results for the performance of a cluster of processing elements are perfectly valid and accurately show the effects of varying the number of PEs and the graph distribution. In short, the results of many simulations have proved extremely useful in identifying the strengths and weaknesses of the RMIT dataflow model.

2.9.1 Simulation Details

The simulator has a resolution of 100 nSec, all times are a multiple of this figure. As seen in figure 2.6, tokens are assumed to be read from and written to queues at a rate of 1.0 µS per word; tokens of varying lengths are accurately simulated using this figure. Note that the input and output queues are correctly simulated; they are entirely independent of the processing element internal operation. Network contentions are handled by detecting the situation where two output queues are scheduled to write into the same destination input queue during the same time interval. DFSIM keeps a count of the maximum queue occupancy during the simulation.

The matching unit is assumed to take a time of 5η µS / search where η is a factor that reflects the efficiency of the hashing scheme used for tagged tokens [1]. Currently η is set to 1 for untagged tokens and 1.5 if a tag is present (the matching unit can handle untagged token traffic without hash clashes). In addition, there are additional penalties for both successful and unsuccessful searches due to the time taken to process an entry from a located queue or to establish a new entry if the search was unsuccessful; these are currently set to 5 µS and 10 µS respectively.

The execution of nodes is straight forward, with the execution times shown in table 2.1 being used. There is an extra time of 5 µS added to every node evaluation to allow for node store accessing. Result distribution is done by scheduling tokens to be written into the output queue after node evaluation is complete.

Tokens carry an earliest possible creation time tag with them during simulation; it is this time that derives the figures for 'average potential concurrency' and 'potential execution time' for the graph. Simulated time is used to derive the 'average actual concurrency' and 'actual execution time' figures. Average concurrencies are given as the ratio of aggregate machine run time (i.e. the time it would take one element to execute the graph) to the creation times of the last token (shortest time for potential concurrency, and simulated time for actual concurrency).

	Node Mnemonic	Evaluation Time (µS)
arithmetic nodes:-		
	ADD, SUB	5.1
	ABS	3.5
	MUL	7.1
	DVD	10.3
	MOD, DIV	5.0
	NEG	3.5
	SQT	10.7
	SIN, COS	39.1
	TAN	47.3
	ASN	58.1
	ACS	62.5
	ATN	40.3
	LOG	58.1
	LNE	52.5
	EXP	49.7
	SQR	7.1
	RND, TRN	5.5
fast nodes:-		
	DUP, REP, ID, PIP, PRT, PRS, SYN, FIR, S, STS, HD, TL, CON	2.0
default nodes:-		
	All others (including system nodes)	5.0

TABLE 2.1 The node evaluation times used by DFSIM

Machine utilisation is also reported by DFSIM; it is the ratio of average actual concurrency to the number of processing elements used and is a useful figure of merit for a particular graph/machine configuration. This figure can be expected to degrade as the number of processing elements exceeds the available graph concurrency, see §5.7.

2.9.2 An Example

As an example, consider the following simulation results for a sequential solution to the 8 queens problem, figure 2.7. The DL1 source is similar to that of the recursive 6 queens solution given in appendix A.

```
 Data-flow Machine Simulator - Graph File: 8queens.itl - Thu May 28 15:47:25 1987
 Configuration: elements [0..127], element-nodes [0..2047]
 Options: Simlation Hash-Element Hash-Search

 initialise queues
 initialise MS
 initialise NS
 loading commenced
 nodes=1276 priming tokens=27 max element=127

Solution number 1    Solution number 2    Solution number 3    Solution number 4
(Tests made  876)    (Tests made  264)    (Tests made  200)    (Tests made  136)

Q . . . . . . .      Q . . . . . . .      Q . . . . . . .      Q . . . . . . .
. . . . Q . . .      . . . . . Q . .      . . . . . . Q .      . . . . . . Q .
. . . . . . . Q      . . . . . . . Q      . . . Q . . . .      . . . . Q . . .
. . . . . Q . .      . . Q . . . . .      . . . . . Q . .      . . . . . . . Q
. . Q . . . . .      . . . . . . Q .      . . . . . . . Q      . Q . . . . . .
. . . . . . Q .      . . . Q . . . .      . Q . . . . . .      . . . Q . . . .
. Q . . . . . .      . Q . . . . . .      . . . . Q . . .      . . . . . Q . .
. . . Q . . . .      . . . . Q . . .      . . Q . . . . .      . . Q . . . . .
```

```
Solution number 5 …  Solution number 90   Solution number 91   Solution number 92
(Tests made  504) …  (Tests made  136)    (Tests made  200)    (Tests made  264)

. Q . . . . . .   …  . . . . . . . Q      . . . . . . . Q      . . . . . . . Q
. . . Q . . . .   …  . Q . . . . . .      . . Q . . . . .      . . . Q . . . .
. . . . . Q . .   …  . . . . Q . . .      Q . . . . . . .      Q . . . . . . .
. . . . . . . Q   …  . . Q . . . . .      . . . . . Q . .      . . Q . . . . .
. . Q . . . . .   …  Q . . . . . . .      . Q . . . . . .      . . . . . Q . .
Q . . . . . . .   …  . . . . . . Q .      . . . . Q . . .      . Q . . . . . .
. . . . . . Q .   …  . . . Q . . . .      . . . . . . Q .      . . . . . . Q .
. . . . Q . . .   …  . . . . . Q . .      . . . Q . . . .      . . . . Q . . .
Total number of solutions          92
Tests made to exhaust board       868

                         Potential    Achieved          %
         Nodes Fired                   2897628
    Tokens Generated                   4256031
         Time (Sec.)    8.509193     9.704139
     Av. Concurrency        18.3         16.1        87.7
  Node Functions/Sec.     340529       298597
  Machine Utilisation                                12.6

            Activity           %
     Node Evaluation        15.8
         LQ/OQ Write        16.5
          LQ/IQ Read        16.5
      Matching Store        50.9
     Structure Store         0.0
        Network Wait         0.2

 Func     uS Time%     #%  Mon Diad Stor Frst Prot Prme Strm Head Tail Cons
  SYS     10   0.1    0.1
  DUP      7  38.7   47.0 47.0
  PRT      7   0.8    1.0                       1.0
  LMC     10  10.8    9.2                       9.2
  EMC      7   1.0    1.2                       1.2
  ADD     10   3.3    2.8  2.0  0.8
  SUB     10   1.0    0.8       0.8
  A       10   1.2    1.0  1.0
  R       10   0.8    0.7  0.7
  E       10   0.8    0.7       0.7
  SWI     10  17.2   14.6      14.6
  PIT     10   7.0    6.0  0.2  5.8
  PIF     10   9.8    8.3  0.0  8.3
  PIP      7   0.6    0.7  0.4  0.3
  PRS      7   0.1    0.2       0.2
  AND     10   1.3    1.1       1.1
  TSB     10   1.9    1.6       1.6
  STB     10   0.5    0.4       0.4
  CLB     10   0.5    0.4       0.4
  EQ      10   0.9    0.8  0.8
  LT      10   0.2    0.1  0.1
  LE      10   0.6    0.5  0.5
  CHR     10   0.0    0.0  0.0
  STD     10   0.2    0.2       0.2
  YLC     10   0.3    0.2  0.2
  STC     10   0.3    0.2       0.2
                          53.0 35.5  0.0  0.0 11.4  0.0  0.0  0.0  0.0  0.0

 Assumptions: Single CPU PE (M68020, M68881 @ 10MHz), Queues 1.0 uSec. per word
 MS 5x1.0{1.5(Colour)} + 5(Success){10(Fail)} uS.

  Locality      Words      W/Sec         %
     Local     201928      20808       0.8
 Non local   25578044    2635787      99.4
```

FIGURE 2.7 Simulation of 8queens.itl (the sequential 8 queens solution)

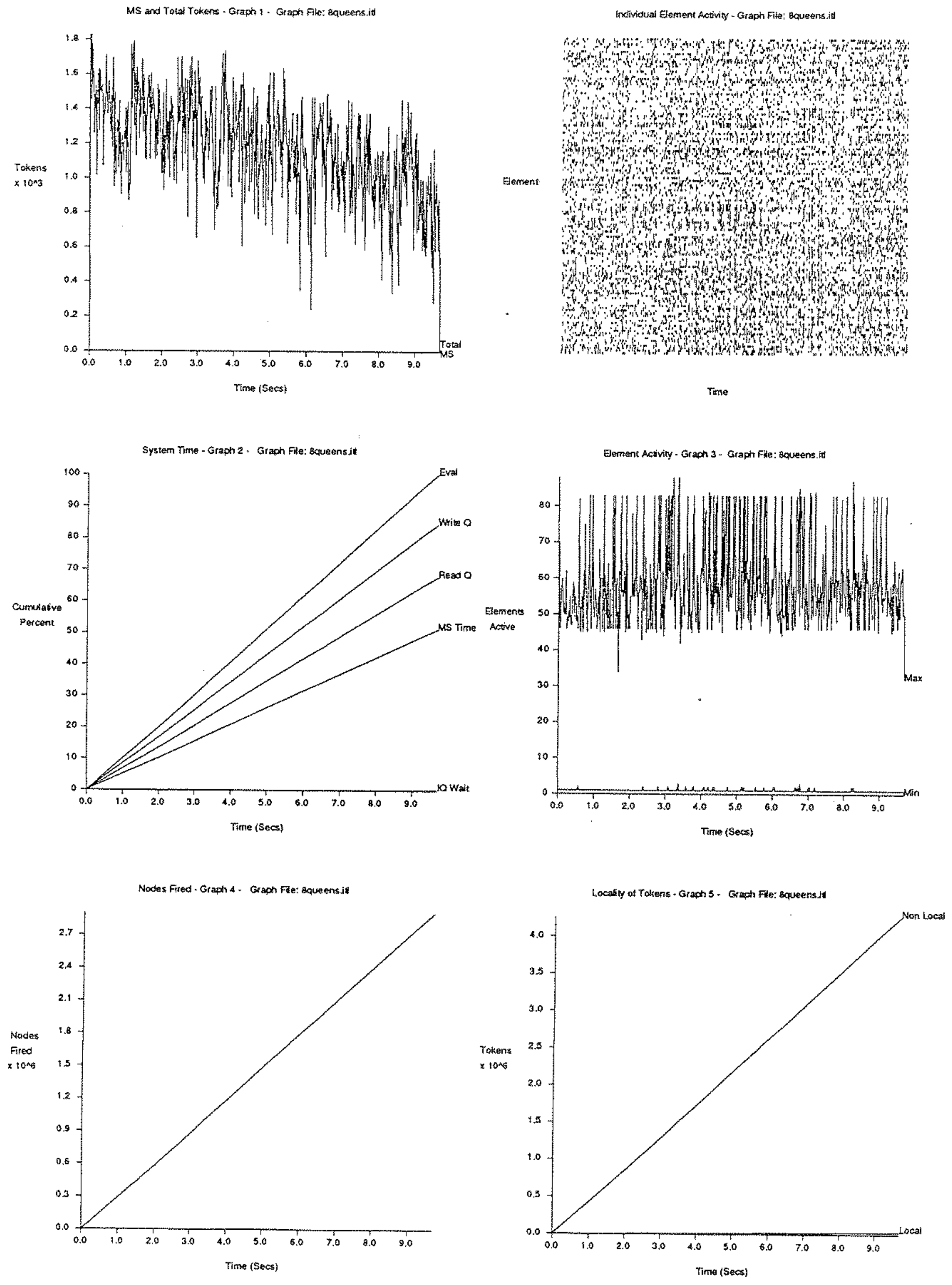


FIGURE 2.8 Performance Graphs for 8queens.itl

2.9.3 Explanation of Simulation Results

Referring to figure 2.7, DFSIM reports the simulated machine configuration, the options used in the simulation, and the static graph size (nodes and priming tokens). Output from the simulated program follows (in this case eight of ninety two solutions to the eight queens problem are shown).

A table of run time results follows, in which the average concurrencies, execution times and machine utilisation are given. This example was carried out in full simulation mode, but there is also a simple emulation mode which emulates a machine with just one processing element, making the simulation faster, although the true values of actual concurrency and execution time are not returned. Next is the activity table which shows the percentage of aggregate time spent performing different tasks; this table does not indicate 'idle' time, which is obtained from the machine utilisation figure.

The dynamic breakdown table shows figures for individual nodes and matching functions. Node evaluation times are shown, including the extra 5 μS allowed for node store accessing. Other entries indicate the percentage of aggregate machine time taken by each node (`Time%`) and the percentage by number for each node and matching function (`#%`). The locality table shows the number of words sent to local and output queues (tokens average about 6 words each for this graph).

Other information available, but not shown here, are tables of token usage by type and length, and a table showing the maximum number of tokens in each matching unit (arc store), local queue, and input queue. Finally, data files are produced which enable the graphs shown in figure 2.8 and chapter 5 to be plotted.

Figure 2.8 shows graphs of machine activity during the simulation. Graph 1 shows the number of tokens in the matching store and in total. Graph 2 shows accumulated time as a percentage of time spent in the different modes of machine operation, idle time is not shown; note that the percentages are cumulative. Graph 3 shows the maximum and minimum number of elements active during each time step. Graph 4 shows nodes fired vs time, its slope corresponds to machine activity. Graph 5 is similar, but shows the number of tokens sent locally and externally during the run. Finally, there is an individual element activity picture which shows a black spot for each element that is active during each interval. The vertical axis represents elements 0 to 127, while the horizontal axis corresponds to simulated time as in the other five graphs. Graph hot spots show as dark areas in the activity picture. Note that graph 3 is the 'integral' of the activity picture in the vertical axis.

Chapter 3
MATCHING FUNCTIONS

3.1 Hybrid Matching Functions

Matching functions are the rules which govern the operation of the matching unit in determining which nodes are ready to fire. Because of the special hybrid nature of the RMIT architecture, several unique and diverse matching functions are supported, and research into the value of additional functions is on going. It must be emphasised that because of the experimental nature of the project at this stage, the matching functions to be used in the next generation processing element will not necessarily be the same as those described here.

The original FLO matching functions are *monadic*, *diadic* and *storage*, but this research clearly demonstrates the advantage of the additional matching functions: *first*, *prime*, *simple protect*, *complex protect*, *stream*, *head* and *tail*. Most of these new matching functions arise through the queueing of tokens on arcs. This chapter explains these functions and their implementation in some detail, especially with regard to future, possibly discrete matching unit implementations.

The matching functions are illustrated by state transition diagrams. A practical implementation might include a micro-programmed controller to carry out the action associated with each transition, supported by parallel hardware to determine any special conditions leading to the next state. Actually, the diagrams could be implemented by a simple, dedicated state machine, but there is a need to maintain flexibility in an experimental design of this nature.

The state information implied in these diagrams is a combination of information held in the *input map* and information obtained by associatively searching the *arc store*. For example, a two input node may have tokens of several different colours stored in the arc store. In this case, the input map entry would merely indicate that the node is not empty, and the associative 'colour search' is made to find stored tokens of the same colour as the current input token, or to detect emptiness for that colour. The colour search is the critical operation of the matching unit and has been shown to be a limiting factor in the performance of dynamic architectures [29]. A simple linear search has been shown to work adequately for a large class of problems [18], but is clearly inadequate in the general case since no assumptions can be made about the way colours will appear in the graph. Reference [1] outlines a proposal for an efficient associative search mechanism to be used in the next generation RMIT machine.

3.2 Monadic

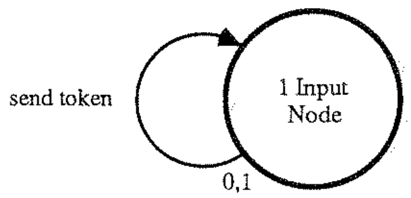


FIGURE 3.1 The *monadic* matching function

The *monadic* matching function is shown in figure 3.1. Tokens arriving at a one input node (or a two input node with a literal) are simply sent on to the next stage in the pipeline (the *pipe queue*), and the node remains in the '1 Input Node' state. *Monadic* is the simplest of all the matching functions and most dataflow architectures treat it as a special case, since tokens destined for a one input node may bypass the search unit altogether. This is usually achieved by having tokens carry the matching function of the destination node with them (attached by the predecessor node), however, in the RMIT architecture the matching function is stored in the *input map* at the beginning of the pipeline in each processing element. Incoming tokens index the input map, based on node number, where the matching function is found along with the current state of the node. In the emulator code, the '*monadic/1 input node*' state is just one entry in a vector table of all possible states.

3.3 Diadic

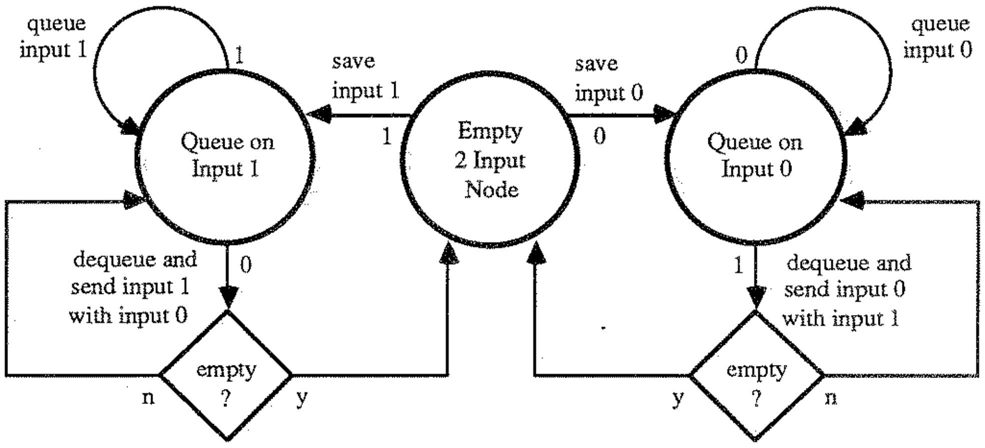


FIGURE 3.2 The *diadic* matching function

Diadic is the simplest matching function which actually requires arc store accesses to be made. In most dynamic dataflow models, *diadic* is made very simple because only one token is allowed per arc at run time. This greatly simplifies the structure of the arc store since there is no need to hold queues of tokens. The advantage is often negated to a large extent however, because the hash tables usually used to implement such an arc store still require queues or special overflow processing to cope with synonyms [16]. A truly associative token store would not suffer from this problem and possibly represents the best answer to the matching problem, despite its difficulty to implement. Simple arc stores made from Content Addressable Memories (CAMs) have been proposed but are limited in size by the present state of CAM technology, although this situation should change in the future. In any event, a simple associative arc store might not be suitable for some of the more complicated matching functions that have been defined.

In the RMIT system, *diadic*, shown in figure 3.2, is rather unique in that unlimited unmatched token queues are allowed on the inputs to diadic nodes. Separate queues are maintained for tokens of different colours, but the arc store is optimised for tokens with no colour. This optimisation, together with the fact that tokens without a colour field (tag) are physically shorter than those with a colour, is shown to improve the performance of graphs that do not require coloured tokens (§5.6).

3.4 Storage

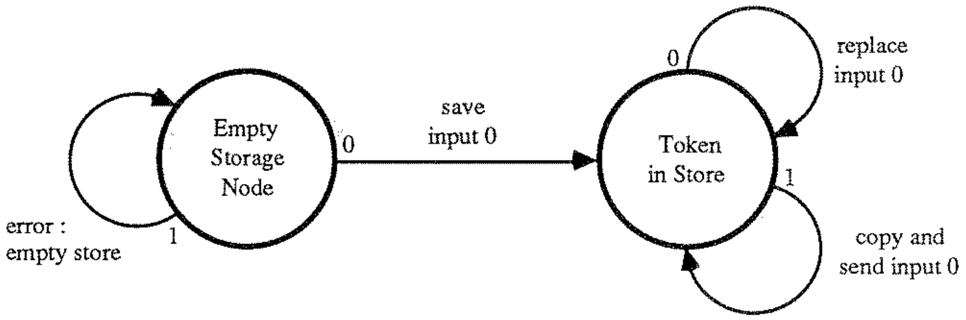


FIGURE 3.3 The *storage* matching function

The *storage* matching function, figure 3.3, is the last of the original FLO matching functions to be incorporated in the current RMIT system. Some examples of uses of the `storage` node (`S`), which is the only node to use *storage*, are given in [24], where use is made of their inherently indeterminate nature in a 'set point controller' as part of a decentralised industrial control system. This node can also be used to implement a general lazy evaluation scheme for some languages (§4.3.4).

3.5 First

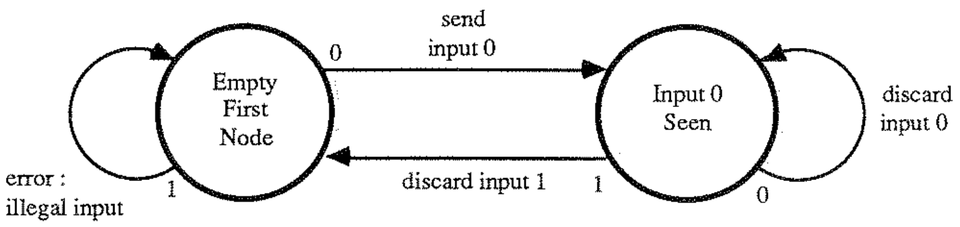


FIGURE 3.4 The *first* matching function

The *first* and *prime* (§3.6) matching functions were introduced to allow the priming of shared subgraphs (§4.3.5). Both of these functions work by recording state information in the arc store to show that a particular colour has been seen on input 0 of this node, see figures 3.4 and 3.5. The first call to a shared subgraph can thus be detected, and appropriate action taken to simulate the effect of priming tokens. *First* is considered the more fundamental of the two functions and has been accepted into the basic node set in the form of the `first` node (`FIR`). This node transmits input 0 unchanged if its colour has not been seen. Successive tokens on input 0 are discarded by the matching unit. Many examples of `first` node usage are given in the code templates of chapter 4.

`First` and `prime` nodes have the disadvantage of leaving state information in the matching store, so that graphs which use either of these nodes are *unclean* (§4.2.5). The problem can be largely overcome by applying special code transformations which detect the completion of a function invocation (e.g. by detecting the return of all or even just some of the outputs) and *signal* the graph to reset any `first` and `prime` nodes by directing a token to input 1. This cleans the arc store for that colour. It can be very inefficient however, to clean out the graph for one invocation, only to find that a new data set enters that code segment with the same colour, as can happen in a queued environment. Similar problems have been reported by other researchers [28], with the result that the code transformations described are not yet implemented in our system; more reliance is currently made on graphs that do not need the *first* function.

3.6 Prime

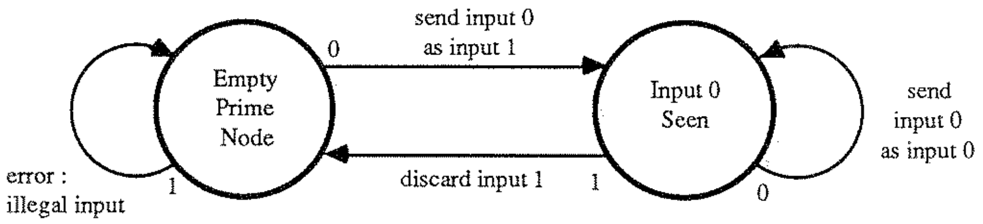


FIGURE 3.5 The *prime* matching function

The *prime* matching function, figure 3.5, is similar to *first* in the way that state information indicating that colours have been seen is saved in the arc store. The `prime` node (`PRM`) was designed to optimise a particular use of the `first` node (as used in the basic expansion of **head** in §4.3.8.4), but is not considered of fundamental importance. Thus, the compilers for the RMIT system will only plant `prime` nodes if an extended code generation toggle is set, the default setting for this toggle is off (see appendix B). The `prime` node always has a literal associated with it, and its operation is to output the literal and the input token when the colour has not been seen. Only the input token is output if the colour has already been seen. This effectively primes the output arc with a copy of the literal token.

Prime requires a special interaction between the matching store and the execution unit, since the matching store has to inform the execution unit whether to send the literal or not. The method currently used is to use the input point bit in the input token's description (§2.7, app. D) to indicate whether to send the literal or not. The input token always arrives on input 0 of the `prime` node, but if the colour has not been seen then the matching store sets the input point bit to a 1. The execution unit simply inspects the input point bit and if it is set, the literal and the input token are sent, else just the input token is sent.

Although the current implementation only uses *prime* on the `prime` node, it would be possible to generalise it to any input of any node. This has not been done at present because of the unclean problem that exists with *prime*. In fact, the problem is made even worse in the case of diadic nodes where there would be no way of providing a reset input, and a diadic node with a literal already present clearly could not use *prime*.

3.7 Protect

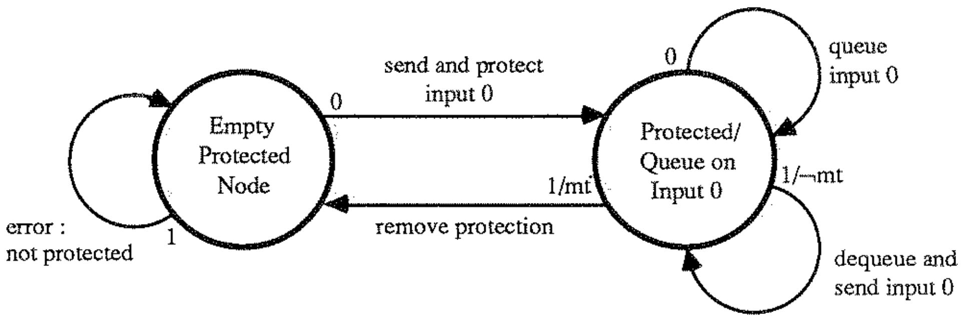


FIGURE 3.6 The simple *protect* matching function

The *protect* matching function, figure 3.6, is a recent addition to the system and is only used with the extended code generation option. It serves three very useful purposes:

- It avoids a priming of shared subgraphs problem with **protect** statements
- It provides a useful optimisation for certain code templates (i.e. static iterations which would otherwise use a controlled merge construct, §4.3.6.1)
- It solves the problem of unclean graphs and indeterminacy of DL1 conditionals and loops (§4.3.3 & §4.3.6)

Certain characteristics of the code templates which use protected nodes allow a simplified implementation of the matching function that does not allow tokens to queue up on input 1 of a protected two input node, e.g. the controlled merges of conditionals. Hence, no provision is made for queues on input 1 in the simple version of *protect*. Protected nodes behave as if a *boolean true* priming token is present on input 1 before the graph starts to execute. An initial token passes through input 0 and enters into the graph region requiring protection, (usually a non-reentrant graph segment required to support queues). When the protected region terminates, a token is returned to input 1 of the protected control node which cleans the arc store and allows more tokens into the protected region. The important difference between *protect* and *first* is that while input 0 is blocked, *protect* queues tokens on input 0 whereas *first* discards them. *Protect* is correspondingly more difficult to implement than *first*.

For all protected nodes, input 1 acts as a control input which is asserted by a *boolean true* token, thus the *protect* function only makes sense for a small but very important subset of the entire node set (i.e. all the nodes which may require *boolean true* priming tokens). The claim that *protect* solves the unclean graph problem for DL1 loops and conditionals is justified by observing that the only priming tokens required here are indeed *boolean true*. Similar solutions can be applied to other templates with other types of priming tokens, but the problem was greatly simplified by limiting *protect* to *boolean trues* when designing the extended code templates for the enhanced DL1 compiler. Weng's TDFL language [50], which DL1 is partly based on, also has this characteristic, although the fact was unexploited in that language.

Like other matching functions, a special set of nodes has been defined to use *protect*, these nodes are `eager-merge-control (EMC)`, `lazy-merge-control (LMC)` and `protect (PRT)` (§4.3.3 and appendix C).

3.8 Complex Protect

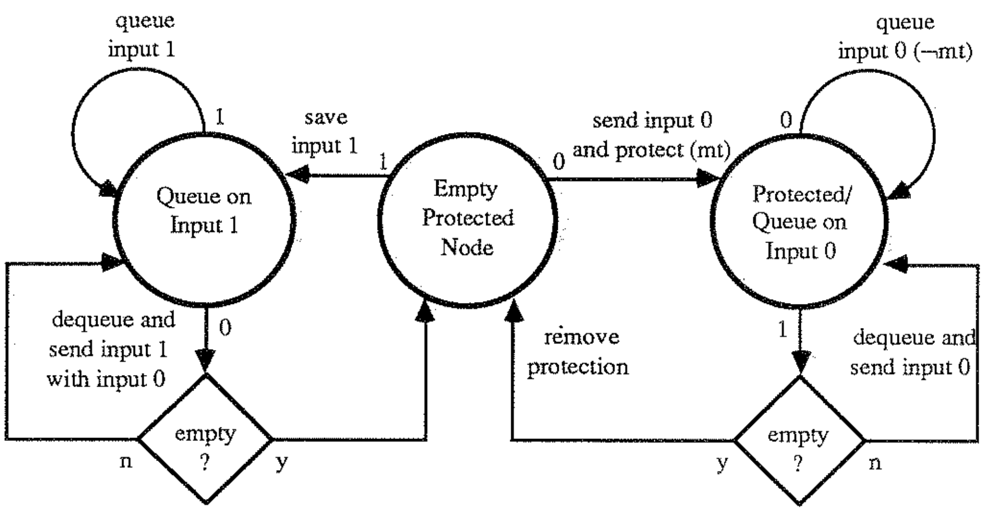


FIGURE 3.7 The *complex protect* matching function

Figure 3.7 shows the *complex protect* matching function, which is identical to simple *protect*, except that queues are allowed on input 1. This allows resetting tokens to be sent to input 1 before all of the corresponding input tokens have passed through input 0. Interestingly, this matching function is potentially easier to implement than simple *protect* because it has the characteristic of behaving as a simple one input node for initial tokens arriving on input 0 and then effectively becoming a two input node until the protection is removed. Tokens always queue on input 1 as in an ordinary diadic node. This operation is made clearer by observing the similarity between the state diagrams for *complex protect* and *diadic*.

Although not implemented at this stage, *complex protect* will be of benefit in a revision of the RMIT node set currently being defined [25]. In this revision, the nodes `up-sequence (US)` and `down-sequence (DS)` generate a sequence of integer counting tokens and boolean control tokens to improve the efficiency of **for** loop type constructs. The control tokens can then be queued on the control inputs of `protect` nodes at the inputs to static iterative constructs, but such nodes would have to allow queueing on that input, i.e. these nodes would require the *complex protect* matching function.

3.9 Stream

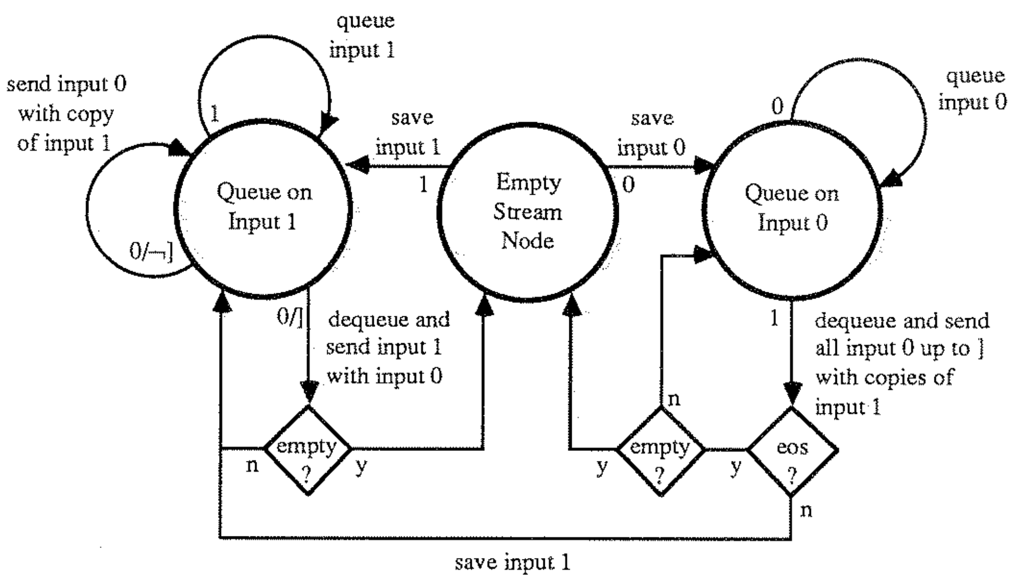


FIGURE 3.8 The *stream* matching function

The *stream* matching function, figure 3.8, has been designed to improve performance of stream macros (§4.3.8.2). Although shown for a stream arriving on input 0, it is equally valid when applied to input 1 of a diadic node (as in 'STS', see below). It is a very flexible function because its use is valid for nearly all nodes. Its operation is simply to reuse a token on a node's input (0 or 1) for every element of a stream that arrives on the complementary input.

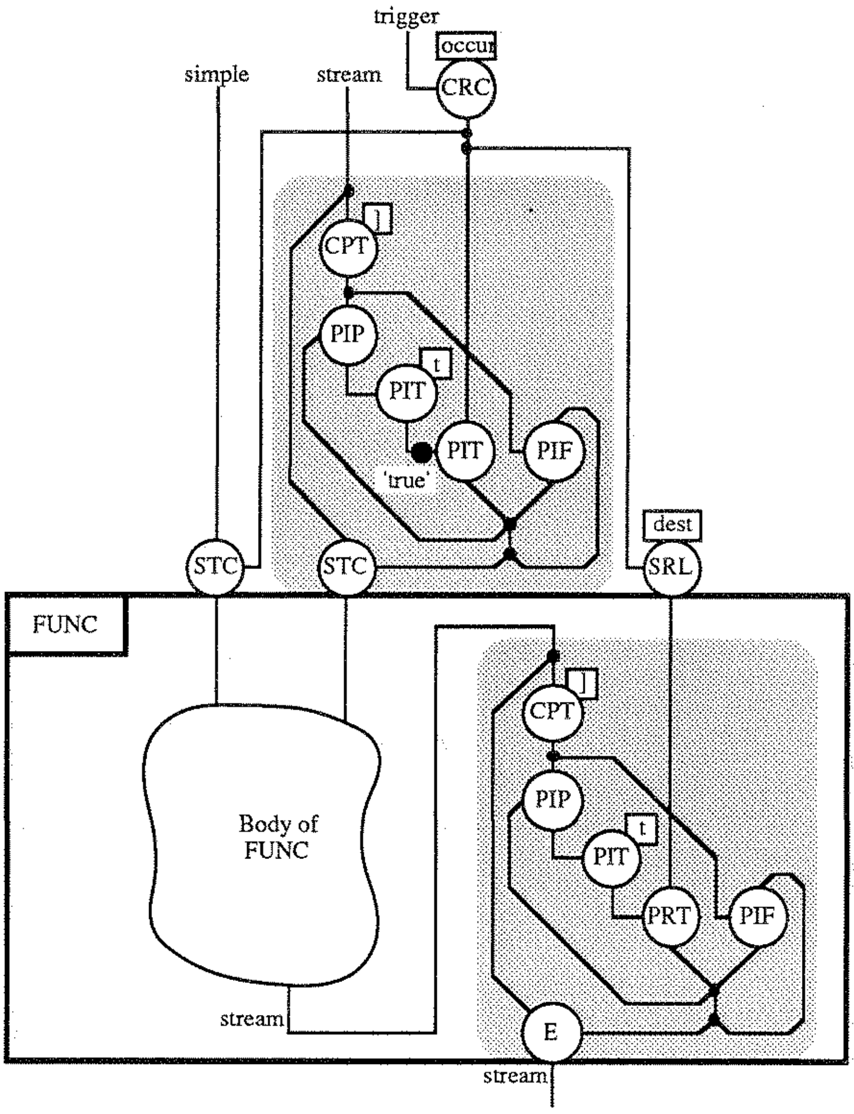


FIGURE 3.9 Typical discrete code (shaded) for stream functions

Using *stream*, stream entry and exit to and from shared subgraphs using the new *create colour / set colour* mechanism (§4.3.5.2) is reduced from the complicated, inefficient structure of figure 3.9 to the simple, efficient structure of figure 3.10. In figure 3.9, the shaded areas show the code planted to regenerate the context (colour) of this call to FUNC for each element of the input stream, and to regenerate the return address for each element of the output stream. The code includes a priming token which is avoided inside the shared subgraph by enabling extended code generation, which results in the placement of a protect node. Note that only one return address is generated since the length of the output stream cannot be determined at compile time, of course it is not necessarily equal to the length of the input stream.

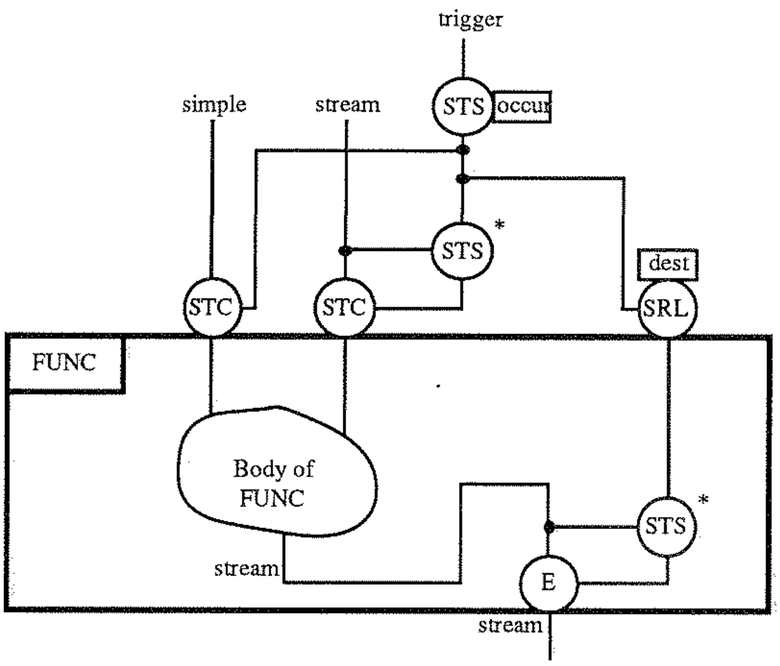


FIGURE 3.10 Improved code using *stream*

The `stream-store (STS)` node uses *stream* to replace the shaded macros of figure 3.9. In figure 3.10, `STS` is used in combination with other nodes (`set-colour` and `exit`) to effectively make these nodes operate over streams rather than simple tokens. As pointed out above, many nodes can be used in this way, especially control nodes like `pass-if-true`, `pass-if-false`, `switch`, etc., see also §4.3.8.4.

To further improve the performance of stream graphs, it would be possible to use *stream* on the inputs of existing nodes rather than always interposing the `STS` node. This transformation saves two nodes and two tokens per stream element per application and is the optimal strategy for stream graphs without actually defining even more powerful nodes specially for handling streams, see figure 3.11. In this figure, an 's' on the input of a node indicates the *stream* function applies to that node, with the tokens making up the stream arriving on the labelled input. The transformation requires that the matching function be specified independently to the rest of the node description, which is currently not the case because the original FLO node set made little use of special matching functions. A new node has always been defined to handle a new matching function to date, but in the future, node descriptions will include explicit rather than implicit matching function specification.

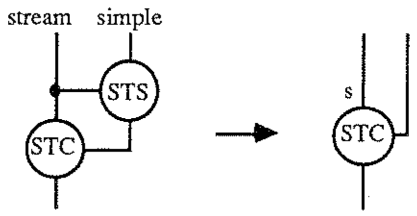


FIGURE 3.11 The transformation for optimal stream control

3.10 Head and Tail

Head and *tail* (figures 3.12 and 3.13), are simple matching functions which solve one remaining problem with stream graphs on the RMIT architecture. Problems exist with the predefined DL1 stream functions **head, tail, empty** and **get** such that it is difficult to produce clean graphs (§4.3.8.4). With these functions, the machine graph must be arranged so that it appears that a stream has just gone past, and for the graphs to be clean it is required that no priming tokens are used or state information be left behind. The problem proves to be difficult using the matching functions defined so far.

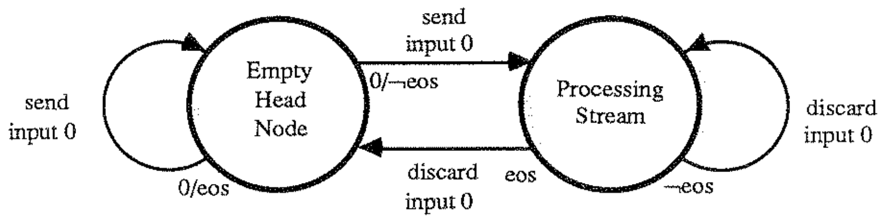


FIGURE 3.12 The *head* (of stream) matching function

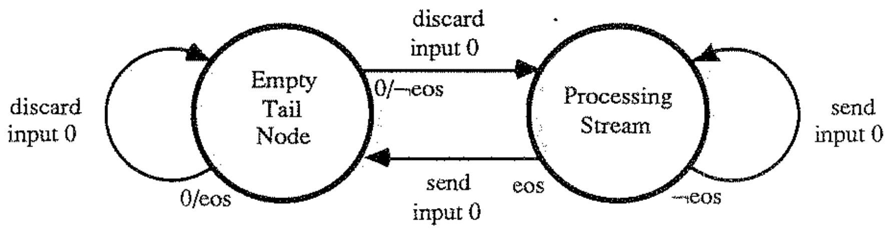


FIGURE 3.13 The *tail* (of stream) Matching Function

The new matching functions, *head* and *tail*, solve the problem by effectively carrying out the entire **head** and **tail** operations within the matching unit itself, **get** can be made up of a call to **head** and a call to **tail** while **empty** simply uses **head** and one other node. An input stream (to the matching unit) of say 1,2,3,4,] will produce an output (from the matching unit) of 1 for *head* and 2,3,4,] for *tail*. The execution unit simply passes these tokens to the destination node(s). Thus there are one node implementations of **head** (`head (HD)`) and **tail** (`tail (TL)`), a two node implementation of **empty** (one `head` and one `compare-type` to literal *end-of-stream*) and a three node implementation of **get** (one `duplicate`, one `head` and one `tail`). This gives major static and dynamic performance improvements (§5.4) over the older implementations, as well as producing the very desirable clean graph property for these functions.

3.11 Cons

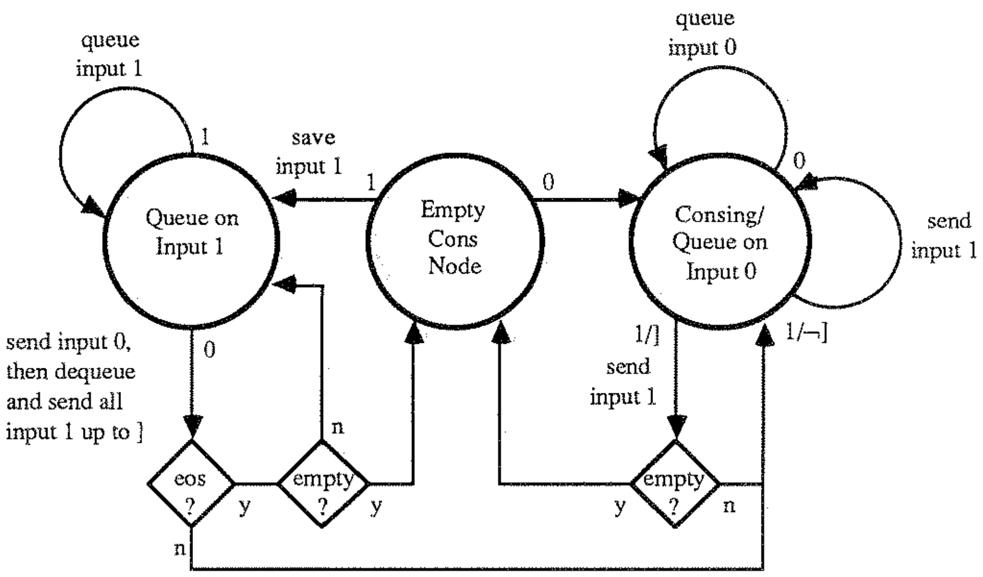


FIGURE 3.14 The *cons* matching function

Cons, figure 3.14, was added to complete the matching function based implementation of the five basic stream operators. Even though the templates for the stream **cons** function using the basic or extended node sets were clean, they still exhibited poor static code size and unacceptable dynamic performance, similar to **head** etc.. The state diagram for *cons* is similar to that for *stream* except that *stream* was shown applied to the opposite input and that the actions taken with certain transitions are somewhat different.

In the simulator, DFSIM, which does not exactly simulate the pipe queue of the RMIT processing element, *cons*, *tail* and *stream* have been implemented by having the routines that simulate the matching unit set a special flag variable. During node processing, while this flag is set, the evaluation routines continue to call the matching unit until no tokens are left.

Chapter 4
DATAFLOW LANGUAGES AND CODE GENERATION

4.1 Introduction

The development of a new computer architecture brings with it the need for new programming languages designed to extract the best possible performance from that architecture. It is impractical to program a dataflow machine in a sequential language like FORTRAN for example, since that language has been designed for execution on a sequential computer, although it is true that dataflow compilers can be written for these languages [27].

There are several features of imperative languages like FORTRAN that make them largely unsuitable for execution on a dataflow machine. Two of these features in particular are *multiple (sequential) assignment* and *side effect based computation*. It is interesting to note that both of these features have evolved largely as a consequence of program execution on sequential machines and not because the language itself requires their use in the expression of an algorithm, it merely permits it. In this chapter, these features and others will be examined with regards to their implications for the dataflow language DL1 (Dataflow Language One), which is currently the main language in use at RMIT. Problems with DL1 and the improvements that this research has led to are also explained by illustrating the dataflow code laid by the compiler for different high level constructs.

4.2 Properties of Dataflow Languages

Dataflow languages have been developed with many properties that suit them for data driven execution in a multiprocessor environment. In this section, the more important of these properties will be examined and some terms defined.

4.2.1 Implicit Control

In a dataflow language, the programmer is not required to specify execution control by writing statements in a certain order, rather, it is done at run time by the dataflow machine itself. Control is therefore implicit in a dataflow language, being derived solely from the data dependencies between identifiers, as opposed to the explicit control flow of imperative languages. This can result in a significant reduction in the complexity of the programming task itself and it is no accident that simplifications like this also make dataflow languages, and functional languages in general, more suitable for mathematical analysis, which is often touted as one of their most important features.

Some languages, e.g. P5, which is a dataflow implementation of a subset of PASCAL [51], still require statements to be written in a strictly 'define before use' order, but even in these cases, control is purely data driven and may not correspond to statement ordering. In fact, even a pure dataflow language could sensibly enforce the define before use rule, because it leads to the important class of *acyclic* dataflow graphs. Acyclic graphs do not suffer from the initiation and termination problems that are associated with their cyclic counterparts. This will become clear when iteration is discussed in §4.3.6.

4.2.2 Single Assignment

The *single assignment rule* is another simplification to the semantics of a language. It changes the interpretation of a variable from something that has a certain value at a certain time (i.e., the changing contents of a memory location), to something that is just defined in terms of other values and conditions, the precise physical representation being unimportant. Variables may not be reassigned in a single assignment language (SAL) and thus are often referred to as *items*, *identifiers* or *values*. Special cases do exist however, where the term 'variable' makes some sense because multiple assignment appears to apply. Thus *loop variables* are referred to in an iteration, although once a loop is unfolded single assignment is again seen to apply (§4.3.6.2).

In an imperative SAL, the programmer must still ensure that an identifier's value has been computed before its use, even though there is no ambiguity as to what value that identifier will eventually take. The programmer is not relieved of the burden of explicit control specification and for this reason most imperative languages do not enforce single assignment, foregoing it instead for the 'efficiencies' of multiple assignment (e.g., LISP cf. PURE LISP). In a dataflow SAL, the programmer specifies data dependencies as a list of definitions, like mathematical equations (hence the term *definitional languages*). The truth of a definition does not depend on its position in the program segment over which it applies, nor on the time at which it is evaluated. This freedom from interaction and lack of side effects is what allows statements to execute in parallel, constrained only by the data dependencies between them.

4.2.3 Functionality

Functional dataflow nodes (and graphs) have the value of their output tokens fully determined by their node descriptions (and connectivity) and the value of their input tokens. They operate without side effects, simply consuming inputs and generating new outputs to be sent to successor nodes. The edges of a graph must behave as pure identity functions to fully support functionality, e.g., the boundless FIFO buffers used at RMIT have this property even in the presence of token queues. Restrictions on the functionality of arcs include the common 'one token per arc rule' which gives rise to deadlocking possibilities [36]. Note that a functional graph may contain some non-functional nodes, and that improperly connected functional nodes may yield a non-functional graph.

A functional graph does not have to be determinate, and support for *nondeterministic* programming can be easily provided [13]. Most of the graphs considered in this thesis are determinate, although there is a need to be able to build arbitrary graphs using a symbolic high level language, rather than pure ITL (ch 2 and appendix D). Such graphs require the language used (DL1) to provide some nonfunctional, low level features.

4.2.4 Non-strict Evaluation

A function which is *strict* in a given argument may not begin executing until that argument is fully evaluated; an over all strict function is strict in all of its arguments. Strictness is sometimes required to guarantee correct execution, but, in general, non-strict evaluation is preferred as it leads to improved performance through increased concurrency. Non-strict evaluation increases run time asynchronism while reducing 'bursty' activity; it is supported at all levels of the RMIT dataflow model, from high level function calls to low level node evaluation (`cons`, `arb`, `sts`, etc.).

4.2.5 Well Formed and Clean Graphs

A *well formed* graph is one that returns to its initial state, or an equivalent state, after each application. 'Equivalent' means that graphs which validly store variable state information are considered well formed for the purposes of this research. Such state information can be easily held in a dataflow graph by recirculating or storing state variable tokens of the appropriate type.

Clean graphs are graphs that start and finish each application in an empty state, i.e., no priming tokens are required and no tokens are left in the matching store. These features are very important since priming tokens are difficult to support in shared subgraphs (§4.3.5), and tokens left in the matching store can be expensive in memory requirements. In addition, a clean graph is known to have terminated when there are no tokens left in the machine.

4.2.6 Reentrancy

Reentrancy is a feature of dataflow graphs which permits multiple code applications to execute in parallel. The queued static model supports reentrant filters (§4.3.8.1) and loops/tail recursions (§4.3.6) by allowing tokens from different applications to queue on arcs, leading to efficient pipelined execution. Queues must be kept in order, and nondeterministic software and hardware merges avoided, to prevent different data sets from incorrectly *scrambling*. Dynamic architectures use token tagging or code copying to make all applications entirely independent. This extracts more potential concurrency (at the cost of tagging or code copying) and allows generalised shared subgraphs, including multiple recursions (§4.3.5). Machine data paths do not have to keep token queues correctly ordered and code templates may safely use the simple nondeterministic merge. The hybrid model combines the static and dynamic modes and therefore requires safely reentrant graphs which support queueing and code sharing. This chapter shows how many dataflow code templates have been adapted to the hybrid model, e.g., by eliminating merges and unclean code.

4.2.7 Graphical Languages

Graphical languages have been proposed for dataflow machines because of the ease of representing an interconnection of nodes as a directed graph. The experience of this research indicates that a graphical language would indeed be ideal as a programming aid, but it must be remembered that a simple representation of interconnected nodes is a long way from being a high level language. In DL1 for example, a single statement can lead to the compiler planting dozens of nodes. A high level graphical language should therefore abstract complex node templates just like a textual language does.

Having drawn numerous low level machine graphs in the design of templates and debugging of programs, it is felt that a graphical form would be best as an assembly language equivalent, rather than as a concise high level language. It is certainly quicker and easier to debug machine code in a graphical environment, than in the textual machine language equivalent. Figure 4.1 illustrates the type of textual/graphical programming dichotomy being considered, where there is a one to one correspondence between the textual and graphical machine code forms and a possible relationship between the high level forms as well. All graphical code representation is currently hand drawn.

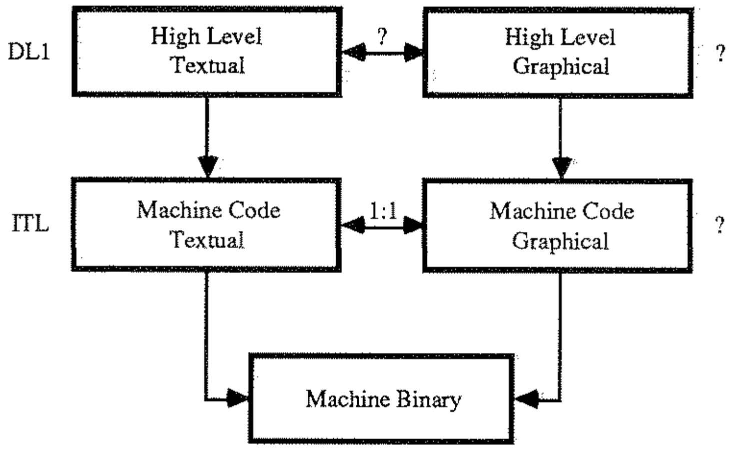


FIGURE 4.1 A textual/graphical programming dichotomy

4.3 Dataflow Language One (DL1)

DL1 was designed by C.P. Richardson to aid in his Ph.D. research at Manchester University [42]. This was after he had developed a large program in ITL, to simulate a dataflow laser range finder object recognition system, as part of his M.Sc. [41]. DL1 started out as little more than a symbolic assembly language for the FLO system, but was quickly enhanced to a high level SAL with facilities for resource management, code sharing, block structure, limited optimisation (e.g., the use of height balanced trees for multiply, addition, duplicate and other reduction/expansion operators), etc.. The syntax is based on PASCAL and Weng's TDFL [50]; a complete description can be found in [39].

DL1 is essentially a functional language, but this work has led to many revisions being made to it and the RMIT node set in order to improve this aspect. A clear distinction is now made between functional statements/expressions (**functional definition**, **if**, **subgraph call**, etc.) and nonfunctional statements/expressions (**join**, **switch**, **oldif**, etc.) [39]. Originally, little distinction was made and it was up to the programmer to add extra control statements like **protect**, and to keep careful track of the use of identifiers to ensure that programs were determinate, well formed, reentrant, etc.. Never the less, DL1 supports a necessary and sufficient complement of nonfunctional and low level operators to ensure its value as an arbitrary graph building tool and as an intermediate form for higher level languages. The rest of this chapter will include details of the enhancements made through this research.

4.3.1 Expressions and Expression Lists

In most conventional programming languages, expressions are limited to unit arity so that complex assignments involving many variables have to be done one component at a time. Procedural languages like PASCAL and C can be quite verbose as a result, although special features like complex data structures and assignment help to overcome this problem. Even so, there is still a need to declare precisely each structure to be used and assignments are restricted to variables which are related through these definitions.

Most SALs on the other hand, place no restriction on the number of components an expression can have and do not require the structure of such complex expressions (e.g., *lists, tuples, informal records*) to be predeclared. Unlimited expression lists provide for a more *unified syntax*. Functions may return one result and yet still be used like they were simple identifiers. Multiple definitions can be made in one statement by assigning an expression list to an output list. Actual parameters in a function call simply make up an expression list, which unifies the syntax for predefined functions as well as for user defined functions with variable numbers of arguments.

The original DL1 (old DL1) was very restrictive in its use of expression lists, whereas now they can be used anywhere that is sensible, including inside other expression lists. About the only place that isn't sensible is in the *factor* of an expression, although some languages even allow this, e.g. Glauert's LAPSE language which allows 'informal records' in a factor, but which allows no mathematical operations on such a factor [28]. In LAPSE, informal records are enclosed in square brackets but DL1 has no such mechanism, instead a *comma* operator is recognised which continues an expression list until no more commas are found.

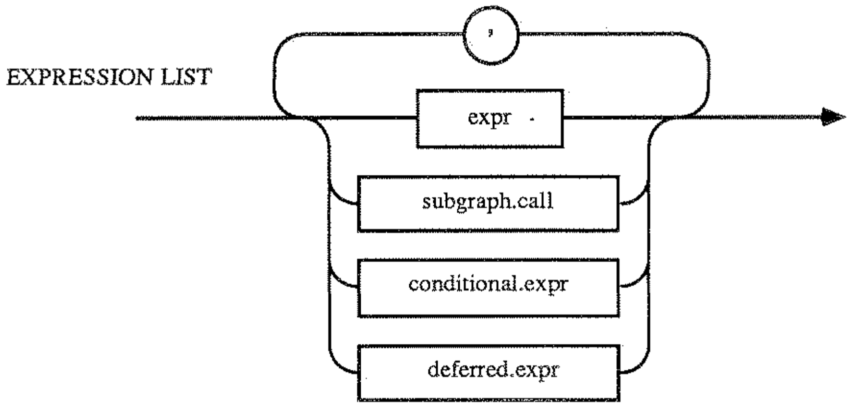


FIGURE 4.2 The new syntax of 'expression list'

The syntax diagrams of appendix B make it clear as to precisely where expression lists are allowed. Apart from this more flexible, unified use of expression lists, one other significant improvement has been made to the expression list syntax itself, figure 4.2. Old DL1 only allowed expression lists to be made up from simple (unary) expressions, but now expression lists can be assembled from expressions of any arity. This allows for a remarkable improvement in the readability and conciseness of a DL1 program, since previously one would have to assign n-ary expression lists to n individual identifiers and then use those identifiers in building up other lists. Figure 4.3 shows a simple example of this, involving a mixture of subgraph calls, conditional expressions, etc.. The components from which an expression list may now be made are: simple (unary) expressions (the trivial list), n-ary function (subgraph) outputs, n-ary conditional expressions and n-ary deferred expressions.

```
subgraph c_mul(ar, ai, br, bi: real) -> (or, oi: real);
     begin
          (* complex multiplication *)
          ar * br - ai * bi -> or;
          ar * bi + ai * br -> oi;
     end (* c_mul *);
```

Now compare

```
(* i1 and i2 are temporary complex numbers *)
if bool then xr, xi else yr, yi -> i1r, i1i;
c_mul(ar, ai, br, bi) -> i2r, i2i;
c_mul(i1r, i1i, i2r, i2i) -> zr, zi;
```

with

```
c_mul(if bool then xr, xi else yr, yi endif, c_mul(ar, ai, br, bi))
     -> zr, zi;
```

FIGURE 4.3 Using expression lists

4.3.2 Determinacy

Determinacy is a fundamental property of all but a few of the basic operators used at RMIT, so that these operators will produce the same, predictable results, under all circumstances, e.g., regardless of the order of arrival of their inputs. An arbitrary graph, made up of determinate nodes with a strictly one-to-one connectivity, is itself guaranteed to be determinate [50]. Furthermore, because the architecture models arcs as boundless FIFO queues, then data can be streamed through such a graph with no effect on its functionality or determinacy (a queue of results is produced with an order corresponding to that of the input data).

Some operators have indeterminate firing conditions, e.g., they may fire on a subset of their input arcs (but not all operators which do this are indeterminate). These include `identity (ID)`, `storage (S)` and `merge`. The latter is a special case since a physical operator does not actually exist, rather the merging is done in the communications network by directing more than one arc to the same destination point. These operators can all lead to non-reentrant graphs if not used appropriately.

Indeterminate operators can be used to build 'nondeterministic' graphs useful in many applications, e.g. real time controllers [24], committed choice logic programming [49], 'resource managers' [13], etc.. They are also useful in some code templates which are over all determinate, e.g. conditionals (§4.3.3) and protected static loops (§4.3.6.1). These templates rely on carefully laid code that creates (possibly artificial) data dependencies between tokens. Other operators that are timewise indeterminate but which are used by DL1 in a determinate manner are `first (FIR)`, `prime (PRM)` and the predefined system nodes in most elements, `input/output (e.-16...31/-32...47)`, `set-max-occurrence (e.-2)` and `exception (e.-3)`, see appendix C.

There is a group of nodes which are themselves determinate but which may introduce indeterminacy through the communications network by altering the dynamic connectivity of the graph. This effect may be observed whenever the context tags carried by tokens are altered. These nodes are `set-destination (STD)`, `yield-colour (YLC)`, `set-colour (STC)`, `arg-entry (A)`, `return-entry (R)`, `exit (E)`, `create-colour (CRC)` and `set-return-link (SRL)`. Again, the DL1 compiler uses these nodes in a manner which guarantees determinacy.

Mention was made above of a class of nodes which fire on a subset of all possible input arcs but which are never the less determinate. These are the non-strict nodes `arbitrate (ARB)`, `synchronise (SYN)`, `stream-store (STS)`, `head (HD)`, `tail (TL)`, `cons (CON)`, `protect (PRT)`, `eager-merge-control (EMC)` and `lazy-merge-control (LMC)`. These new, non-strict nodes are crucial to the efficient performance of many graphs, examples of their use and of the revisions made to improve the overall determinacy of DL1 machine graphs are given in the following sections.

4.3.3 Conditional Expressions

The nature of SALs makes it difficult to support conditional statements, because there would have to be multiple definitions for each identifier to be conditionally defined. In fact, there has to be precisely one definition for each conditional identifier, for each possible condition. This restriction ensures that each identifier is functionally defined, since after the code block has executed, each will have been assigned to exactly once. With a simple boolean predicate, two sets of definitions (the *true branch* and the *false branch*) must be chosen between. This is usually coded as a selection between two expression lists, rather than definition (assignment statement) lists, as in case (a) below.

DL1 took a unique approach to this problem by defining an **if** statement which was actually three statements in one. Only one of these forms was functional and only one (a different one) was reentrant. The three forms are described below, together with their revised versions. Note that being a statement and not an expression, **if** could not be used in expression lists and always had to be assigned to some identifier(s). Also, in new DL1, the **if** statements referred to here are still available for compatibility and comparison, as '**oldif**'.

(a) 'Selecting' **if** statement, a functional, non-reentrant form

e.g., `if bool then else x else y -> result;`

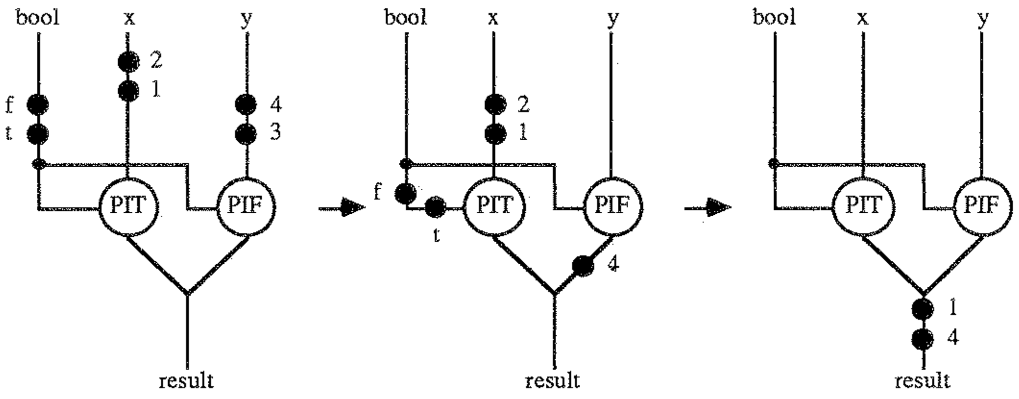


FIGURE 4.4 The non-reentrant **if else else** template

This form is functional, but non-reentrant because if more than one data set is directed at the graph, then *true (false)* results from one data set may race through the indeterminate merge with *false (true)* results from another set, figure 4.4. This race actually occurs at the merge after the two gates as all the other operators are determinate. Richardson recognised this fact and designed the **protect** statement which allows tokens on a group of arcs to be protected by tokens on another group of arcs. In the example given, we would say "**protect** bool **with** result", resulting in the graph of figure 4.5.

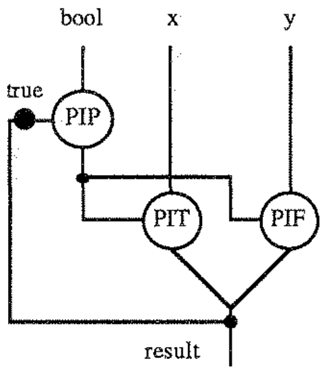
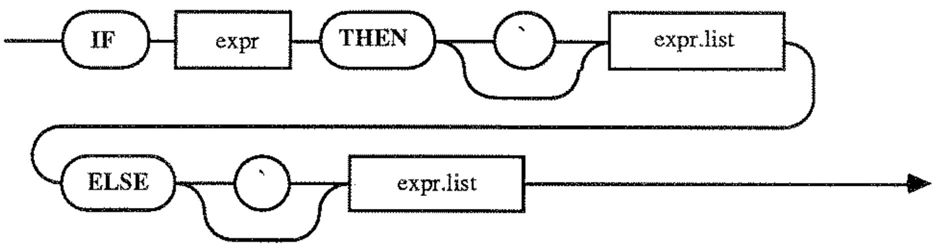


FIGURE 4.5 A reentrant form using **protect**

Unfortunately, this graph is less efficient than the unprotected version and requires a priming token, so that it cannot be used inside shared subgraphs (§4.3.5). The new form for **if** (the *conditional expression*) is shown in figure 4.6, where the syntax of the statement has changed (see also appendix B) and the `pass-if-present` (`PIP`) node, priming token and `duplicate` node are combined to form a new non-strict template called `eager-merge-control`. It is known as eager because it expects data on both gates even though one set will be discarded, and is shown in a square outline because its form is variable, depending on which compiler options are used. One option in particular, the extended node set option (compiler toggle [w+]), will expand EMC into just one node, see §4.3.4.4 and figure 4.21.

CONDITIONAL EXPRESSION



e.g., **if** bool **then** x **else** y -> result;

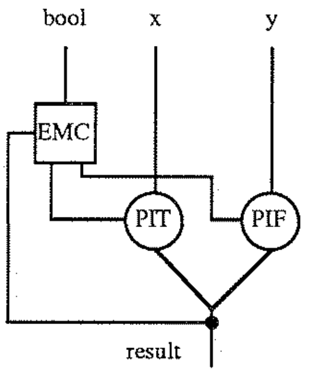


FIGURE 4.6 The revised syntax and expansion of **if**

Note that **if** actually returns an expression list, and the two input lists and the output list must all be the same size; the examples have shown lists of size one for simplicity. For expression lists with more than one component, one copy of the entire graph of figure 4.6 is planted per component. To be suitable for use inside of shared subgraphs, EMC should be well formed and clean, i.e., it should not contain any priming tokens. This interaction between priming tokens and clean graphs is often observed since a well formed graph with an initial priming token will return to that state and is therefore unclean. The optional '`' symbols are explained in §4.3.4.2.

(b) 'Merging' **if** statement, a nonfunctional, non-reentrant form

e.g., **if** bool **then either** x **else** y -> result;

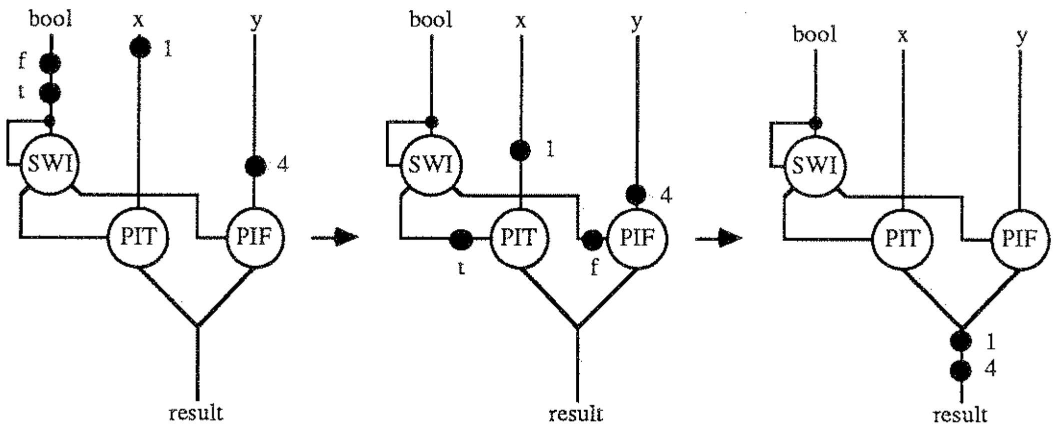


FIGURE 4.7 The lazy non-reentrant **if either else** template

This form is similar to that of (a), the difference being its *lazy* operation (§4.3.4.2). Figure 4.7 shows results being returned in the wrong order just as in the example of (a), but here the selection of the appropriate inputs should be made twice; once elsewhere in the graph, so that nothing arrives on the false input when the control is true, and once in the template itself, where the `switch` node directs the control token to the appropriate gate. A very interesting point about this template is that it is actually redundant as it stands and is effectively equivalent to a straight merge of the true and false inputs. However, when protected it does produce a reentrant graph as in part (a).

A determinate, well formed and clean version with a new syntax has also been defined for this form, figure 4.8. **Join** is actually a statement which cannot be used as part of an expression list, unlike an **if** expression, also **join** is clearly nonfunctional since it does not take one token from each and every input to produce its output. Indeed, for programs written in revised DL1, this nonfunctional form is virtually useless by itself, but it can be combined with another statement (**switch**) to give *conditional statement execution* (see (c)):

e.g.,

```
if bool then either x else y -> result;
protect bool with result;
```

becomes

```
join bool then x else y -> result;
```

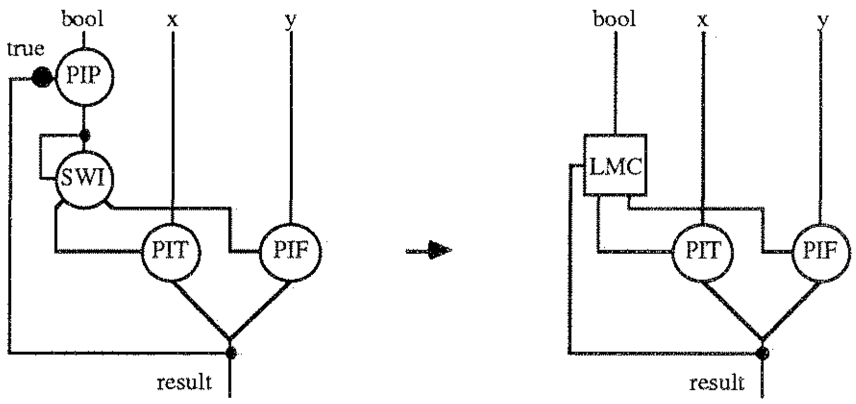


FIGURE 4.8 Protected and revised versions of **if either else** (**join**)

(c) 'Switching **if** statement, a nonfunctional, reentrant form

e.g., `if bool then x -> res1 else res2;`

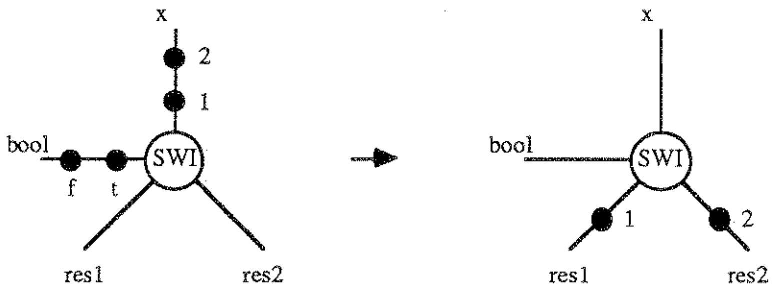


FIGURE 4.9 The switching **if** statement

The code laid for the switching **if** statement remains unchanged in the revisions to DL1, figure 4.9, however to avoid confusion it has been renamed to **switch**, with a syntax as shown in figure 4.10. Like **join**, it is a statement and not an expression. This is unavoidable in the case of **switch**, since its output syntax would make it difficult to use in an expression list. In any event, special operators like **switch**, **join**, **protect**, etc., are best left as isolated statements to improve code readability. **Switch** can be used with **join** to give conditional statement execution (e.g. program Binary_stream_sum #2, appendix A).

SWITCH STATEMENT

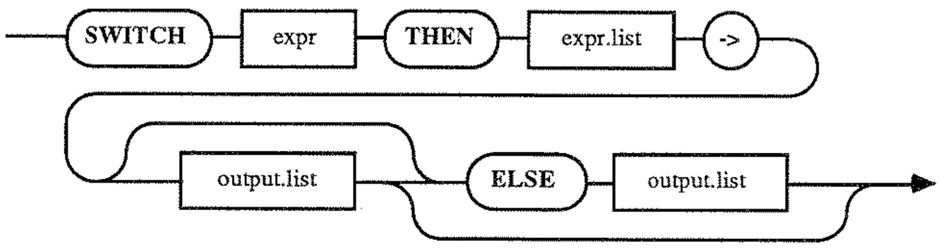


FIGURE 4.10 The syntax of switch

Either output list specification may be missing or may include the special destination **null**. In either case, the `switch` node is replaced by a `pass-if-true` or a `pass-if-false` node as appropriate, to discard tokens destined to **null**. One gate node is placed for each component of the input list.

4.3.4 Eager and Lazy Evaluation

Eager evaluation of an expression means that all inner arguments are evaluated first, then surrounding sub-expressions and so on until the outermost result has been computed. Non-strict, eager evaluation is the natural mode for data driven computation since, in dataflow, computation begins with the arrival of data tokens on input arcs. Eager evaluation can lead to increased concurrency, e.g., in conditional expressions (§4.3.4.2), but it can also lead to redundant computation and explosive activity, which can result in inefficiencies in a machine with limited resources. Also, eager evaluation is not always 'safe' and erroneous conditions, such as nonterminating recursions, can result.

Lazy evaluation takes the opposite approach, whereby expressions are not evaluated until their results are required. Since DL1 is eager by design, many applications of lazy evaluation, e.g., to provide *call by need* for function arguments [5], will not be considered in detail here, although two important aspects of lazy evaluation, i.e., excessive expression recomputation and graph *throttling* as a means of run time resource control, are briefly discussed in the following paragraphs.

Fully lazy evaluation can lead to excessive recomputation of expressions defining duplicated data items. This can be prevented in a single assignment language by using once only lazy evaluation, followed by storage and copying on request. In our dataflow system, this can be achieved by using a `first` node to detect the initial request for a value (by a nondeterministic merge of all possible requests), which fires the defining expression. The result is then placed in a `storage` node which is fired by this and further requests, in the form of addresses. A `set-destination` node sends a copy of the result to each requesting address. This scheme is not used in DL1, since it is known exactly how many copies of each result are required and it is more efficient to simply duplicate the value that many times. Also, the storage node method will not handle queues correctly, i.e., it is not reentrant.

Throttling is used in graphs which generate excessive concurrency (in bursts), which can overload machine storage space. Under lazy evaluation, computations are deferred until their results are required, these results can then be consumed straight away. This is also useful in the processing of infinite data structures, e.g., streams, which can be read as required, rather than flooding the graph with uncontrolled input. As a result, DL1 provides a deferred expression (**on**) which allows the passage of tokens to be controlled by the presence of other tokens in a damand driven manner. Other methods of throttling include limiting the number of concurrent invocations of loop bodies and shared subgraphs, etc., although care must be taken to avoid deadlocks.

4.3.4.1 Eager Evaluation in Conditionals

In DL1, conditional expressions are a source of redundant computation, because only one of the branch expressions is used and the other is discarded. However, despite this redundancy, there are still conditions under which eager evaluation is suitable for conditional expression evaluation:

- side effects *must not* be generated by the expression (list) that is eventually to be discarded
- the evaluation of the expression to be discarded *should not* lead to illegal or improper conditions like runaway recursions or passing empty streams into stream graphs
- the concurrency gained by evaluating both expressions in parallel with the predicate *should not* be lost through 'swamping' of the machine

The first condition addresses the unsafe nature of eager evaluation. It is perhaps strange, in that DL1 is a supposedly functional language without side effects, but only correct computations are guaranteed to be side effect free. The simplest example of this type of side effect is a fatal error which may have been avoided if not for eager evaluation, i.e., one that occurs in the branch to be discarded. In addition, there are certain basic operators which produce side effects, e.g., input and output nodes, where improper execution will have obviously undesirable effects on any input and output streams referred to. Some other nodes (system nodes in particular) may produce side effects and are therefore unsuitable for eager conditional evaluation.

The second condition is similar to the first in that its effects on the computation will only be felt through side effects, however, it is only a recommendation as many error situations like this are recoverable. The most important of these are simple node errors (arithmetic, type, etc.) which will not in general have adverse side effects. A section of graph which executes eagerly and generates errors, simply transmits them as ? tokens to successor nodes, so the eventual output(s) of that graph will also be errors, but most importantly, the graph will have executed functionally. Some errors which *may* not be recoverable are runaway recursions which use up too many context colours from the available pool, and passage of empty streams into stream graphs which, because of the way stream operators like **head**, **get**, etc. are implemented, might operate nonfunctionally. Some of these error conditions are still subject to change, so that for the time being we will merely point out the possibility of problems in this area.

The third condition states that although eager evaluation of a particular expression may be entirely valid, its use may generate extra (redundant) computation to the point where limited machine resources cause the final result to actually be delayed. In other words, a shortening of the critical path through a graph with a probable increase in available concurrency, at the expense of extra run time activity, does not guarantee a speed up in the production of the final result. This would not be true if such a graph could be 'perfectly' distributed on a machine with sufficient parallelism, but even if this were the case, machine utilisation (§5.7) would certainly decline.

```
if bool
then expr1(a), expr2(b)
else expr3(c), expr4(d)
-> result1, result2;
```

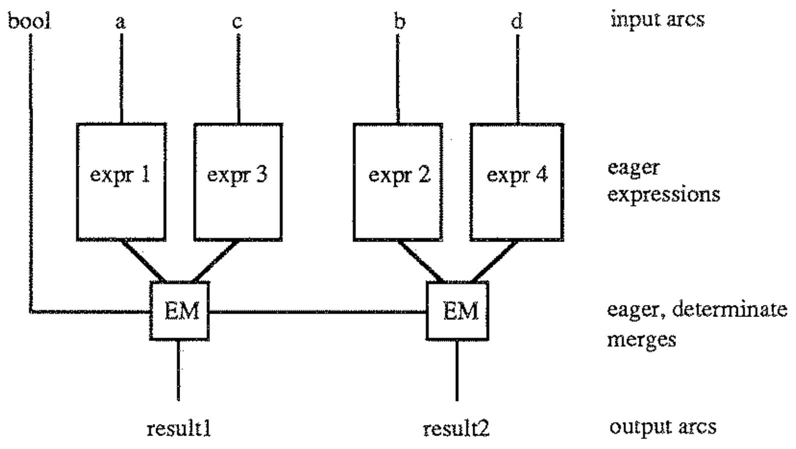


FIGURE 4.11 Eager, conditional evaluation

Figure 4.11 shows the functional definition of two identifiers (result1 and result2) using a conditional expression with eager branch expressions. The `eager-merge (EM)` macro is explained in §4.3.4.3.

4.3.4.2 Lazy Evaluation in Conditionals

In a lazy conditional expression, the predicate is evaluated first and only then is the corresponding branch expression evaluated. In the dataflow graph, branch expressions are delayed by *gating* their input arcs. In addition, tokens destined to the inputs of the unselected lazy expressions must be *killed*, figure 4.12. The meaning of the "' annotations in conditional expressions is now clear, an expression so annotated is to be evaluated lazily. The annotation applies to the entire branch expression list in the current implementation. Figure 4.12 shows a conditional expression with lazy branches.

```
if bool
then ` expr1(a), expr2(b)
else ` expr3(c), expr4(d)
-> result1, result2;
```

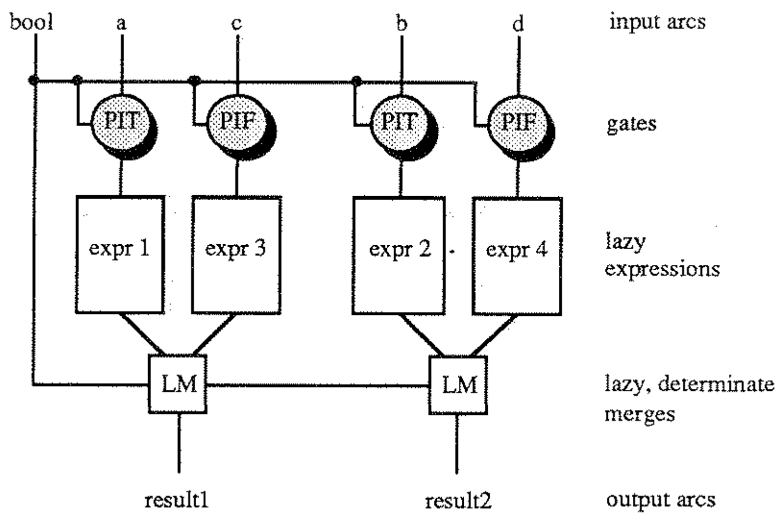
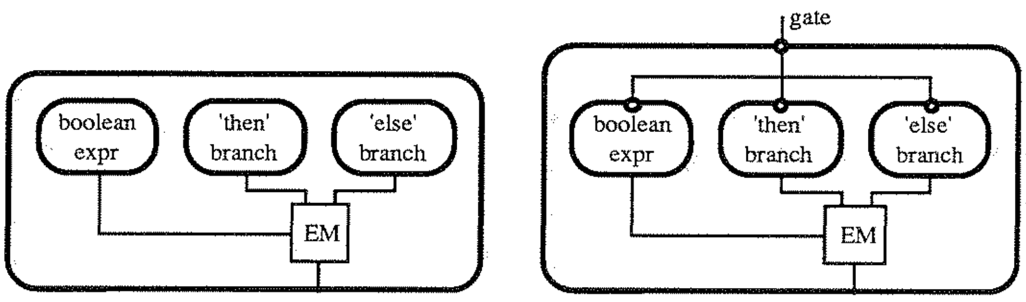


FIGURE 4.12 Lazy, conditional evaluation

4.3.4.3 Eager, Lazy and Hybrid Code Templates

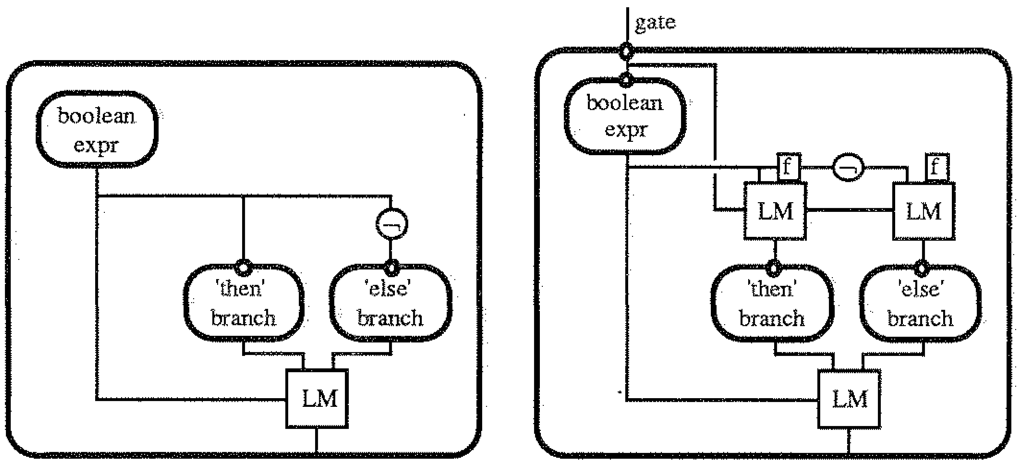
There is no reason why the branches of a conditional expression should both be either eager or lazy, thus it is valid to use lazy annotation on only one branch. As conditionals are themselves expressions and can therefore be used in the branches of other conditionals, according to the new syntax of expression lists, then we may have *lazy conditional expressions*, as shown by the template of figure 4.13. The examples of §4.3.4.1 and §4.3.4.2 are in fact both eager conditional expressions, the second of which has lazy branches. This section will show the templates used for all possible cases and give the expansions for the macro operators used in these templates.



e.g., **if** a > 0 **then** sqrt(a) **else** sqrt(-a)   e.g., `**if** ok **then** yes **else** no

FIGURE 4.13 Eager and lazy conditional expressions

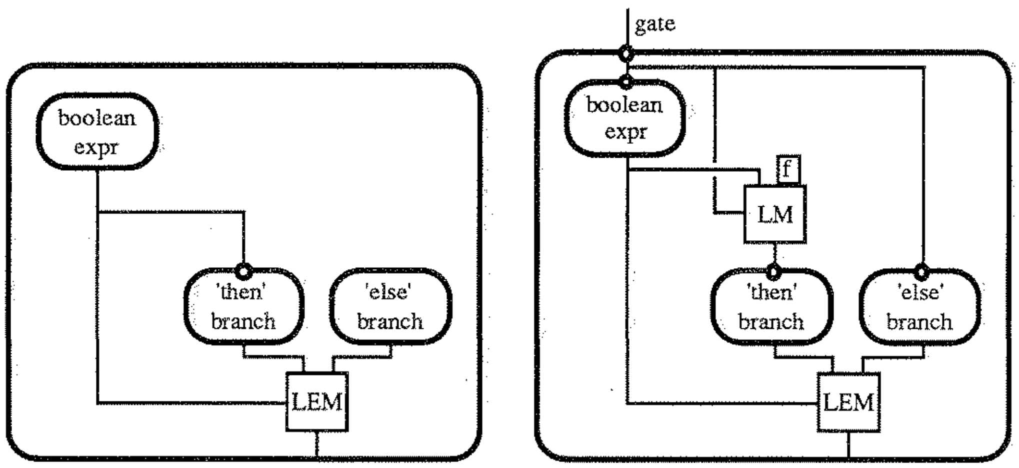
Templates are shown in heavy, rounded rectangles and may refer to each other, possibly recursively. All examples are shown with only one element in the true and false branches, merges and gates being simply repeated for branches with more than one component. Gates arise from conditionals using lazy annotation and are shown entering gated templates through circles at the top. In the case of a gated (lazy) conditional using lazy annotations itself (nested lazy evaluation), the second gate is derived from the first, together with the predicate, sometimes requiring a logic inversion, see figures 4.14, 4.15 and 4.16.



e.g., **if** a > 0 **then** \`sqrt(a) **else** \`sqrt(-a) e.g., \`**if** ok **then** \`yes **else** \`no

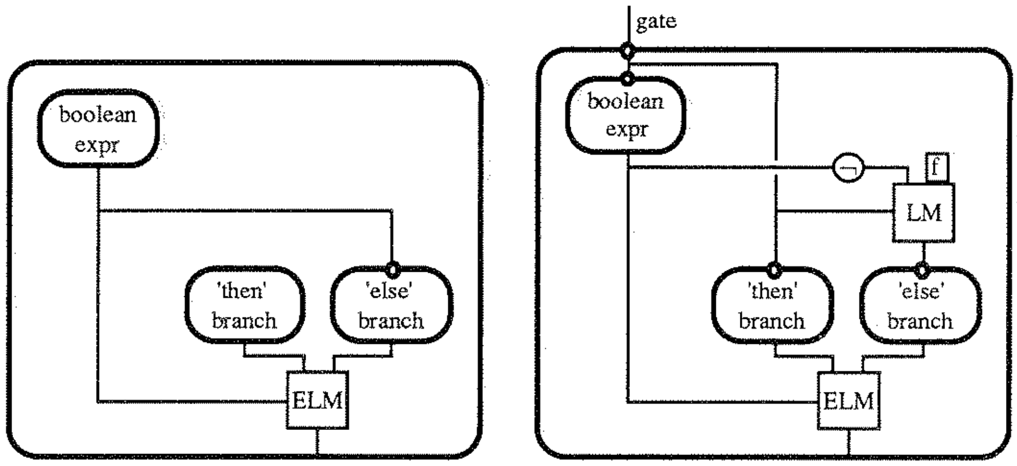
FIGURE 4.14 Eager and lazy conditional lazy expressions

The LEM and ELM macros referred to in some of these templates are *hybrid merges* which always expect a token on the eager input, but only on the lazy input when the predicate has the appropriate value.



e.g., **if** a > 0 **then** \`sqrt(a) **else** sqrt(-a) e.g., \`**if** ok **then** \`yes **else** no

FIGURE 4.15 Eager and lazy conditional lazy/eager expressions



e.g., **if** a > 0 **then** sqrt(a) **else** \`sqrt(-a) e.g., \`**if** ok **then** yes **else** \`no

FIGURE 4.16 Eager and lazy conditional eager/lazy expressions

In some templates, LM is shown with a literal 'boolean false' on one input. Literals like this are suitable for both eager and lazy evaluation, since they do not leave a token on an arc when not selected, i.e., they are clean.

4.3.4.4 Eager, Lazy and Hybrid Merge Expansions

The expansions of the merge macros depend heavily on compiler options set in the source program. The options which effect the expansions are *determinate code* ([d±]), which influences code reentrancy, and *extended node set* ([w±]), see also appendix B. These options were originally introduced for testing purposes and were retained to maintain compatibiltiy with older versions of the compiler and source programs. Now they are retained as options only because the node set has not been finalised and because they are very convenient in the study of how different graph expansions affect static and dynamic performance (ch 5).

The determinate code option currently affects only those programs which use the original indeterminate **if else else** or **if either else** (now renamed to **oldif**) statements. If [d+] is selected, then DL1 will automatically 'protect' all **oldif** statements, saving the programmer the responsibility of adding **protect** statements to guarantee reentrancy. In future, it will be possible to override the automatic protection built into the new merge macros by using [d-], as not all graphs must be reentrant.

It is critical that expansions do not include priming tokens so that the templates can be used in shared subgraphs and to a lesser extent because priming tokens represent unclean code. The extended node set has been designed in conjunction with the matching functions of chapter 3 with precisely this in mind. In addition, one extra node, `first (FIR)`, has been added to the basic node set so that some sort of expansion, albeit far from ideal, is available for shared subgraphs when using that node set.

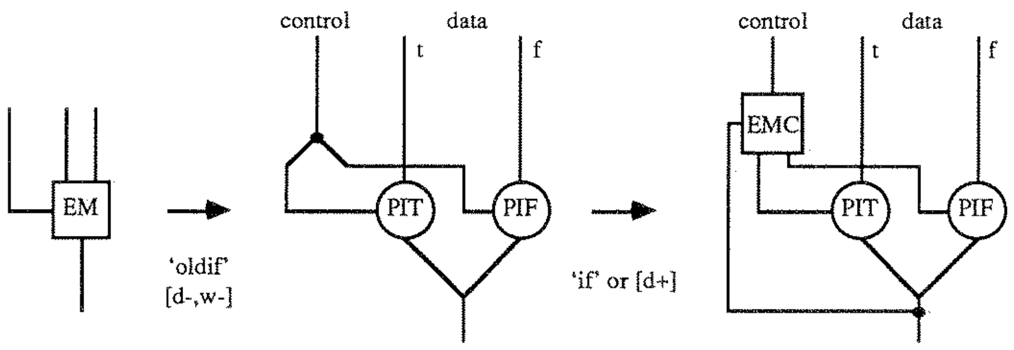


FIGURE 4.17 Expansion of the `eager-merge (EM)` macro

Figures 4.17 and 4.18 show the expansions of the basic EM and LM macros. 'Protection' is used in all cases except for the **oldif** statement with the [d-] option, thus ensuring reentrancy.

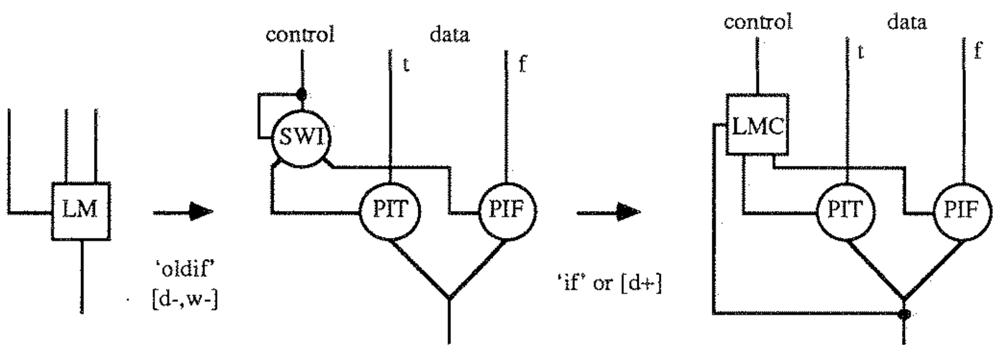


FIGURE 4.18 Expansion of the `lazy-merge (LM)` macro

Figures 4.19 and 4.20 show the expansions of the hybrid eager/lazy and lazy/eager merges. Note that an optimisation is available here in that new nodes could be defined to absorb the `PIF(false)` and `PIT(true)` nodes back into the output of the preceding `DUP` or `EMC` node. Such nodes could be named `ELD`, `LED`, `ELC` and `LEC` for `eager-lazy-duplicate`, `lazy-eager-duplicate`, `eager-lazy-control` and `lazy-eager-control` respectively.

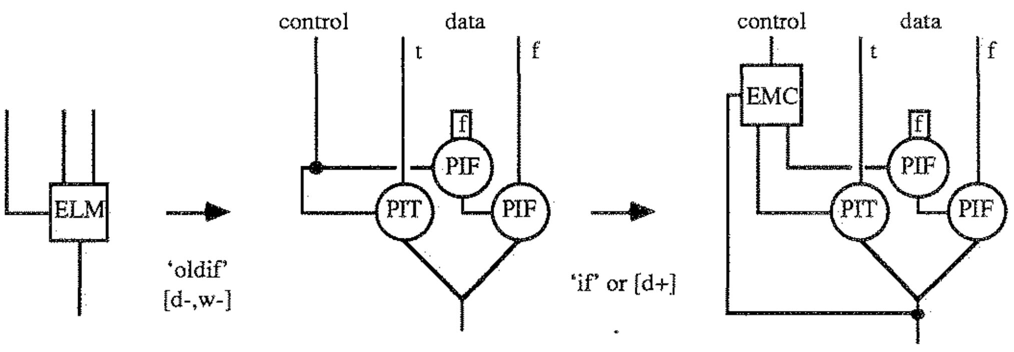


FIGURE 4.19 Expansion of the `eager-lazy-merge (ELM)` macro

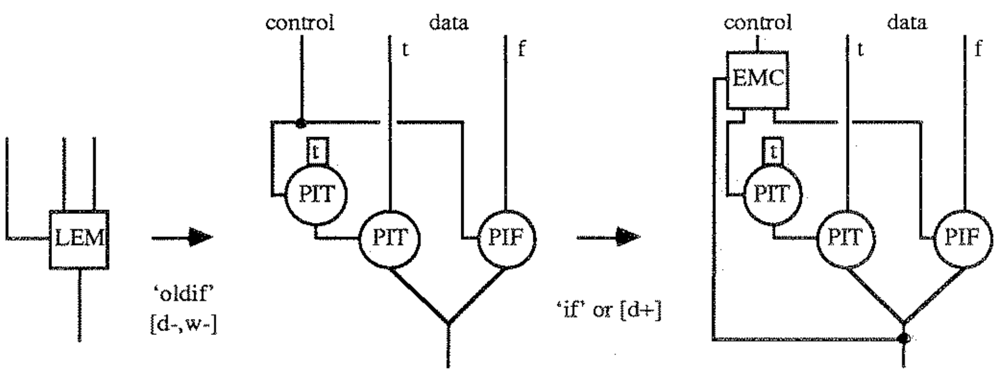


FIGURE 4.20 Expansion of the `lazy-eager-merge (LEM)` macro

The expansions of the merge control macros (figures 4.21 and 4.22) clearly show the relationship between protection, priming tokens and `DUP/EMC (SWI/LMC)` nodes. In particular, an `EMC` node is equivalent to a primed, protected `DUP`, and a `LMC` node is equivalent to a primed, protected `DUP` plus `switch`. Also of special interest is the expansion used in the case of shared, unextended code. In this case, a `first` node is used to simulate the action of the priming tokens at the expense of one token being left in the matching unit for each different context (colour) processed, i.e., an unclean graph is produced.

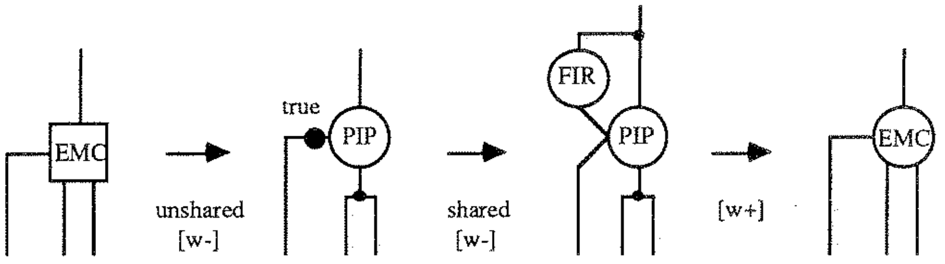


FIGURE 4.21 Expansion of the `eager-merge-control (EMC)` macro

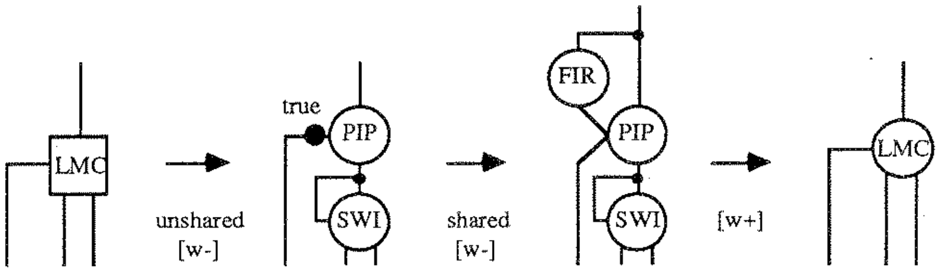


FIGURE 4.22 Expansion of the `lazy-merge-control (LMC)` macro

4.3.5 Subgraphs

Subgraphs are the DL1 equivalent of conventional procedures, functions and macros. Like other functions they should maintain the order of queues so that outputs are returned in the same order as inputs are received (this effect can be simulated without requiring the subgraph itself to support queueing, e.g. by sorting tagged outputs). The two types of subgraph provided by DL1 are the *unshared* and *shared* subgraph. This research has led to the correction of some faults in the shared subgraph calling mechanism and some syntactic improvements which allow functions with one output to be declared without an output list, e.g.,

```
        subgraph fred(a,b: integer): integer;
cf.     subgraph fred(a,b: integer) -> (out: integer);
```

and the general expression list optimisation which allows subgraph calls to be used directly in building expression lists (§4.3.1).

Unshared subgraphs are block structured macros which are copied wherever they are called. They do not allow code sharing, including recursion, except for that obtained by streaming token queues through them (§4.3.8). They are the default form used by DL1 since they provide for optimal run time performance, at the cost of expanded graph size. An analysis of the performance of unshared subgraphs can be found in ch 5 along with refs [42, 41, 2].

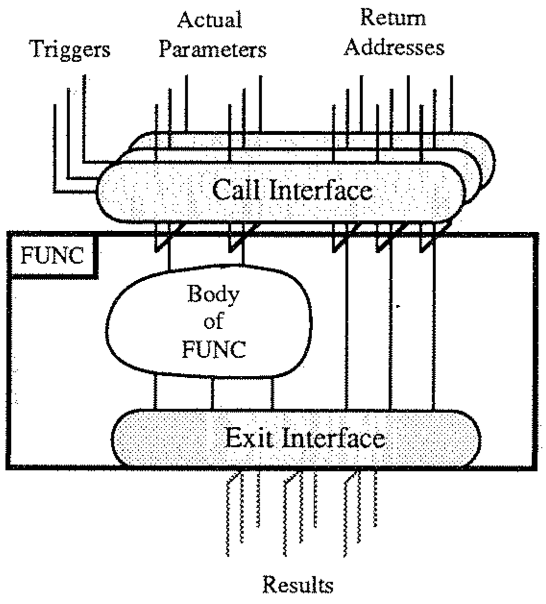


FIGURE 4.23 The general data driven call mechanism for shared subgraphs

Shared subgraphs are implemented using a *calling mechanism*, which *tags* (or labels) argument tokens and untags and routes result tokens to their respective return addresses, figure 4.23. A *trigger* may be required to generate the tag if one of the actual parameters is not used for this; any suitable identifier can be used, or it can be derived from a single 'global' trigger which is passed through all subgraph calls. Having passed through the call interface, uniquely tagged tokens from all calling graphs are merged at the entry to the shared subgraph, thus providing the desired code sharing, but some concurrency may be lost if all shared activations are executed in the same machine region (§4.3.7, [7]). If the tags are not unique, then tokens involved in different calls to a shared subgraph, but with identical tags, may interact or *scramble*. The tagging scheme used by old DL1 does not guarantee the required uniqueness (§4.3.5.1). Note that each call interface forms a part of the calling graph, but the exit interface, through which all results must be returned, forms part of the called graph.

A problem exists with the priming of shared subgraphs, since when tags are generated, there is no simple mechanism for generating the priming tokens. The problem is virtually unique to static/queued dataflow systems, because if there are no queues, then priming tokens may be generated along with the instantiation tag (the priming token will not be created more than once because the trigger will occur only once). Note however that even a purely dynamic machine may generate 'queues', as long as there is never two tokens with the same tag on an arc at the same time. There is also a determinacy issue here, since a priming token generated at run time may be involved in an indeterminate race with other run time tokens destined for the same arc; this can also happen when priming tokens are sent in with the graph description at load time if the graph is allowed to start execution before all the priming tokens have arrived. The general solution to this problem is to only use priming tokens where no races can occur, i.e., no tokens can arrive on a primed arc until the priming token has. An example of the problem and how it is overcome can be seen in figures 4.39, 4.40 and 4.41. In these diagrams, the template for **head** of stream is logically primed, first with an actual token, second with a `first/pass-if-present` pair and last with a `prime` node. The second case is the dangerous one and the primed arc must be protected with a second pip node as shown in figure 4.40. A straight merge of the `PIP(true)` and `CPT([])` outputs would almost certainly fail due to the priming token arriving after its own trigger. This could also happen to the graph of figure 4.39 if the program can start executing before the priming token arrives. Figure 4.41 is safe because the `prime` node has the protection of 4.40 built into it, i.e., it always emits the priming literal first. The templates for **cons** show safe cases which are determinate; no race can occur because the priming token is needed before any run time tokens can be directed to the primed arc. Precise details of `first` and `prime` were given in §3.5-6.

4.3.5.1 The Copy Number Mechanism

DL1 uses a low level *computed copy number mechanism*, provided by the original FLO node set. Figure 4.24 shows this type of shared subgraph call in machine graph form.

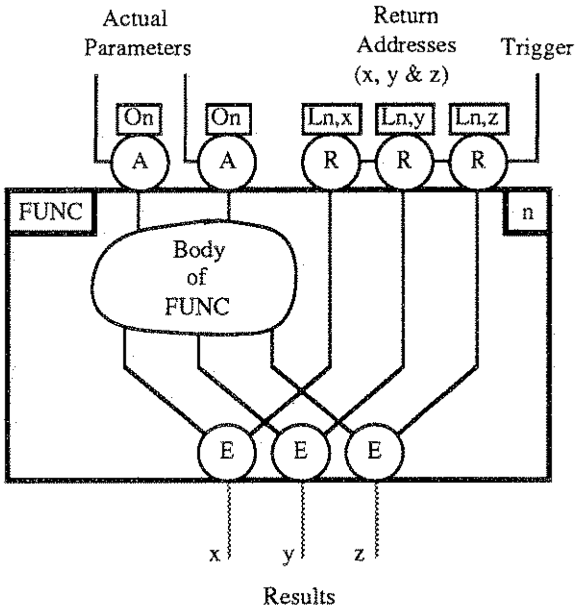


FIGURE 4.24 Template for a shared subgraph call using copy numbers

Shared subgraph applications are characterised by a *copy number* which is computed by the call interface from the incoming token's copy number and a literal *occurrence* datum assigned to each interface. DL1 programs are statically divided into *levels* and *occurrences*, corresponding to their shared subgraph call structure. Every nested call to a shared subgraph increases the level by one. Every call to a given shared subgraph from a particular level is given a unique occurrence number, OCC, starting at one. A system constant, MAXOCC, defines the maximum number of calls allowed to any particular shared subgraph at each level. The copy number at the outermost textual level is zero and is optimised by allowing tokens with a zero copy number to actually not carry a copy field at all.

On entry to a shared subgraph, a new copy number is computed by:

```
Newcopy = Oldcopy x MAXOCC + OCC;
```

and on exit, the old copy number is restored using:

```
Oldcopy = (Newcopy - OCC) div MAXOCC;
```

In figure 4.24, special nodes `A (argument-entry)`, `R (return-entry)` and `E (exit)` perform the copy number computations described above. The `A` node carries a literal *occurrence* token which has the values of `OCC` and `MAXOCC`. The `R` node carries a literal *link* token which also carries an occurrence field as well as the return address (element, node, input point) associated with its particular result. The *link* token is sent to the E node, with the new copy number as a tag, where it is used to untag and route the corresponding result.

Some interesting observations can be made on this tagging mechanism:

- Copy numbers are wasted whenever MAXOCC exceeds the actual number of shared subgraph calls at any given level. This is nearly always the case since MAXOCC must be large enough to cater for the greatest possible value of OCC, i.e. MAXOCC ≥ max(OCC). A special constant, TOPOCC, allows MAXOCC to be exceeded on the first level by offsetting the values of OCC in the top level call interfaces. This adjustment cannot be used at lower levels, or scrambling may occur.
- The subtraction of OCC in the computation of Oldcopy is unnecessary, since integer division by MAXOCC will discard any remainder less than MAXOCC.
- Important information about the history of a token is contained in its copy number. In particular, the values of OCC can be determined for every level back to the top level of the graph, where the copy number is always zero.
- The copy numbers correspond to the branches of an n-ary tree where n = MAXOCC. This tree is the parallel machine's equivalent of a *stack*. The copy number formulae are the equivalent of *pushing* and *popping* this stack.
- This mechanism can be supported by a distributed environment, one of the key points behind its conception [42, 24, 7].
- The compiler cannot *lay* a call interface until the call level is known. Thus code generation for call interfaces must be delayed until a call can be traced to the bottom or zero level.

The copy number tree referred to above is designed to correspond to the static call structure of the source program. It is assumed sufficient that the tree connectivity be a super-set of this static structure, but it will be shown that this is not sufficient, because no provision is made for the dynamic connectivity introduced by (mutual) recursion. A recursive subgraph exists at many levels equal to and greater than the level of its first instantiation, in particular, a self recursive subgraph exists at all such levels.

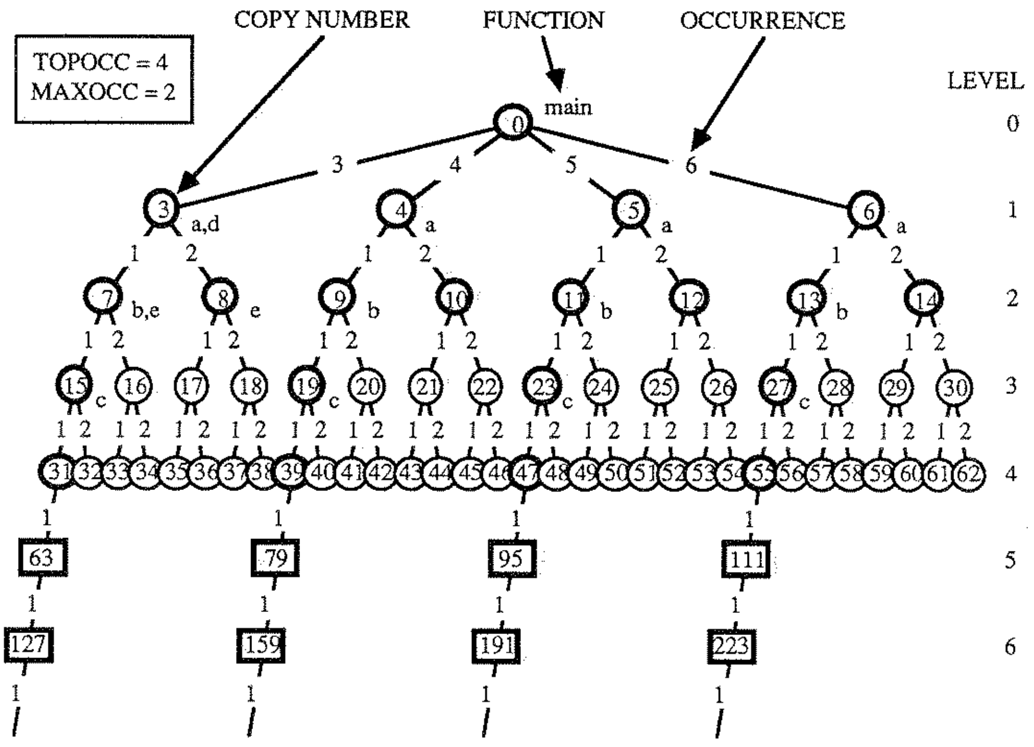


FIGURE 4.25 A copy number tree showing the mapping of the program structure of figure 4.26

Figure 4.25 shows a copy number tree with TOPOCC=4 and MAXOCC=2, notice how the values of OCC on the top level are offset in this case to ensure that copy numbers are unique between levels 1 and 2. The values of OCC and COPY shown will be the same for any compilation with these values of TOPOCC and MAXOCC, but different graphs will map onto this static structure differently. The diagram also makes reference to five functions a,b,c,d and e and shows how the graph structure of figure 4.26 maps onto this copy number tree. Figure 4.26 shows this mapping in the opposite direction.

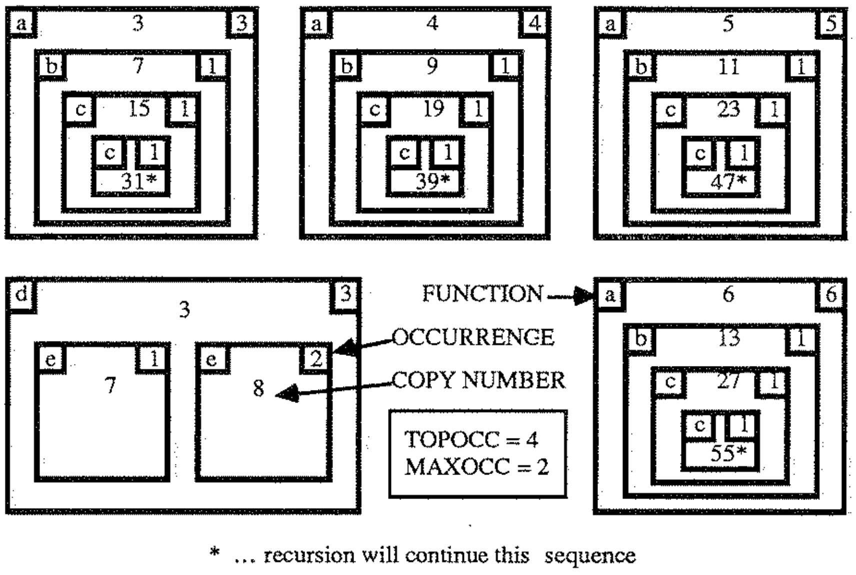


FIGURE 4.26 A program structure corresponding to the mapping shown in figure 4.25

This first example shows a *safe* case in that no scrambling can occur, i.e., in figure 4.25 there are no instances of any function occurring twice with the same copy number and correspondingly, in figure 4.26, there is no copy number appearing twice in the instantiation of any function. Despite being safe, this example still exhibits many problems associated with the computed copy number mechanism. The function c is only singly recursive and generates a wasteful copy number sequence, MAXOCC can't be set to 1 because there are two calls to e. With a 32 bit tag field, c can recurse at most $\log_{MAXOCC}(2^{32}) = 32$ times before exhausting the available tag pool. In general, copy number trees are exponentially wasteful of the available tag space whenever max(OCC) < MAXOCC. To overcome this problem, the original FLO specification calls for extendible tag fields, but this is an unsatisfactory solution given that a 32 bit field should enable 2^{32} concurrent instantiations of *each* function. Even a 16 bit tag field would be adequate for most practical programs if one could make use of every available tag for each function.

```
[o1,1] (* MAXOCC, TOPOCC set to 1 *)

        shared subgraph a(i,j: integer): integer;
        begin
                (* a(i,j) reproduces i by recursion, then adds j to it *)
                if i > 1 then `a(i-1,j)+1 else 1+j -> a;
        end;

        shared subgraph b(i,j: integer): integer;
                shared subgraph c(i,j: integer): integer;
                begin
                        a(i,j) -> c;
                end;
        begin
                c(i,j) -> b;
        end;

        begin (* main *)
                prime begin
                        4 -> x1;
                        400 -> x2;
                        6 -> y1;
                        (* 600 -> y2 *);
                        end;
                (* b(), deprived of its second arg, should produce no result *)
                a(x1,x2) -> (1,-32,0); (* send result to console *)
                b(y1,y2) -> (1,-32,0);
        end.
```

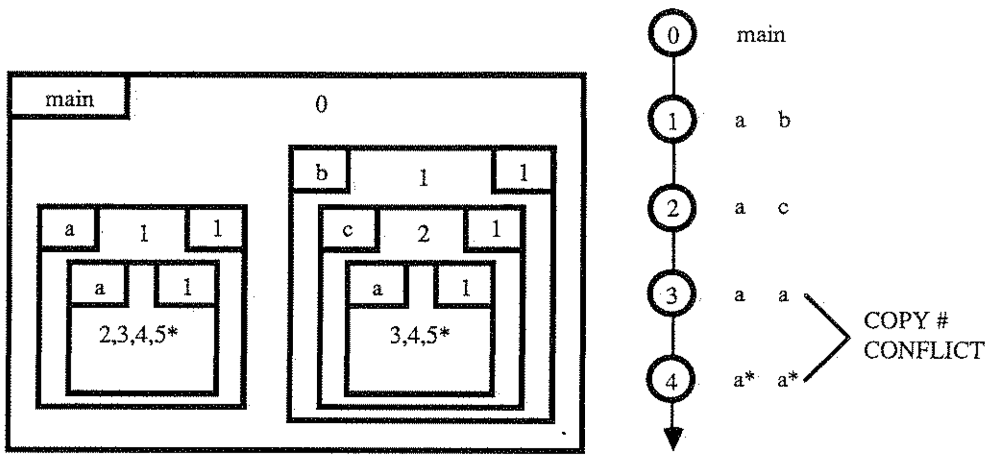


FIGURE 4.27 An unsafe program showing how conflicting copy numbers arise

Figure 4.27 shows a program that is not safe, the program and copy number diagrams clearly show how several instances of the recursive function `a` are created with the same copy numbers. It is therefore possible for tokens involved in logically distinct calls to `a` to scramble, with the result that the program shown produces a result of `406` for `b`, instead of the correct `404` for `a`. This occurs because the actual parameter `x2` of the call `a(x1,x2)` is being usurped by the call `b(y1,y2)`. This could not happen if procedure application was *strict* and *atomic* (thus requiring *all* arguments to be evaluated and the passing of parameters to occur *indivisibly*), but the non-strict nature of DL1 machine graphs allows the call to `b()` to proceed all the way down to where its nested call to `a` requires the second argument, `j`, to appear. Of course, this `j` never does arrive, but the `j` of the logically distinct top level call to `a(x1,x2)` does, with the same copy number, so that an improper match occurs.

One way to guarantee unique copy numbers would be to give a different value of `OCC` to each and every call of a shared subgraph regardless of the static call level. Unfortunately this method still does not overcome the problem of copy numbers being wasted, unless all subgraphs have the same number of occurrences. Also note that the copy number mechanism fails in the case of functions being passed as parameters, or being dynamically instantiated in any other way, since the mechanism relies on the compiler being able to generate a suitable value for `OCC`. Given these problems and a desire for safe, non-strict procedure application, a new method that guarantees unique tags for logically distinct applications has been designed for future language implementations. Unfortunately, it is not without disadvantages of its own.

4.3.5.2 The Unique Colour Mechanism

Returning to the general call interface of figure 4.23, a mechanism is required that will append a unique tag to each argument, where the actual value of the tag may or may not be of any special significance. In the previous discussion on computed tags it was pointed out that the value of the tag may contain important information on a token's history and this is seen as an advantage of that method, for example, it may help in debugging a graph by aiding in the interpretation of a matching store dump or run time inspection. Never the less, all that is really required of a tagging scheme is a guarantee of uniqueness and the ability to restore the caller's tag when a function returns results. In addition, in the presence of queues, a shared subgraph should return results in the correct order, so that it must be reentrant for a given calling colour. A procedure that returns no results, i.e., one that is redundant or one that operates by producing side effects, need not restore the tag of the calling graph at all.

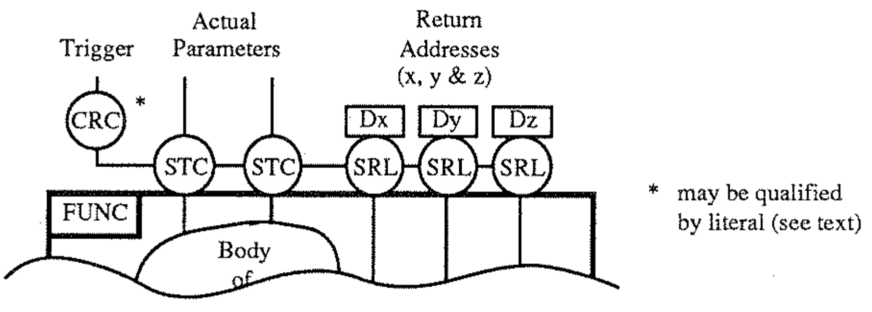
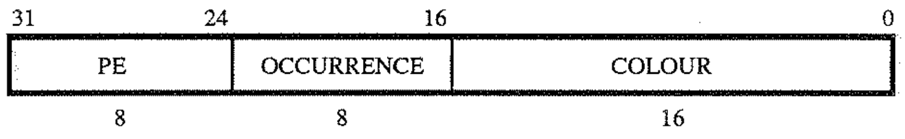

Trigger
Actual Parameters
Return Addresses (x, y & z)
CRC
*
STC
STC
Dx
Dy
Dz
SRL
SRL
SRL
FUNC
Body of
* may be qualified by literal (see text)

FIGURE 4.28 Template for a shared subgraph call using unique colours

The unique colour scheme, figure 4.28, uses a new call interface using `create-colour (CRC)`, `set-colour (STC)` and `set-return-link (SRL)` nodes, note that `set-colour` is identical to `set-copy` and shares the same mnemonic, see appendix C. The exit interface is the same as that shown in figure 4.24. The CRC node generates a unique colour, which characterises this particular call and is appended to the actual argument and return address tokens by the STC and SRL nodes respectively. The SRL node takes a literal destination token and converts it to an *environment* token by adding an <oldcolour> field to the destination token. The `exit` node uses this field to restore the caller's colour to the return tokens. The behaviour of `exit` depends on the type of the token on input 1, since it is also used in the copy number call mechanism outlined previously.

The implementation of CRC is critical for it must be usable in a distributed environment, e.g. it can not use a global counter. The current implementation uses a separate counter for each processing element which increments with every access to CRC, thus ensuring a unique colour sequence within a PE. To ensure uniqueness between colours from different PEs, the colour is structured as in figure 4.29. The <PE> field indicates the originating processing element number (0 to MaxPE, where MaxPE ≤ 255) and prevents tokens tagged with colours from distinct PEs from scrambling as they merge into a shared graph region.



PE	The originating processing element
OCCURRENCE	An optional field, further qualifying the tag origin
COLOUR	The actual colour of this tag

FIGURE 4.29 The structure of a 'colour' tag

The <occurrence> field in the colour tag is set by an optional *occurrence* literal on the CRC node itself. This field is an attempt to add to the meaning of the colour tag by helping to identify its origin. Its precise use is undefined in the current implementation and the colour counter is free to overflow into this filed at present. It could also be used to hold an 'index' field for data structures, as in the Manchester design [29], although the limitations on data structure nesting implied by this field are just as undesirable as those on iterations due to the Manchester *<Iteration Level>* field (§4.3.7). The RMIT system currently handles 2^{24} concurrent calls to any shared subgraph from any given PE (2^{32} in all), a counter overflow produces an appropriate error token.

Points to note:

- Colour allocation on a per PE basis may help in the study of dynamic graph partitioning (i.e. run time balance). The colour itself represents the total number of calls made (as given by the <colour> subfield) from interfaces within that PE.
- The <PE> field requires each PE to know its own identity, this may have implications for the design of dynamically reconfigurable machines. A similar requirement is placed on machine design by the use of a local queue to minimise network token traffic.
- The structuring of the colour tag to include a <PE> subfield, although necessary, may limit the effective tag pool size. This is because it is not guaranteed to have an equal distribution of calls to each and every shared subgraph among each and every PE, although dynamic graph unravelling is an attempt to do just that (§4.3.7).
- Colours may be further structured to indicate the name of the calling graph, the particular interface, etc.. This would necessitate allocating counters on a 'per interface' basis and could further reduce the effective tag pool size (see above).
- Like the copy number mechanism, this mechanism can be supported in a distributed environment.
- Unlike the copy number mechanism, the compiler can now lay a call interface as soon as a call is made.

The precise operation of the CRC node is under review, so that an alternative scheme which avoids the overflow problem may be implemented in the future. Such a scheme could be to allocate colours from a free list, or pool of available colours, and to return colours to this pool when a call is complete. A *signalling* scheme as described in [48] can be used to detect the termination of a call at the cost of increased graph complexity. Also, the generated colour should ideally be reused for each further call with the same calling colour to ensure that queues are kept in order, but other methods can be used for this, e.g. protection, see also §5.9.3. Tagging schemes similar to the one described here can also be found in [28] and [44].

4.3.6 Iteration and Tail Recursion

DL1 provides no direct syntactic support for iterative functions, instead they may be expressed as either static or dynamic tail recursions. In a tail recursion, the function either returns a result (at termination), or calls itself and returns the result of that call, with no post-processing. A static tail recursion forces a time dependency between tokens in all cycles, achieved by queueing; but in a dynamic tail recursion, the bodies of all cycles are independent of each other and can execute in a different context (unfolding). A tail recursion, static or dynamic, must be lazy with respect to its arguments, to prevent run away and to ensure clean execution.

A scheme for translating ***while*** and ***for*** loops into their tail recursive counterparts is given by Nikhil et al [37]. The idea behind this translation is to split the loop body into local and result (new) bindings. In a clean DL1 loop, all tokens in the local bindings are either used to compute the results or are discarded, local bindings/tokens are therefore referred to as *loop temporaries*. Results are passed to the next cycle through a nondeterministic merge in a static iteration, or through the argument list in a dynamic tail recursion. The originating call must initialise all result bindings. Loop invariants must be passed in the argument list with loop variables, as no '*loop constant*' storage is provided by the RMIT hardware. When the termination condition is satisfied, the last cycle may return the result(s) directly to the calling context (tail optimisation).

4.3.6.1 Static Iteration

The classical approach to iteration treats each cycle as occurring in the same context, i.e., no unfolding. In a von Neumann machine the program flow jumps back to the entry point of a loop at the start of each new cycle, and has access to static, *in place* variables in an efficient, space saving manner. There is no need to push a data stack with each new iteration and the final result may be computed and returned directly by the terminating cycle. A similar strategy can be used in a dataflow graph, provided that the loop body code is reentrant. The jump can be achieved by a simple merge of initial call tokens and result tokens, which continue to circulate through the loop until termination. The loop is used in line, so no result distribution or tagging is required. Figure 4.30 shows a static dataflow translation for loops of the form:–

```
initialisation(...)
while condition(...) do
        loop-body(...)
return result-exp(...)
```

A ***repeat*** loop simply places the termination condition and switch after the loop body.

As shown by shading in figure 4.30, some parts of the translation are not always present:–

- There may or may not be any recirculating loop invariant data. Such data implies a level of inefficiency in this model, since it must be copied several times.
- The overall iteration is non-reentrant, and protection is required before the nondeterministic merge if the code is to be pipelined. This protection can be reset by a false condition (e.g., a token from the false switch output). The protected merge arrangement is more efficient than a primed **join** construct in this case, although the latter is superior in most other cases because of its local protection.
- The result expression evaluates in the same context as the calling code and therefore may access free variables in the surrounding graph. This may not be true in an unfolded iteration (§4.3.6.2).

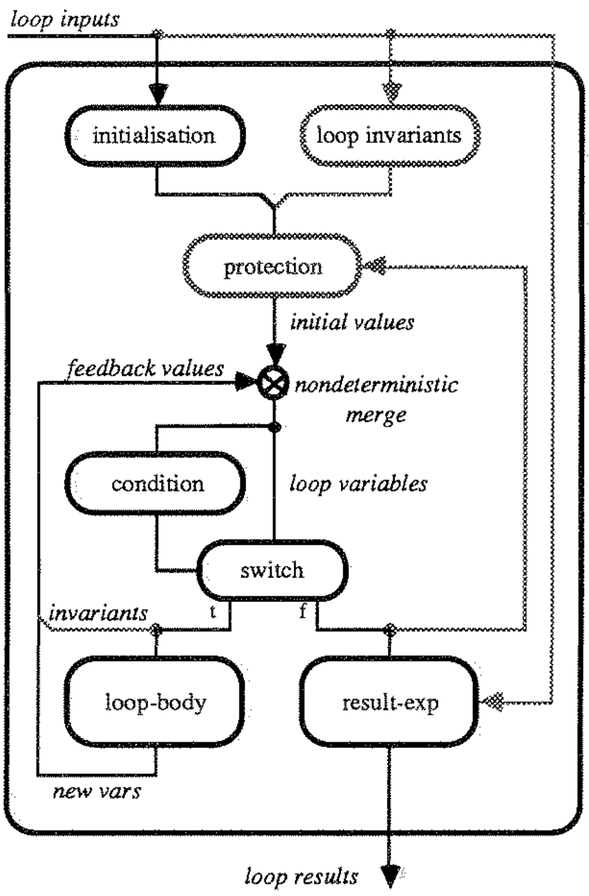


FIGURE 4.30 Static dataflow *while* loop translation

```
(*
   INT and T2 are constants,
   close_range is a free variable in the surrounding graph
*)
... -> close_range;
..., ... -> Itheta, Iphi;                  (* initialisation *)

(* merge initial and 'new' values to give loop variables *)
merge (Itheta, Ntheta) -> theta;          (* indeterminate *)
merge (Iphi, Nphi) -> phi;

(* evaluate condition, and 'switch' variables *)
switch laser_miss(theta, phi) and theta < T2
then theta, phi, true
-> Btheta, Bphi, null                     (* loop body inputs *)
else Ftheta, Fphi, done;                  (* final values *)

(* loop body *)
laser_plot(Btheta, Bphi) -> null;         (* plot all 'miss points' *)
Btheta + INT -> Ntheta;                   (* new theta (variable) *)
Bphi -> Nphi;                             (* new phi (invariant) *)

(* optional protection for pipelined use *)
protect theta, phi with done;             (* ensures reetrancy (optional) *)

(* result expression *)
Ftheta, laser(Ftheta, Fphi) < close_range -> ... , ...;   (* loop outputs *)
```

FIGURE 4.31 DL1 description of a static *while* loop

Figure 4.31 shows a DL1 ***while*** loop which moves a laser scanner in the `theta` direction until it strikes an object or reaches a limit on `theta`. It plots all 'miss points' in the scan and returns the final value of `theta` and a boolean which indicates whether an object found was closer than a certain range. The lack of syntactic support for static loops is immediately obvious from this figure, the program must be layed out carefully if the final code is to be well formed and clean. This is true of any graph which uses nonfunctional statements like **switch** and **merge**.

4.3.6.2 Dynamic Tail Recursion and Loop Unfolding

The laser scanner loop can be expressed as a dynamic tail recursion, as in figure 4.32.

```
shared subgraph scan(theta, phi: integer) -> (Ftheta, Fphi: integer);

       subgraph body(Btheta, Bphi: integer) -> (Ntheta, Nphi: integer);
       begin (* body *)
              laser_plot(Btheta, Bphi) -> null;        (* plot all 'miss' points *)
              Btheta + INT -> Ntheta;                  (* new theta *)
              Bphi -> Nphi;                            (* new phi (invariant) *)
       end;

begin  (* scan *)
       if laser_miss(theta, phi) and theta < T2
       then `scan (body(theta, phi))             (* lazy evaluation *)
       else theta, phi                           (* NB. no access to 'close_range' *)
       -> Ftheta, Fphi;                          (* returns final loop variables *)
end;

(* INT and T2 are constants, close_range is a free variable in the surrounding graph*
... -> close_range;
..., ... -> theta, phi;                          (* initialisation *)
scan(theta, phi) -> Ftheta, Fphi;                (* call it *)
Ftheta, laser(Ftheta, Fphi) < close_range -> ... , ... ;  (* result expression *)
```

FIGURE 4.32 Dynamic tail recursive equivalent of ***while***

Similar identifier names, comments and layout have used to aid comparison with the previous example. In fact, this version could easily be expressed more concisely than shown here. The code is now more obvious and much easier to write, although certainly not as easy as if the syntax included a looping primitive. The program is now entirely functional and automatically determinate, reentrant and clean. One problem is the requirement to set the loop body apart in a subgraph of its own. This occurs often in DL1 programs, as it is the only way to introduce temporary bindings and is equivalent to the 'bindings' part of a SISAL or Id Nouveau *let* expression [35, 37].

The main advantage of writing this program as a dynamic tail recursion will become apparent when graph unravelling is discussed in §4.3.7. By creating a new context for each loop cycle (as is done by the tagging of the shared subgraph arguments), the entire loop can be *unfolded*. In this case, all activations of the procedure `laser_plot` can proceed in parallel, spread out over the entire dataflow machine (machine graph *unravelling*). The same is true for evaluation of new bindings in the loop body and for evaluation of the termination condition. This approach clearly generates very much more concurrency than the static version, since in that case, all loop cycles were constrained to execute in the same physical graph region.

There is a subtle issue regarding the evaluation of the result expression. If the result is evaluated in the context of the final recursive call, then any free variables that are required must be circulated as loop invariants to deliver them into that context (unlike the static translation). Alternatively, the result expression may be evaluated in the calling context, which requires all loop variables used in that expression to be returned. In any case, loop variables not involved in the result must be explicitly discarded. If tail optimisation is not used, as in figure 4.32, then it could be expensive to return a large number of loop variables to the outer context through the shared subgraph return mechanism, but the result itself may consist of many tokens equally as expensive to return. Similarly, free variables used in the result expression may be costly to include in the argument list of the recursion, because they must be tagged and circulated. There is a significant difference between these two cases however: the tagging of extra loop arguments may proceed in parallel with loop variables and thus presents little added cost; on the other hand, the return of extra loop results can not be done in parallel. These effects are investigated in chapter 5, but tail optimisation also has a large impact on this discussion.

There is no special template for the translation of a tail recursion without optimisation, since it is just coded like any other recursion, using the general shared subgraph caling mechanism.

4.3.6.3 Tail Optimisation

Static loop translations involve no context changes and return their results directly to the calling graph (*tail optimisation*). However, the overall graph is non-reentrant and will require protection to preserve determinacy. Dynamic loops, expressed as recursive shared subgraphs, are always reentrant and determinate because they return their results in order through a series of exits and conditionals (the tail); tail optimisation apparently cannot be used without sacrificing this order preserving property. Therefore, a tail optimised recursion cannot be pipelined without protection, as in the static case. Chapter 6 outlines some proposals which should prove more effective than protection in maintaining queue order, since far less concurrency is lost.

To implement tail optimisation in a dynamic loop the calling context is made known to the terminating cycle by including an extra argument which is the context (colour) of the calling graph. The results now exit through a `set-colour` node statically linked to the calling graph. Figure 4.33 shows a template for the translation of ***while*** loops into tail optimised machine graphs. Figure 4.34 illustrates the various loop translations at run time.

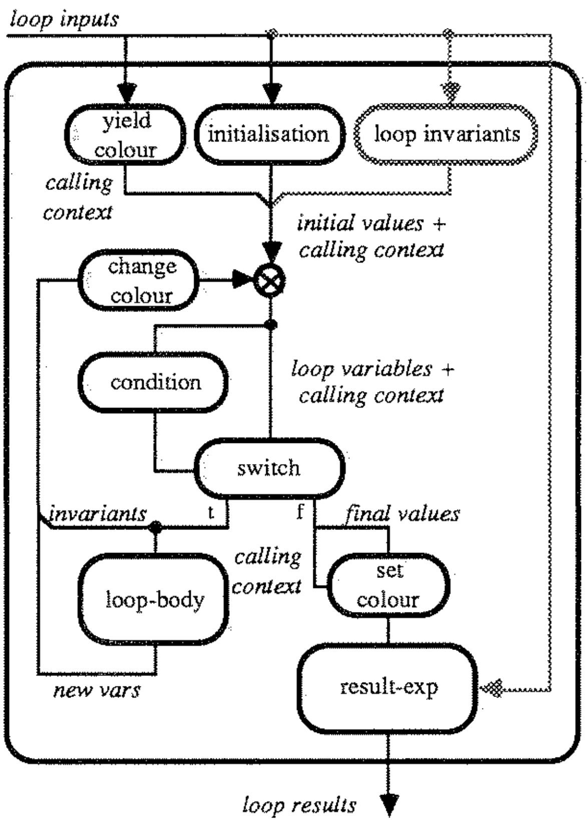


FIGURE 4.33 Dynamic ***while*** loop with tail optimisation

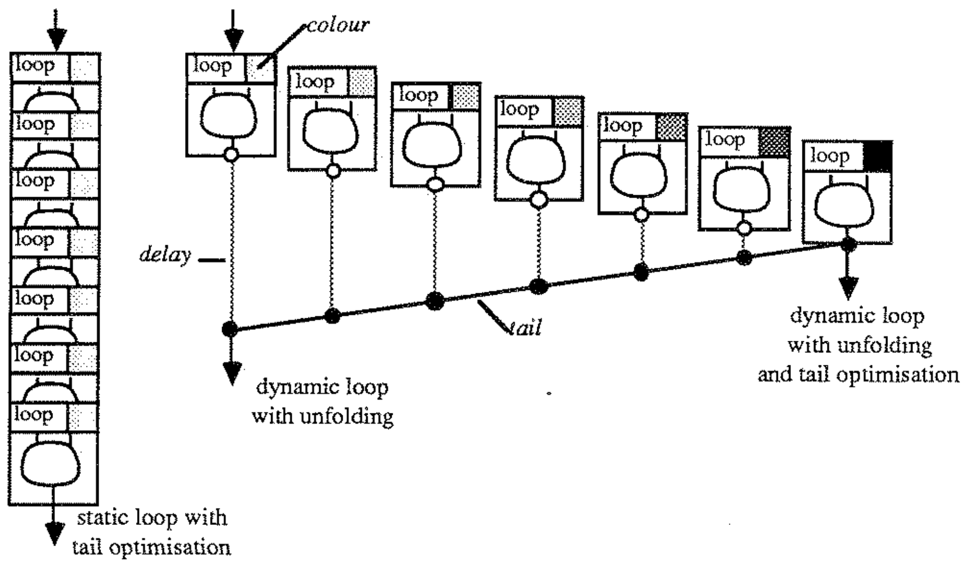

colour
loop
delay
tail
dynamic loop with unfolding
dynamic loop with unfolding and tail optimisation
static loop with tail optimisation

FIGURE 4.34 Run time characteristics of loops

4.3.7 Dynamic Graph Unravelling

It has been shown how the DL1/FLO system uses token tagging to implement code sharing/reentrancy in shared subgraphs, recursions and loops (as tail recursions), but the real power of tagging lies in its use to *unravel* dataflow graph execution. This is the process of executing multiple instances of a node, i.e. different coloured firings, in parallel. This is in keeping with the fundamental principle of dataflow based parallel computation that *any operations not dependent on each other for data may be executed in parallel.*

The processes of unravelling a dataflow graph and code sharing are similar in that the same tagging scheme is used to implement both ideas. In code sharing, the different instantiations of a single node are confined to one processing element or evaluation unit, thus reducing achieved concurrency through sequential evaluation of that node and communications problems brought about by corresponding *hot spots* in the graph. Unravelling, on the other hand, allows each different instantiation of a node to be executed independently, on separate processing elements. In the best case, when all instantiations have no data dependencies, they will be available for parallel execution and may execute as soon as a tagged much occurs.

The MIT Approach

The use of tags in the extraction of parallelism at run time was developed by Arvind and Gostelow in their work on the *U-interpreter* (unravelling interpreter) while at the University of California, Irvine [7]. They describe not only the use of tags as an implementation strategy for shared subgraphs, but also a model for execution that takes advantage of tagging to extract *maximum* available machine graph concurrency. In particular, they identify shared procedure application (including recursion) and dynamic loop unfolding as sources of concurrency, and structure their tags accordingly:-

```
<Tag> ::=
            <Color>
            <Physical-instruction-address>
            <Initiation-number>
```

The format shown is actual;y the one used currently by Arvind and the MIT dataflow group; it differs somewhat from the format described in [7] which was more conceptual in nature. *<Color>* separates tokens from disjoint procedure/loop invocations (nested loops are given a new *<Color>* because of limitations in the *<Initiation-number>* field). The actual destination node number is given by the *<Physical-instruction-address>* field and is considered part of the tag by the MIT group on the grounds that it is used in the token matching operation. *<Initiation-number>* is a finite implementation of an iteration counter and as such has overflow problems associated with its use, this field is part of an optimisation used in the control of dynamic loop unravelling.

The proposed MIT machine is a sophisticated implementation of the unravelling interpreter [7, 9]. Groups of processing elements, called physical domains (PDs) cooperate in the unravelled execution of loops and procedures. Within a given PD, a form of code sharing is achieved by allocating copies of the code body not to each processing element, but to consecutively addressed groups of PEs called physical subdomains (PSDs). A physical domain is made up of an integral number of subdomains, each of which has exactly one copy of the code body. A single PSD can utilise the static concurrency in the structure of a code body (i.e., its width), while several PSDs can execute #PSD/PE procedure or loop bodies in parallel.

The Manchester Approach

A very similar tagging scheme, based on *<Activation Name>*, *<Index>* and *<Iteration level>* fields was independently developed at the University of Manchester, England, by the group of John Gurd, Philip Treleaven and Ian Watson [31] at around the same time as Arvind, Gostelow and Plouffe's original work [8]. *<Activation Name>* and *<Iteration Level>* are similar to MIT's *<Color>* and *<Initiation-number>* fields respectively. Like the MIT design, nested iterations are not directly supported, but are implemented by calling inner loops through a procedure interface, thus changing the *<Activation Name>* before the inner loop begins. In this new context, the inner loop is free to reuse the iteration counter. The *<Index>* field provides for concurrency in operations on array like data, which was initially held in the matching store, rather than a separate structure sore unit. The Manchester group's work also emphasises that the tagged token approach increases available run time concurrency. More recently, the Manchester group have abandoned the *<Initiation-number>* field, due to the overflow and nested loop problems. This problem has been avoided altogether at RMIT since generating activation names, or colours, can be just as efficient as an iteration counter, much less complicated and free of restrictions, hence our emphasis on tail recursive loop implementations.

The single ring Manchester machine achieves an extra degree of unravelling through a tagged matching store and *multiple* function units within a single processing element. The function units are used in a manner that allows several instantiations of a node (as well as separate nodes of course) to execute in parallel. Evaluation results are simply merged together, so it is possible for the results from one node, within one PE, to be produced in a different order to which its argument matching occurred. This overtaking is acceptable in the Manchester design because the tags keep all tokens logically isolated, but is unacceptable in the hybrid model, since several node firings may have the same tag when arising from queues of operands.

The Manchester machine is designed to have several processing elements connected by a multi-layer switch, as in the RMIT design, but initial work was carried out on a single element only. More recently however, the multiple PE version of the Manchester machine, the Multi-ring Dataflow Machine (MDM), has been simulated [12]. As part of this simulation, *split functions*, which route tokens between elements based on their tags and static addresses, were investigated. The split function is arranged so that an integral number of copies of a graph can be addressed in a machine of any number of PEs. The results clearly show the effects of machine graph unravelling produced by tagging and splitting.

The RMIT Approach

The original FLO system had no provision for graph unravelling at the machine level since there was only one physical copy of every node in the graph. The graph was simply statically distributed over the entire dataflow machine in a random manner, addresses being based on static element and node numbers. Multiple function units within PEs were allowed, as in the Manchester design, although the problem of avoiding result overtaking was always present and never fully accounted for. In recognition of the need for unravelling, to improve achieved concurrency and machine utilisation, a scheme has been devised for allowing independent (separately tagged) node activations to proceed in parallel. In the following discussion we concentrate on 'colours', although the arguments also apply to tags generated by the copy number scheme.

The initial implementation is similar to the 'splitting' approach of the Manchester group (although developed independently); a hashed addressing scheme is outlined below. It is also illustrative to draw comparisons with the MIT design. To borrow from their terminology, the entire RMIT machine represents one physical domain, responsible for all code execution. Within this PD there are as many physical subdomains as there are processing elements, but a single PE does not represent one subdomain, since a subgraph application with any given colour is still statically distributed over the entire machine. The PSDs actually overlap each other, with the effect that achieved concurrency smoothly degrades as the number of instantiations of a code block increases, rather than suddenly degrading when this number exceeds #PSD/PD, as would be expected with the MIT design.

Currently, each element must hold a copy of the entire graph if dynamic unravelling is used, although a simple change to the hashing function can restrict the number of code copies required [12]. This system achieves high machine utilisation and run time concurrency at the cost of increased storage requirements.

Parallel execution of coloured node activations is achieved by hashing the colour and destination PE fields of a token together to form a physical PE address, like the Manchester split. The effect of including the destination PE in the algorithm is to distribute the computation for a given colour as per the static allocation generated by the compiler and allows more 'intelligence' to be built into the unravelling. The problem of choosing a destination PE at run time, i.e., a suitable hashing algorithm, is similar to the problem of statically allocating a graph within a physical subdomain at compile time. Results are presented in §5.6, which show the simulated effects of the allocation strategies used by DL1 (currently random) and the RMIT emulator (hashed/computed as shown below).

```
(*
 * Compute HashPE from static PE and Colour.
 * Don't hash if token is going to a system node.
 * NumPEs is the number of physical processing elements.
 * Colours include the originating PE number.
 *)

HashPE := PE;
if not SystemNode then begin
        HashColour := Colour;
        while HashColour <> 0 do begin
                HashPE := xor(HashPE, HashColour mod NumPEs);
                HashColour := HashColour div NumPEs;
                end
        end
```

FIGURE 4.35 The hashing algorithm used to unravel machine graphs

The algorithm of figure 4.35 is used in the RMIT simulator, DFSIM, but is oriented towards hardware implementation by the use of exclusive-or rather than addition. As DFSIM can model a user defined number of processing elements, the divisor NumPEs is a variable in the above algorithm, but an actual machine would fix this value. Tokens destined to certain system nodes must not be redirected, since these nodes currently have different meanings on different PEs. The desired features of the hashing algorithm are that the compiler generated allocation is preserved for any given instantiation, and different instantiations are spread uniformally across the machine.

Although the sequence of tags is expected to be well distributed for both copy numbers and colours (counter generated), a simpler scheme such as

```
        HashPE := (PE+Colour) mod NumPEs
or      HashPE := xor(PE,Colour) mod NumPEs
```

is still not used. This is because the originating <PE> field of the colour (which is in the upper byte) would be masked out of the hash so that subgraph calls from different PEs, but with the same <colour> subfield would clash.

Some observations:-

- Static code sharing (i.e., not unravelled) reduces graph size, but lowers available concurrency and machine utilisation.
- Unravelled graph execution requires many copies of shared code to be loaded, but increases available concurrency and machine utilisation.
- Smaller physical domains could be implemented by including domain size and location fields in tags. These fields would be interpreted as PE numbers and would appear in the hashing algorithm for unravelling. This method would relieve the compiler of the burden of keeping track of the absolute static locations of domains.

4.3.8 Sequences and Streams

Queueing is one way of taking advantage of *temporal concurrency* within processing elements, in addition to the *spatial concurrency* achieved by distributed processing over different elements. However, it has the drawback of introducing an extra time dependency between data so that it is not an optimal strategy for graph execution. Some potential concurrency is lost in maintaining all intermediate results in time order, since computations on these results might otherwise proceed in parallel (unravelling again). Many researchers have rejected queueing for this reason, leading to the purely tagged or dynamic architectures with no support for queues on arcs apart from the possibility of colouring the different elements of a queue to keep them isolated. The RMIT group has not rejected queueing for these reasons, but has looked instead for ways to take advantage of queues.

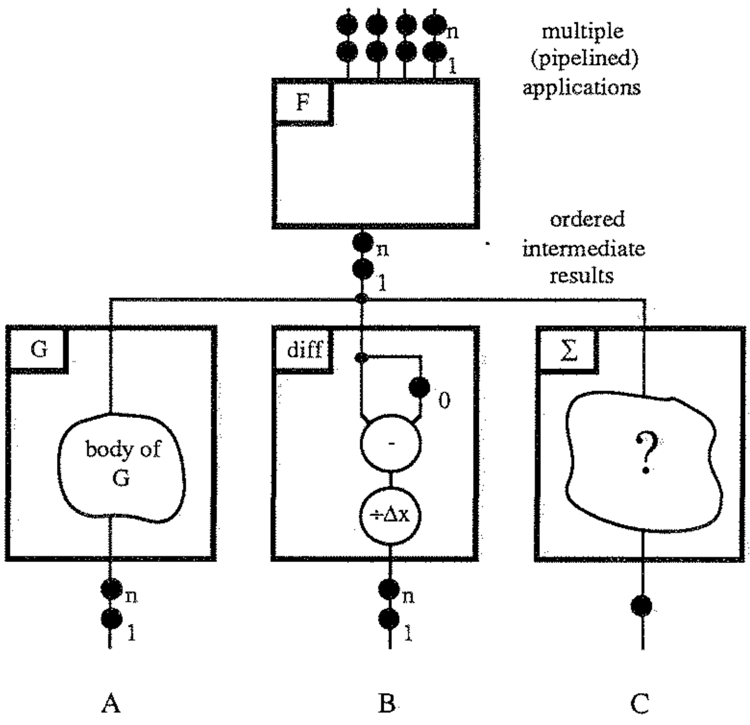


FIGURE 4.36 Multiple function applications

In figure 4.36, the application of function F to a sequence of untagged input tokens may proceed suboptimally because any given result may be held up by a prior result which takes longer to evaluate. Tagging the input tokens so that each application of F can proceed independently will produce the results in the shortest possible time (all else equal), except that a small overhead for the actual tagging will be incurred. Also, in the tagged case, the results will almost certainly be produced in the wrong time order, their actual sequence position has to be derived from the tags they carry.

There are three basic operations that may be applied to the sequence of (possibly tagged) intermediate results produced by F:-

(a)

A function such as G, defined over simple token types, can be applied to the intermediate results to produce a set of output results. If the tokens in the original sequence are all separately tagged, then further unravelling will occur within G, to produce a set of correspondingly tagged outputs in close to minimal time. The time ordering of the tagged results will again change (probably), possibly even back to the order of the original input stream. If tagged results must be output in their correct time order, then special i/o interfaces, which sort the tokens based on their tags, are required. Alternatively, tokens may be output with their tags attached, to be interpreted directly by another dataflow graph.

If tokens are not tagged, then G will operate over the intermediate result sequence in a strictly determinate, functional manner, maintaining time ordering throughout. Some gains are made by having no tags [1], but these will almost certainly be offset by the lost asynchrony due to not unravelling the graph [7].

(b)

The simple differentiator `diff`, shown in figure 4.36 requires that the input sequence arrives in correct time order. This will be true in general of any graph which, like `diff`, uses state information and in some way combines the values of tokens in a sequence. The suitability of the hybrid dataflow architecture to the execution of such graphs has been clearly demonstrated in many applications [41, 42, 24, 2], and is clear from the extreme simplicity of `diff`. The action of these graphs can almost certainly be simulated on machines without queueing, but the code required to manipulate tags and possibly shared storage devices may add significant complexity and inefficiency to the graph.

(c)

Finally, a graph such as Σ may be required that takes all the values in a given sequence and produces a new sequence of different length. In this case, Σ returns just one result, being the sum of the tokens in the input sequence. In fact, Σ cannot be implemented in the context of figure 4.36 because their is no way of knowing when the input sequence is finished. A distinction must therefore be made between *open* and *closed* sequences.

4.3.8.1 Open Sequences (Queues)

Open sequences are supported in DL1 by allowing queues of tokens to be streamed through a graph segment. Such filter type graphs can have state variables, allowing powerful graphs to be built very efficiently, e.g.,

```
subgraph diff(V: integer): integer;
        constant
                dT = 0.01;
        begin  (* differentiate the input sequence *)
                (V - old_V) -> dV;
                dV / dT -> diff;
                V -> old_V;
                prime 0 -> old_V; (* initialise old_V *)
        end;

begin  (* main *)
        ...
        (* read a sequence, differentiate it and write it out *)
        read(input, more) -> V;
        V -> more;
        write(output, diff(V)) -> null;
        ...
        prime true -> more; (* gets it going *)
end.
```

FIGURE 4.37 Program segment that filters a queue

This program segment has one state variable, `old_V`, which is used in a continuous differentiation filter on the input sequence. Elements are processed as they arrive, which is ideal for applications like real time signal processing. The data dependencies are such that no advantage would be obtained by tagging/unravelling, indeed the graph would be substantially more complicated if this was attempted. Note that if `diff` was a shared subgraph, then each application to a given queue would have to have the same colour in order to use the history mechanism provided by `old_V`. The **prime** statement is executed only once and is used to initialise `old_V` and `more` in this graph. As DL1 already provides support for open sequences in sufficient generality, we shall move on to the subject of closed sequences or *streams*.

4.3.8.2 Streams

Weng introduced the semantics of stream based computation on a static dataflow architecture and showed how streams could be used to provide interprocess communication and to increase both available and achieved concurrency [50]. His implementation, in common with the current DL1 approach, is based on queues of tokens (of the base type of the stream, if declared) terminated by a special *end-of-stream* token ']'. An empty stream consists of a single *end-of-stream* token. This implementation forces the structure of a nested stream to be declared statically to avoid ambiguity, e.g., the sequence `1,2,3,4,],5,6,],],]` could be a queue of four simple streams, or a much more complex structure such as '**stream of** (**stream of** (**stream of integer**))', whereby the sequence would be interpreted as `({<1,2,3,4,]>,<5,6,]>,]},])`, with the symbols `()`, `{}` and `<>` representing the three levels of nesting. This interpretation is most easily seen if the sequence is read backwards.

On the RMIT architecture, streams provide enhanced run time performance through pipelining and significantly more powerful semantics for high level DL1 programs, but it must be emphasised that this implementation, which carries streams on data arcs, is not suited to random access of stream elements because of the amount of data copying implied. For example, to find the nth member of a stream requires that every element be processed n times, through n-1 **tails** and 1 **head**. Never the less, sequential access data structures, like streams, can be of great benefit in interprocess communication, and provide a powerful abstraction over simple (atomic) data types. Also, the problems associated with stored data structures are avoided by this approach. The new node set being developed for the RMIT machine supports structure store based lists which allow more efficient pseudo-random access through indexing and indirection. In addition, *start-of-stream* and *stream-separator* tokens, and a more general *stream* matching function will be supported, which allows a fully dynamic implementation, requiring no static declarations at all. In certain cases it may even be possible to unravel streams by uniquely colouring all the elements and processing them in parallel; such a scheme is used in SISAL [35].

4.3.8.3 Stream Functions

Stream 'variables' are indicated by identifiers beginning with an underscore, e.g., `_i_am_a_stream`. This scheme has been chosen to limit the number of 'typing' statements in DL1, and to emphasise the use of streams as sequential rather than random access items by preventing nested definitions. In any event, the code templates provided can not handle nested streams without significant modifications.

Predefined operators on streams are: **bracket, unbracket, head, tail, get, empty** and **cons. Bracket** returns a copy of the input data followed by an *end-of-stream* token. **Unbracket** absorbs all *end-of-stream* tokens passing through it. Note that these two operators are not functional and require extreme care to ensure clean, well formed code.

The new stream functions **head, tail, get, empty** and **cons** allow for easier and safer stream manipulation. **Get** returns both the head and tail of a stream and is slightly more efficient than a **head/tail** pair. **Cons** allows for non-strict stream creation by concatenating a simple element with an existing stream. Because of its non-strict nature, **cons** will begin production of its output stream as soon as its first input arrives and then as successive tokens arrive on its second input, up to and including the *end-of-stream* token. This non-strict operation is important in improving run time concurrency and in providing a logically consistent interpretation of streams, e.g., as potentially infinite data structures. **Empty** returns `true` when applied to an empty stream and should be used to test streams before applying **head, tail** or **get** since these functions are not defined for empty streams. In addition, implicit stream support is provided by many other DL1 primitives including merging, protection, shared subgraph entry/exit, etc..

A recursive subgraph to sum the elements in a stream could be:–

```
shared subgraph stream_sum( _input: integer): integer;
        begin
                if empty(_input)
                then 0
                else ` head(_input) + stream_sum(tail(_input))
                -> stream_sum;
        end;
```

and to sum the corresponding elements of two streams to form a new stream:–

```
shared subgraph _add( _in1, _in2: integer): integer;
        begin
                if empty(_in1) or empty(_in2)
                then ]
                else ` cons(head(_in1) + head(_in2),_add(tail(_in1), tail(_in2)))
                -> _add;
        end;
```

where the *end-of-stream* symbol ']' denotes the empty stream. Lazy evaluation of the else branch is necessary to prevent **head** and **tail** from being applied to empty streams, also, this subgraph is well formed and clean even if the input streams are of different length.

4.3.8.4 Code Templates for Stream Functions

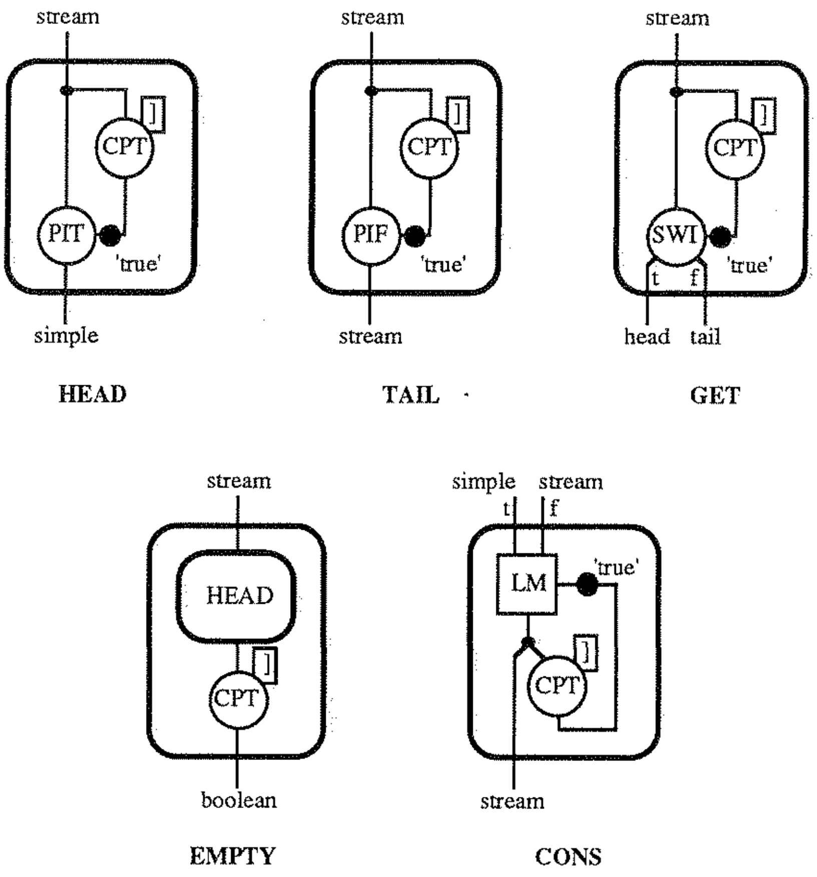


FIGURE 4.39 Simple templates for stream functions

The basic templates shown in figure 4.39 are similar to those proposed by Weng in his original work on streams. Due to the use of priming tokens, these templates are not suitable for shared subgraphs on the RMIT system, but do represent the simplest implementation based on the original FLO node set. The code is well formed, since the functions return to their original state after a stream has been processed, but it is not clean because a token is left in the matching store for each context in which they are activated. One can imagine the matching store slowly clogging up with tokens from unclean functions like this unless action is taken to either prevent or remove these tokens.

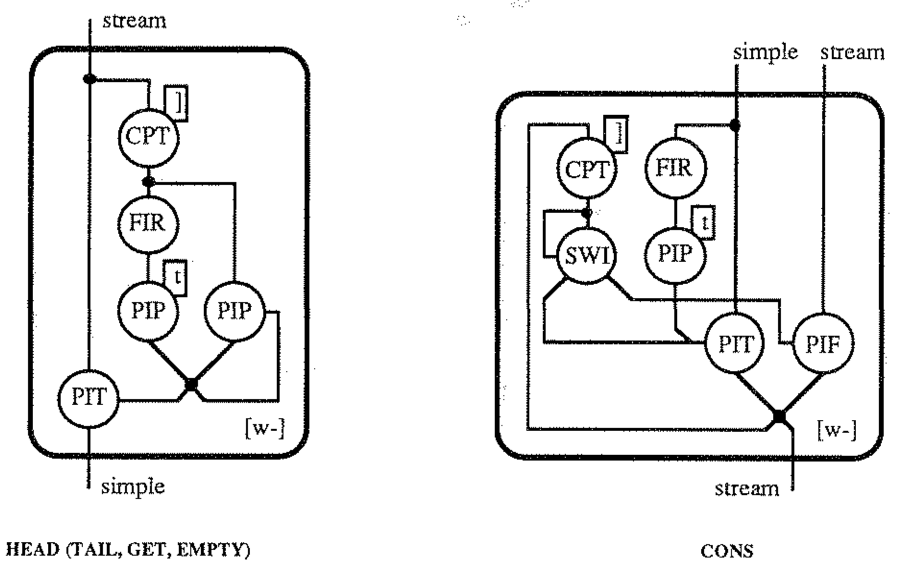


FIGURE 4.40 Stream functions for shared subgraphs using the basic node set

Figure 4.40 shows templates which are suitable for shared subgraphs because they have no priming tokens. These templates are still unclean however, because after the first activation, a *boolean* token remains, as in the prior case. In addition, a further token is left in the matching store to indicate the state of the `first` node.

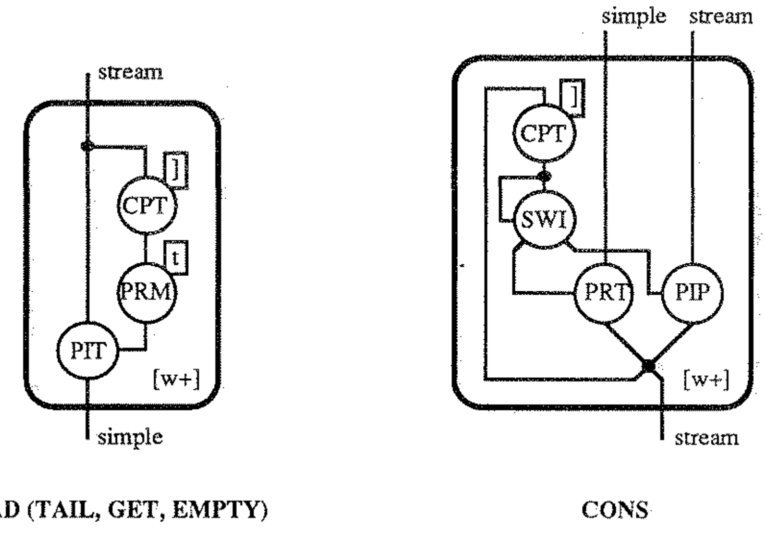


FIGURE 4.41 Stream functions for shared subgraphs using the extended node set

The templates of figure 4.41, show how the extended node set has been used to reduce the complexity of stream functions. The template for **cons** is now clean because the `PRT` node resets after a stream has been processed. The reset feature of the `prime` node is not used in the other templates as it is not determinate (yet), these templates remain unclean.

There is a need to provide macros for several other operations on streams, many of which require a code block that will accept a stream and a simple token and reproduce a copy of that simple token for every element in the stream. This macro is called the `stream-store` function. Its use will be outlined shortly (see also ch 3), but first its expansions are given for the various node set possibilities in figures 4.42 and 4.43. `Stream-store` has been implemented as a non-strict function, like **cons**, to increase concurrency in stream handling macros. Thus a copy of the simple token can be produced before the start of the stream arrives.

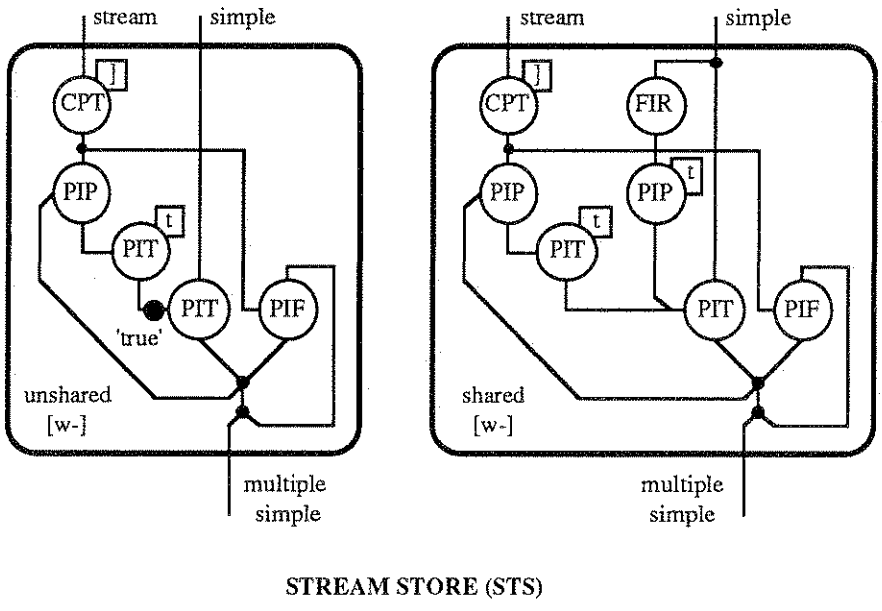


FIGURE 4.42 The stream-store (STS) expansion in the basic node set

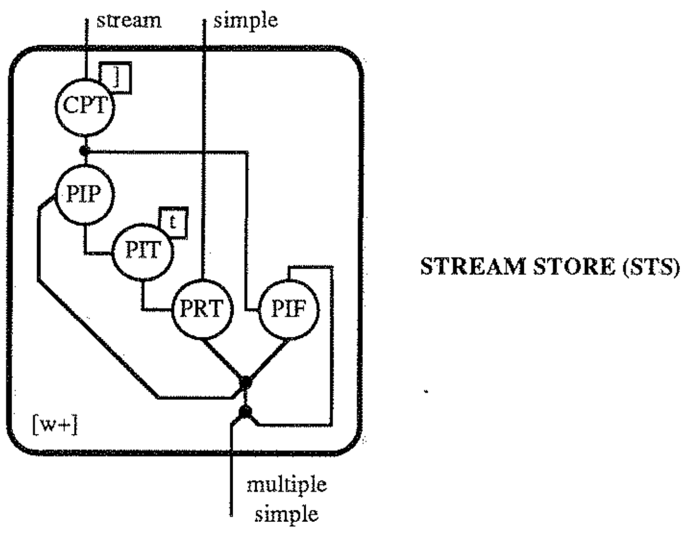


FIGURE 4.43 Stream-store for the extended node set

The use of `stream-store` becomes clear in the following macros. Figure 4.44 shows expansions for stream gates, operators that replace the simple functions of `pass-if-true, pass-if-false` and `switch` when the data to be gated is a stream. They will be referred to as `PIT(S), PIF(S)` and `SWI(S)`.

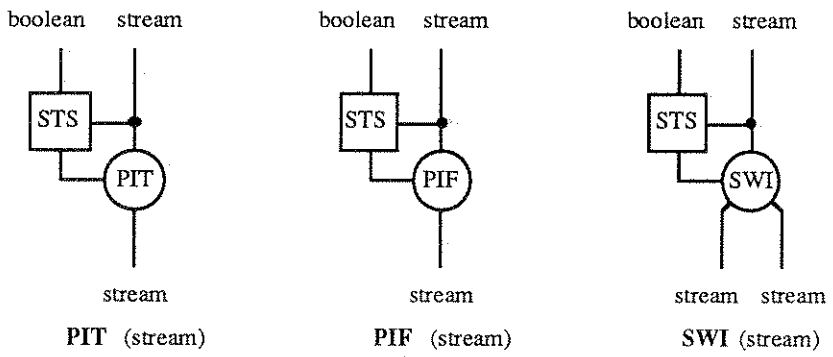


FIGURE 4.44 Path control macros in the stream node set

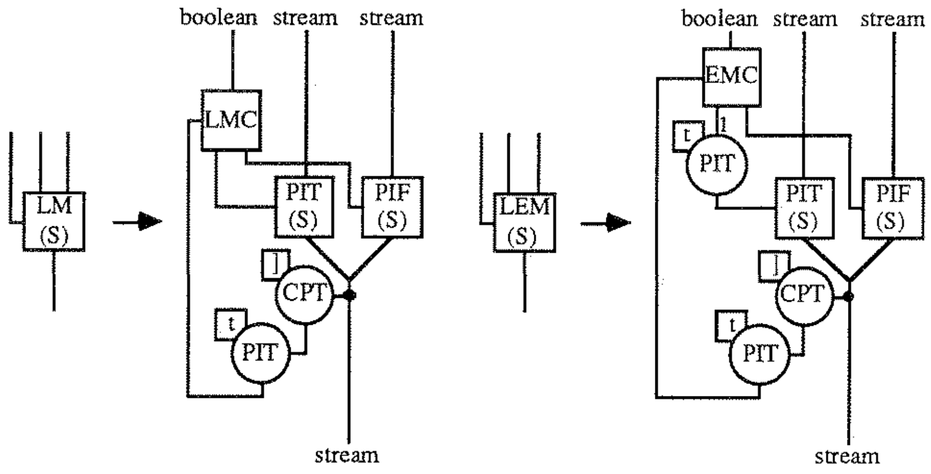


FIGURE 4.45 Controlled merge expansions for streams

Figure 4.45 gives expansions for the lazy and hybrid lazy/eager merge of two streams under control of a single boolean token. Similar expansions apply for the eager and eager/lazy case. With these macros and the gate macros shown above it is possible to handle streams in conditional statements in full generality, using the templates of §4.3.4.3.

Another place where special handling is required for streams is in the interface to shared subgraphs. The idea here is to use `stream-store` to generate multiple copies of tags, return addresses, etc., for appending to the elements of streams involved in shared subgraph calls and returns, see figure 4.46.

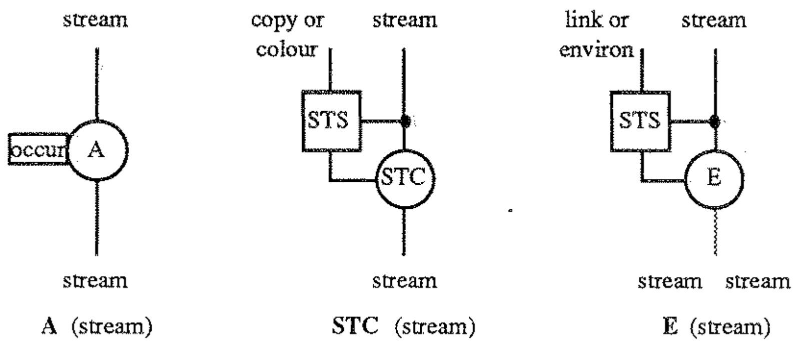


FIGURE 4.46 Shared subgraph expansions for streams

To greatly reduce the complexity of stream graphs, and as part of an experiment into the use of advanced matching functions, single nodes have been implemented for many of the stream functions seen so far, see figure 4.47. These nodes are available when the advanced node set toggle [a+] is set during a compilation. See chapter 3 for a discussion and implementation of the advanced matching functions used by these nodes. Results of simulations involving streams are presented in chapter 5.

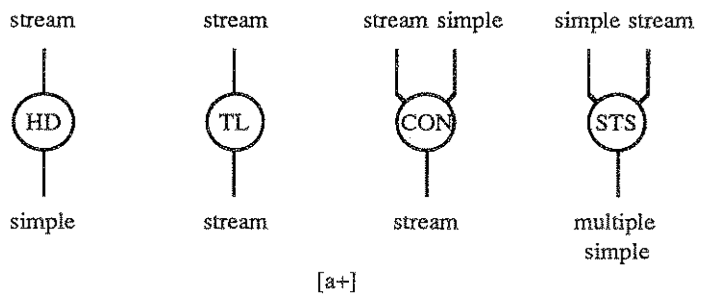


FIGURE 4.47 Special stream nodes in the advanced node set

Chapter 5
SIMULATION RESULTS AND ANALYSIS

5.1 Introduction

In this chapter, various aspects of the dataflow system described in this thesis are simulated and discussed. The timing figures used in all simulations have been derived from an evaluation of the prototype emulator code and represent the performance of a MC68020/M68881 based processing element in single CPU configuration (§2.9). Where possible, results are presented in a comparative manner as absolute values for the dataflow system may bear little or no relationship to analogous values for conventional machines. Emphasis has been placed on trends for future development so that results are relevant to the design of the next generation system where possible.

5.2 Test Programs

A suite of test programs has been written to allow various areas of interest to be studied, especially those features peculiar to the RMIT system. Simulations were performed on the CSIRO Sun 3/260 which handled most graphs comfortably with the exception of the recursive 8 queens solution. This program aborted due to a lack of swap space after generating a massive token load during simulation (> 4 million tokens in transit and in the data structures of the matching store emulator) and took several days to run to completion on an alternative VAX 11/780 system. A 6 queens version simulated in rapid time and was thus preferred to 8 queens. Selected test program listings are included in appendix A.

5.3 Duplicate vs Replicate

During the execution of a dataflow graph, many tokens must be duplicated. This occurs explicitly, where identifiers are used more than once in expressions, function calls, etc., and implicitly where the compiler uses duplication in code templates. In the case where nodes are limited to two outputs, a restriction found in many dataflow implementations, it is necessary to plant a tree of `duplicate` nodes to produce many copies of one token. For obvious performance reasons the duplicate tree is made height balanced, it requires one less node than token copies to be made and has a height proportional to the log of the number of copies. Analysis of the test programs, together with previous studies [41, 42], shows that the `DUP` nodes in these trees consistently account for about 40% of all nodes, both static and dynamic (they form the majority of all one input nodes, which have themselves been shown to account for typically 60 - 70% of all nodes executed).

As an alternative to the token duplication scheme used by DL1 (i.e., to plant a height balanced tree of `DUP` nodes), a new node was defined called `replicate`. `REP` can generate any number of copies of its single input token; it has an arbitrarily long output list and takes time $O(\mathrm{n})$ to process n tokens as opposed to $O(\log \mathrm{n})$ for a `DUP` tree. A `DUP` tree contains `n-1` nodes which are randomly distributed by the compiler (an optimal strategy would take advantage of the wave like processing of the tree to use as few elements as possible, but this would still be at least `n/2` elements since the last wave should execute in parallel). `REP` uses only one processing element and will clearly be more efficient for small `n` due to the large reduction in token traffic. However, a break even point will exist between the execution times of graphs with `duplicate` trees and those with `replicate` nodes.

To study these effects, a graph was simulated which had one (variable sized) duplicate tree, primed with one token. The outputs of the tree were connected to dummy nodes, in this case `AND` with literal `FALSE`, in order to correctly simulate the passage of the result tokens from the tree (these tokens would be discarded if the tree had no output connections). The graph was simulated with 'flattening' on and off. With flattening on, DFSIM treats duplicate trees as single `replicate` nodes with multiple outputs (the DL1 compiler is being modified to support multiple output nodes, as will be used in future node sets). The simulated *result distribution unit* steps through the output list in a time proportional to the queue write time and token size (ch. 2).

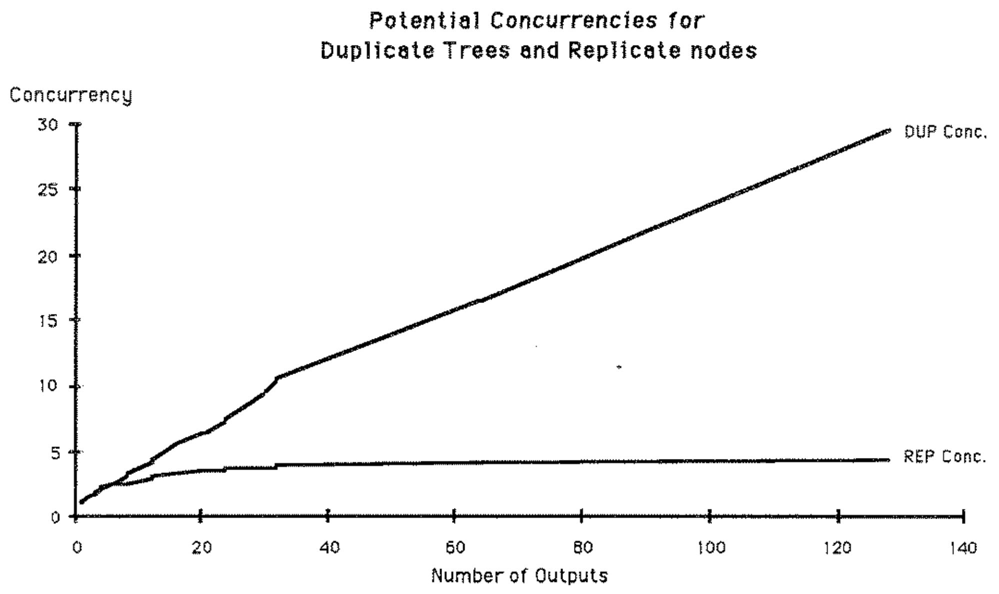


FIGURE 5.1 Potential Concurrencies for duplicate trees and replicate nodes

Figure 5.1 shows the potential concurrencies for different tree sizes. Evaluation of the AND nodes is included in this result. The concurrency rises linearly in the DUP tree case, as would be expected, since the nodes on each level of the tree can potentially execute in parallel. However, for the REP node simulation, the concurrency rolls off at a maximum approaching 4 which represents parallel activity in the destination processing elements. There is no concurrency in the processing of the REP node itself.

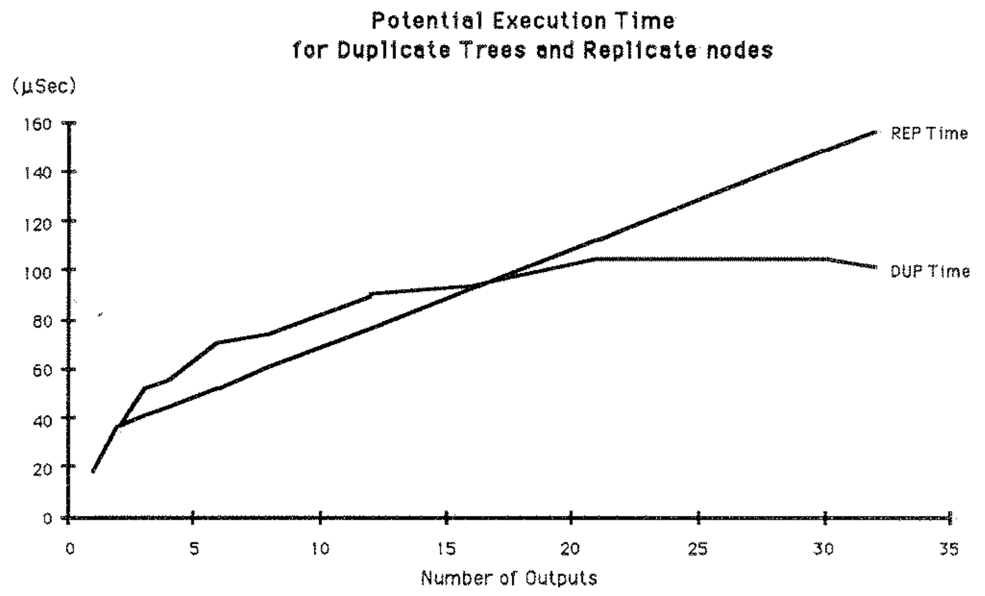


FIGURE 5.2 Potential Execution Times for DUP Trees and REP nodes

Figure 5.2 shows potential execution times for these simulations. As expected, the time taken is logarithmic for DUP trees and linear for REP nodes. The break even point is at about 15 outputs, but this figure is critically dependent upon the times assumed for the various processing element activities involved and even the length of the tokens used (in this case all tokens were untagged booleans, 64 bits long). The effect is certainly present however, and will play a large role in the selection of a maximum destination list size for nodes, if such a limit is imposed. Also, it would be useful in determining a reasonable size for any buffering that may be be required between the evaluation unit(s) and the result distribution unit. Once a reliable figure for the break even point is established for a particular machine, it can be used as the basis for compiler optimisation of the code planted for token distribution, i.e., a `DUP` tree (or even better, a `REP` tree) can be used when the number of outputs exceeds the break even figure.

5.4 Influence of Node Set and Matching Functions

The effect of the special matching functions described in ch. 3 was studied by using compiler toggles to control the code generation for several graphs, all of which deal with streams. By using stream graphs, the effect of the advanced stream node set can be observed, while the results obtained for the extended node set have been found to apply to most graphs, whether or not streams are present. Three cases were considered: the basic node set (the DL1 default, allowing *monadic*, *diadic*, *storage* and (now) *first* matching functions); the extended node set (allowing the *protect* and *prime* functions, which allow cleaner, reentrant code, even in shared subgraphs); and the advanced stream node set (allowing the special *head*, *tail*, *stream* and *cons* matching functions).

Three recursive stream graphs were considered: `reverse`, `serial_sum` and `binary_sum`; listings are given in appendix A. Some simple lazy and eager conditional expressions were also analysed to specifically show the effects of combining the `stream_store (STS)` node with gate nodes like `pass_if_true (PIT)`, etc. (§4.3.8.4).

Table 5.1 lists the simulation results for these test graphs. The effects of the extended and advanced node sets are quite pronounced, generally resulting in clearly improved performance in all areas. All graphs were simulated using a 128 element machine configuration, which results in a low machine utilisation (§5.7) and high concurrencies, and gives a better picture of the execution characteristics of the graph being simulated. Note the high actual to potential yield in the table, being typically greater than 70%. If fewer elements were used, then achieved concurrencies and execution times would be degraded and the element activity plots would be distorted by 'backed up' activity due to high queue occupancy/latency.

Graph "reverse.itl" (128 PEs)

Node Set	**Static Size**		**Dynamic Size**		**Concurrency**	
	(nodes)	(tokens)	(nodes)	(tokens)	(potential)	(actual)
basic	171	76	27783	37417	38.7	29.0 (74.9%)
extended	135	76	27544	35632	45.5	30.6 (67.3%)
advanced	67	50	10967	14009	24.9	18.7 (75.3%)

Node Set	**Execution Time (sec)**		**Instructions/Sec**		**Machine Utilisation**
	(potential)	(actual)	(potential)	(actual)	
basic	0.032134	0.042914	864598	647411	22.6%
extended	0.025726	0.038210	1070668	720858	23.9%
advanced	0.019739	0.026211	555601	418412	14.6%

Graph "serial_sum.itl" (128 PEs)

Node Set	**Static Size**		**Dynamic Size**		**Concurrency**	
	(nodes)	(tokens)	(nodes)	(tokens)	(potential)	(actual)
basic	78	76	13581	18799	22.7	18.5 (81.5%)
extended	56	76	11239	14338	22.9	17.8 (78.0%)
advanced	31	50	3624	5072	12.5	10.2 (81.8%)

Node Set	**Execution Time (sec)**		**Instructions/Sec**		**Machine Utilisation**
	(potential)	(actual)	(potential)	(actual)	
basic	0.026238	0.032178	517600	422053	14.5%
extended	0.020298	0.026031	553689	431748	13.9%
advanced	0.014314	0.017493	253172	207164	4.4%

Graph "binary_sum.itl" (128 PEs)

Node Set	**Static Size**		**Dynamic Size**		**Concurrency**	
	(nodes)	(tokens)	(nodes)	(tokens)	(potential)	(actual)
basic	231	76	56130	76912	61.6	43.6 (70.8%)
extended	165	76	46311	59793	60.6	40.1 (66.1%)
advanced	90	50	18346	25648	33.6	25.9 (77.0%)
switch/join	99	50	21023	28797	35.1	26.7 (76.1%)

Node Set	**Execution Time (sec)**		**Instructions/Sec**		**Machine Utilisation**
	(potential)	(actual)	(potential)	(actual)	
basic	0.041700	0.058932	1346059	952460	34.1%
extended	0.033659	0.050913	1375908	909619	31.3%
advanced	0.027376	0.035537	670161	516258	20.2%
switch/join	0.028192	0.037079	745721	566983	20.8%

Graph "if_lazy_stream.itl" (128 PEs)

Node Set	**Static Size**		**Dynamic Size**		**Concurrency**	
	(nodes)	(tokens)	(nodes)	(tokens)	(potential)	(actual)
basic	34	34	382	528	3.8	3.7 (97.8%)
extended	34	31	382	525	3.8	3.7 (97.8%)
advanced	15	31	155	228	2.6	2.6 (98.7%)

Node Set	**Execution Time (sec)**		**Instructions/Sec**		**Machine Utilisation**
	(potential)	(actual)	(potential)	(actual)	
basic	0.002681	0.002742	142484	139314	2.9%
extended	0.002664	0.002725	143393	140183	2.9%
advanced	0.001827	0.001851	670161	516258	2.0%

Graph "if_eager_stream.itl" (128 PEs)

Node Set	**Static Size**		**Dynamic Size**		**Concurrency**	
	(nodes)	(tokens)	(nodes)	(tokens)	(potential)	(actual)
basic	25	32	298	402	7.9	6.7 (85.7%)
extended	24	29	297	398	7.7	6.8 (88.6%)
advanced	10	29	127	172	14.6	5.1 (34.5%)

Node Set	**Execution Time (sec)**		**Instructions/Sec**		**Machine Utilisation**
	(potential)	(actual)	(potential)	(actual)	
basic	0.000970	0.001132	307216	263251	5.3%
extended	0.000991	0.001118	299697	265653	5.3%
advanced	0.000237	0.000686	535865	185131	3.9%

TABLE 5.1 Showing the effect of extended and advanced node sets on stream graphs

Figures 5.3, 5.4 and 5.5 show the simulated activity plots for the three larger stream graphs. Of particular interest are the token and element activities (graphs 1 and 3) and the individual element activity picture. Graph 2 (System Time), shows how time is divided between machine tasks during the run and is discussed further in §5.8.

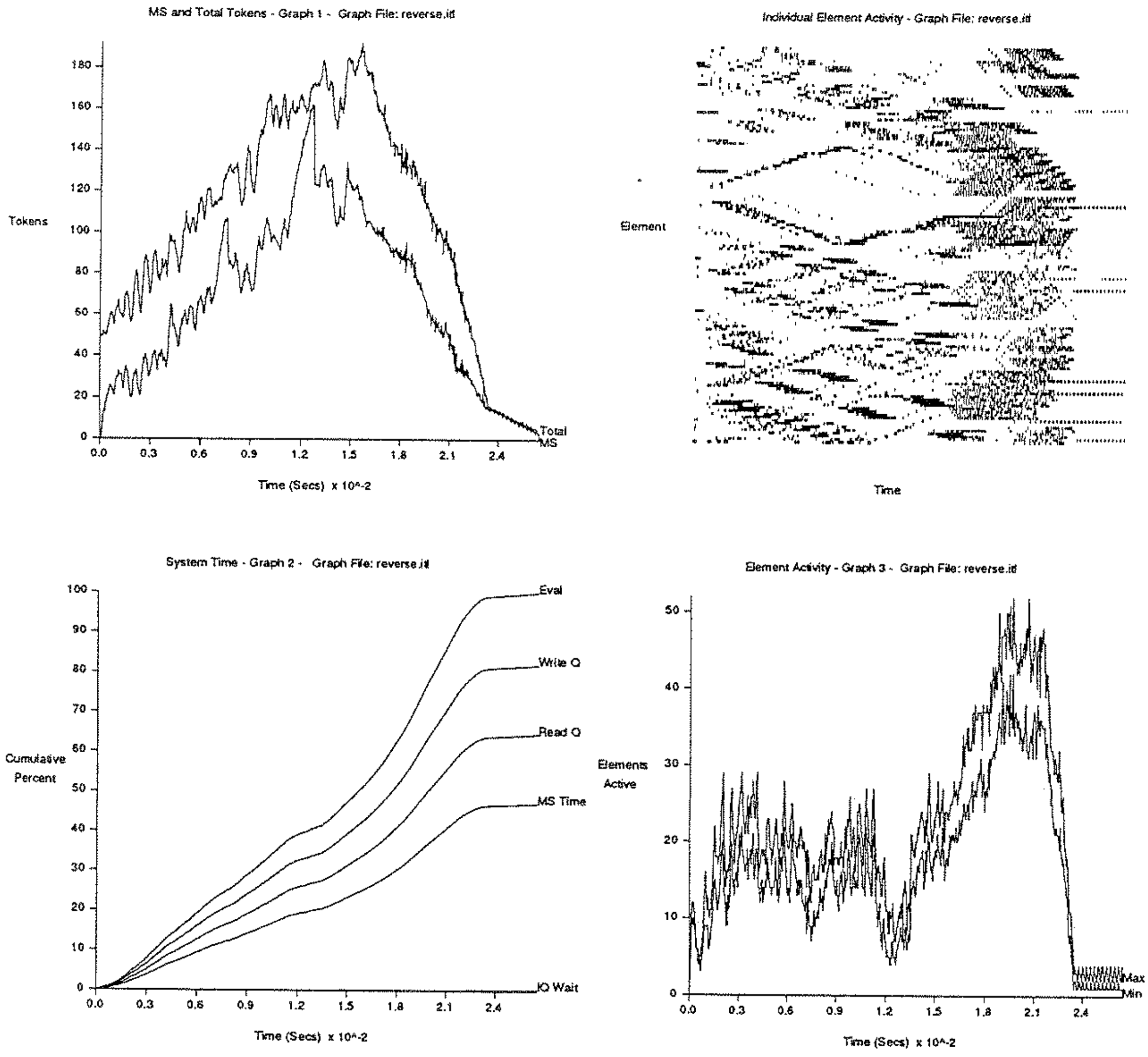


FIGURE 5.3 Activity plots for the stream reversal graph (advanced node set)

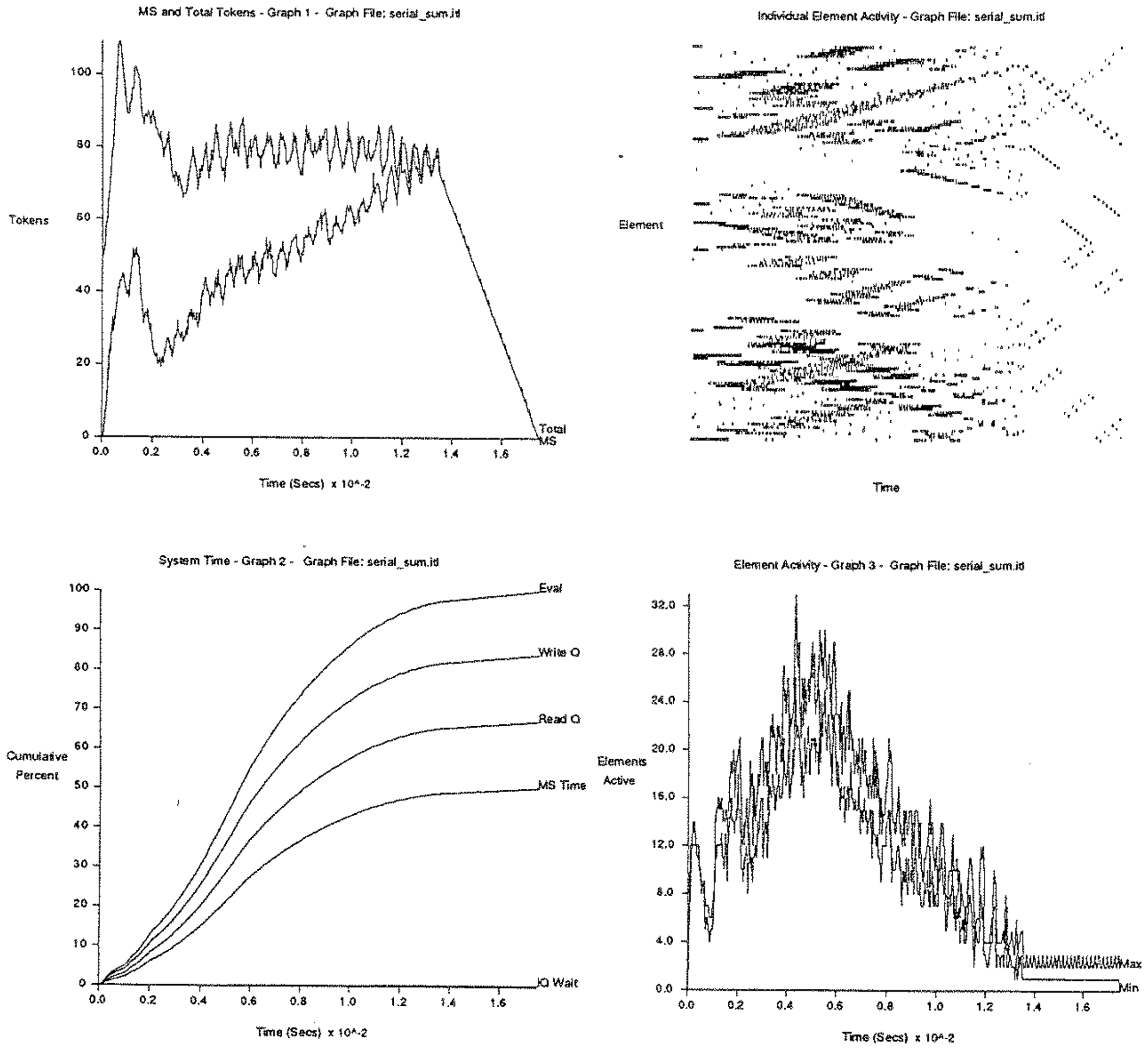


FIGURE 5.4 Activity plots for serial stream summation (advanced node set)

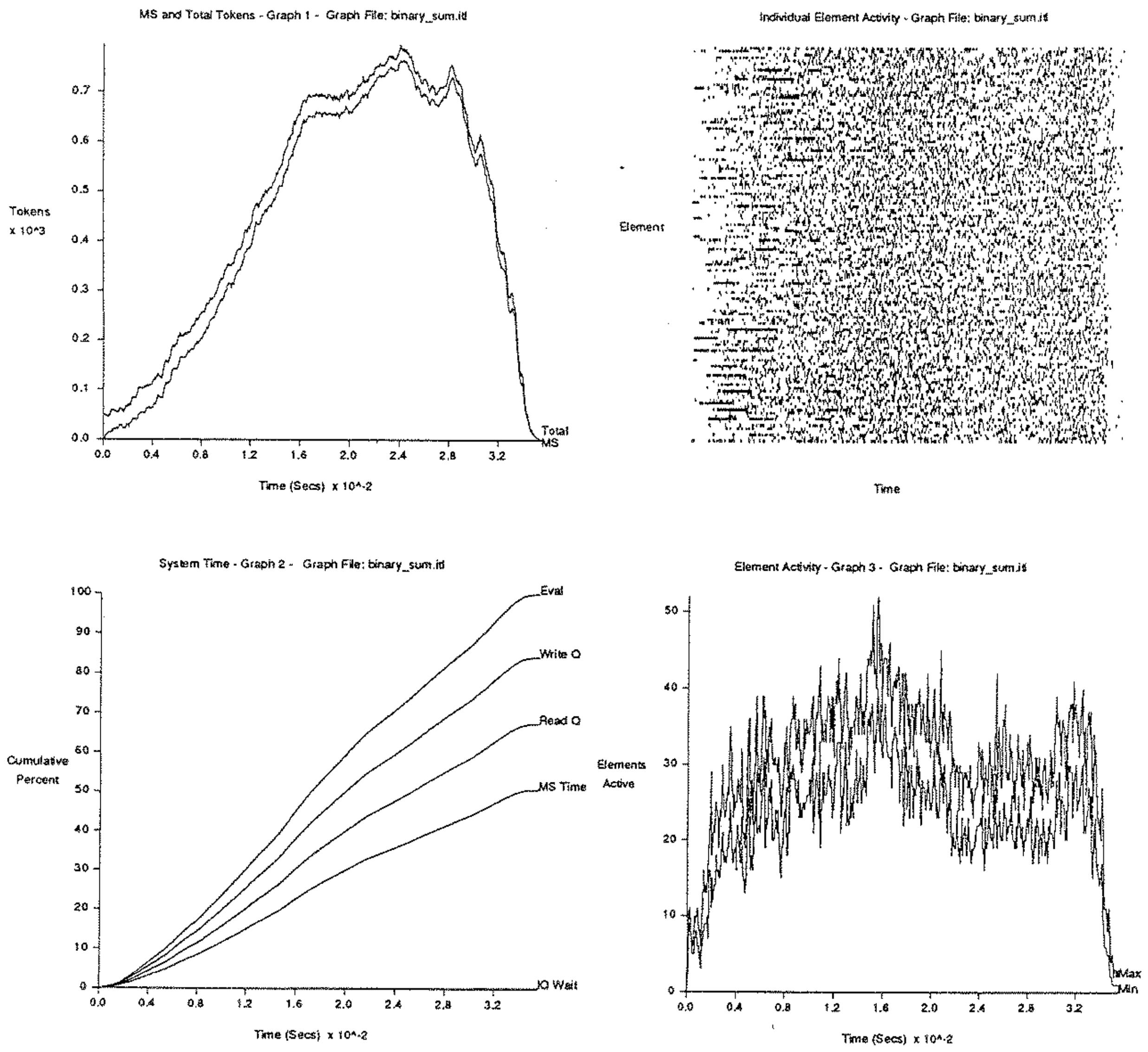


FIGURE 5.5 Activity plots for binary stream summation (advanced node set)

The activity traces for the three stream graphs, figures 5.3, 5.4 and 5.5, are for a single application of each graph (using the advanced/stream node set) to a single input stream of 24 elements. They are all clean graphs which finish with no tokens left in the matching unit/arc store, as indicated by the '`MS and Total Tokens`' plots. All three programs have recursive definitions which specify the solution as some operation on the *head* of the current stream and a result returned by a recursive call(s) with the *tail* of the current stream as input (`binary_sum` is a 'divide and conquer' algorithm which continually splits its input into two streams and adds the corresponding elements of these sub-streams). A trivial result is returned (either scalar or stream) when the current input is the empty stream.

The individual element activity picture gives a lot of information on the precise execution of a graph as it distributes over the multiprocessor. The three stream graphs all show horizontal streaks due to the pipelinig effect of the queued stream tokens. With `reverse`, the initial streaks reduce in length, while others can be seen increasing as tokens are moved from the input stream to an output stream of eventually the same length (these effects are concurrent because of the non-strict nature of stream *consing*). `Serial_sum` is a simpler graph than `reverse` but has similar horizontal streaks reducing in length as the input stream is absorbed until the recursion terminates.

`Binary_sum` has more nodes and generates more tokens at run time than `reverse` or `serial_sum`. Its activity is correspondingly denser and more distributed. The algorithm itself is not as simple or regular as `reverse` or `serial_sum`, which also helps to randomise the computation. Again, horizontal streaks can be detected in the activity picture. Compared to `reverse` and `serial_sum`, `binary_sum` shows a larger increase in the number of tokens generated than in potential or achieved concurrency, this is because a higher proportion of tokens are in the matching store; activity/concurrency is more dependent on how many *free* tokens are generated.

`Reverse` and `serial_sum` exhibit a diagonal pattern in their activity pictures because they use a computed copy number subgraph call with a maximum occurrence of one. This increments the copy number at every recursive call and the dynamic token addressing (unravelling) causes the PE numbers to also increment by one throughout the run. Imposed upon this pattern is the random static allocation used by the compiler at the top (and thus every) level. `Binary_sum` does not exhibit an obvious diagonal pattern because it has a maximum shared subgraph occurrence of two, causing subgraph calls to scatter and wrap around the element range more quickly.

All three graphs have discernable 'tails' in their token and element activity plots (graphs 1 and 3), which are the return sequences at the end of the recursions. During periods of 'stable' graph activity, such as in these tails, the change in the number of tokens over time is proportional to the integral of the element activity, since this integral represents the number of nodes fired. This effect can be observed over small intervals of most activity traces and is clearly evident during the tail of `serial_sum`. More often though, the effect is obscured when different graph regions become blurred by their overlapped results.

`Reverse` is written as a tail recursion without tail optimisation. Its tail performs no algorithmic computation, but merely returns a final result (a stream) from the deepest context, cleaning the graph as it does so. The tail overlaps with other computation, as the non-strict stream consing allows the result stream to be returned as it is created; it represents slightly less than 50% of the total execution time and exhibits high concurrency (at least initially) due to the highly concurrent nature of stream processing. This high concurrency can be clearly seen in the second half of the individual element activity picture for `reverse`.

`Serial_sum` is not a true tail recursion, as it does its summation in the tail, but a low concurrency tail region (with a concurrency between 1 and 3) is clearly evident as the final result is computed and returned. This tail represents about 25% of the entire run and is a major source of inefficiency in this graph. It exhibits a linear fall off in the number of tokens, nearly all of which are in the matching store. At the start of the tail, there are 72 tokens in the machine, being 3 for every level of recursion (one for the return address, one for the as yet unused condition, and one for the number to be added to the partial sum as it is returned). If this graph was rewritten as a tail recursion, the summation would be done in parallel with the call sequence and tail optimisation would remove the tail section altogether since the final result is then returned in one operation.

`Binary_sum` is not tail recursive and like `serial_sum` does its summation during the return sequence. It exhibits relatively high concurrency throughout the run and has a tail which falls away exponentially. As in the duplicate versus replicate discussion, there will be a break even point at which the larger binary summation graph becomes more efficient than the simpler serial summation because it has a similar $O(\log n)$ vs $O(n)$ execution time advantage. In this particular case, `binary_sum` has taken twice as long as `serial_sum` to process the same stream.

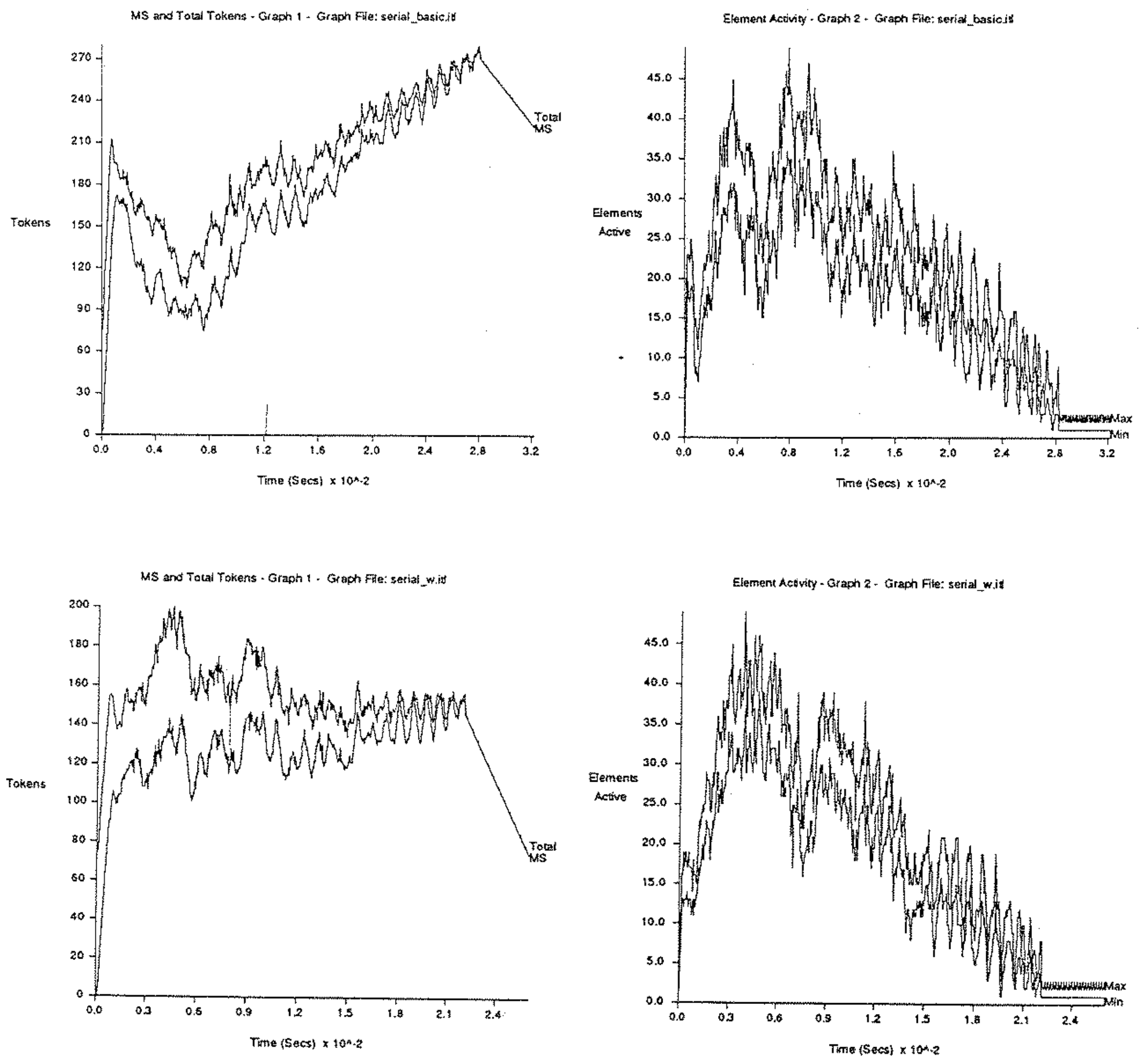


FIGURE 5.6 Token and element activities for serial stream sum (basic and extended)

To further illustrate the difference between the node sets, figure 5.6 shows the token and element activities for `serial_sum` using the basic and extended node sets. The most obvious difference is the non-clean nature of the simulations, with 216 and 72 tokens remaining in the matching store respectively. Also, more tokens are generated than for the advanced node set (table 5.1), with a higher proportion in the matching store (similar to the `binary_sum` simulation).

5.4.1 Effect on Graph Size

	reverse	serial_sum	binary_sum	if_lazy	if_eager
extended	21%	28%	29%	0%	4%
advanced	61%	56%	61%	56%	60%
switch/join	—	—	57%	—	—

TABLE 5.2 Static graph size reduction due to node set

Table 5.2 shows the reduction in graph size due to the use of the extended and advanced node sets for the stream test graphs. The reductions with the extended node set are due to the different expansions of shared conditional expressions where reentrant code is used (all examples used here are statically reentrant so that token queues can be pipelined safely through them; this can not generally be done in other dataflow architectures unless additional tagging code is used to separate contexts). The smaller extended code templates of §4.3.4.4 clearly show how these reductions arise. Also, the difference in size of the `stream-store (STS)` expansion between basic and extended node sets (§4.3.8.4) is significant in the static graph size reduction (many copies of `STS` are planted).

The figures for the advanced/stream node set are consistent for most stream graphs observed to date and represent a significant saving (~55-60%). They include the extended node set savings, as well as the extra savings brought about by use of the special stream matching functions. The code templates for the expansion of all stream functions were presented in §4.3.8.4. It is expected that enhancements of this nature, which lead to smaller and simpler machine graphs, will improve the speed of compilation and machine loading, and aid in the design of graph optimisation routines, although limited study has been done in these areas to date.

The switch/join solution for the binary summation, which uses the advanced node set and a more optimised, though complicated program structure, shows no obvious performance improvement over the original solution. This will not be true in general however, since the straight forward conditional expression, as used in all these graphs, lacks a significant optimisation when the same arc appears as an input more than once, i.e., such an arc need only be gated once, but presently the conditional code template will gate it each time it appears. Optimisations such as this are currently being added to the compiler.

	reverse	serial_sum	binary_sum	if_lazy	if_eager
extended	1%	17%	17%	0%	0%
advanced	61%	73%	67%	59%	57%
switch/join	—	—	63%	—	—

TABLE 5.3 Reduction in nodes executed due to node set

	reverse	serial_sum	binary_sum	if_lazy	if_eager
extended	5%	24%	22%	1%	1%
advanced	63%	73%	67%	59%	57%
switch/join	—	—	63%—		—

TABLE 5.4 Reduction in tokens used due to node set

Tables 5.3 and 5.4 show the results for changes in dynamic graph size, i.e., the number of nodes executed and the number of tokens used, due to different node sets. These results show generally similar trends to the static graph size figures, although they are more variable. In fact, the extended node set has not led to quite the same improvement as in the static graph size analysis. This is because certain nodes in the extended node set, in particular, ones that use the *prime* and *protect* matching functions, have more than one firing condition. This shows up in the number of nodes fired, but not in the static size of the graph. A similar phenomenon is observed with the advanced node set figures because of the nature of the special stream matching functions. Other factors which may lead to a difference between static and dynamic figures are code sharing, unravelling, and reentrancy due to looping or pipelining.

5.4.2 Effect on Concurrency

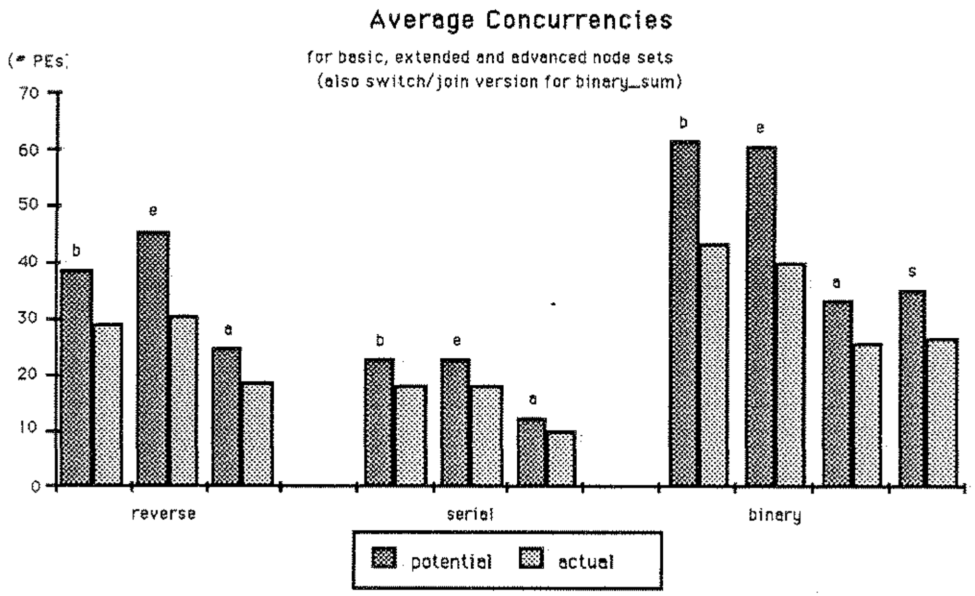


FIGURE 5.7 Changes in concurrencies (potential and actual) due to node set

Figure 5.7 shows the concurrency figures of table 5.1 graphically, while table 5.5 shows the percentage change in potential concurrencies with different node sets. For each of the three graphs there is little difference in potential concurrencies between basic and extended node sets, but both are reduced sharply by the advanced node set. This is to be expected because of the nature of the code templates used in stream graphs. For example, with the extended node set, fewer nodes are generated, but in a 'wider' distribution. This has the effect of reducing graph execution time, even though the over all actual and potential concurrencies may not alter significantly.

With the advanced node set, including the switch/join solution, the potential concurrency is decreased (table 5.5). This is mainly due to the number of nodes executed being greatly reduced (table 5.3). Clearly, care must be taken to ensure that reduced concurrency does not cancel the benefits of fewer nodes, i.e., we should always seek to reduce the *critical path* length. In this case, the loss of concurrency is less than the reduction in nodes fired, which has lead to reduced graph execution time (§5.4.3).

	reverse	serial_sum	binary_sum	if_lazy	if_eager
extended	-18%	-1%	2%	0%	3%
advanced	36%	45%	45%	32%	-85%
switch/join	—	—	43%	—	—

TABLE 5.5 Potential concurrency reduction due to node set

5.4.3 Effect on Execution Times

In the final analysis, it is execution time that must be considered as the critical performance factor. Table 5.6 shows how the simulated potential execution times vary with the node set used.

The improvements are not as great as in the number of nodes executed (table 5.3), since the concurrency is lower and the advanced stream nodes have greater execution times than the ones they replace. Never the less, the gains for the advanced node set are substantial (~35-45%), showing that a tailored node set can more than make up for any loss of concurrency inherent in its use.

	reverse	serial_sum	binary_sum	if_lazy	if_eager
extended	20%	23%	19%	1%	2%
advanced	40%	45%	34%	32%	76%
switch/join	—	—	32%	—	—

TABLE 5.6 Potential execution time reduction due to node set

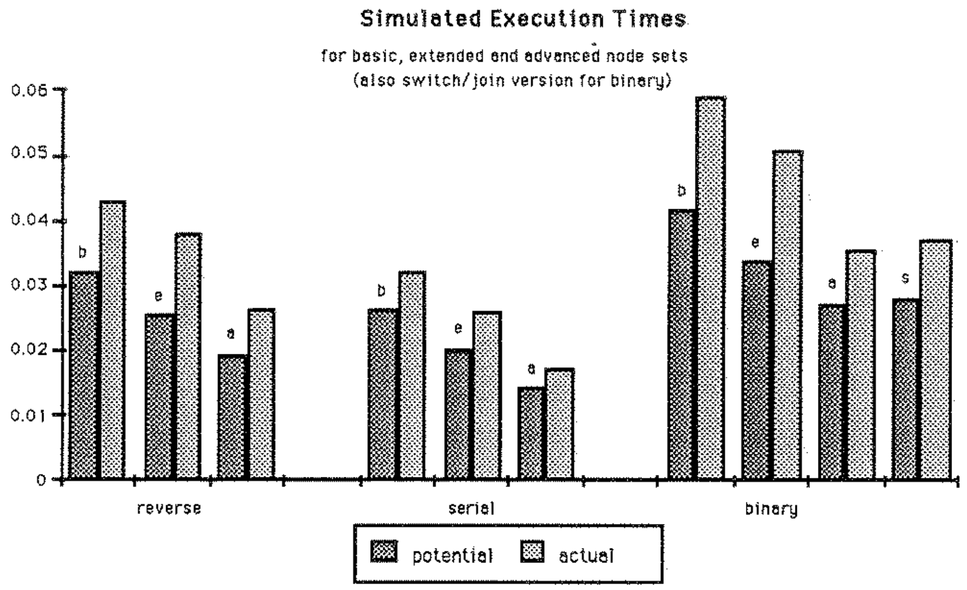


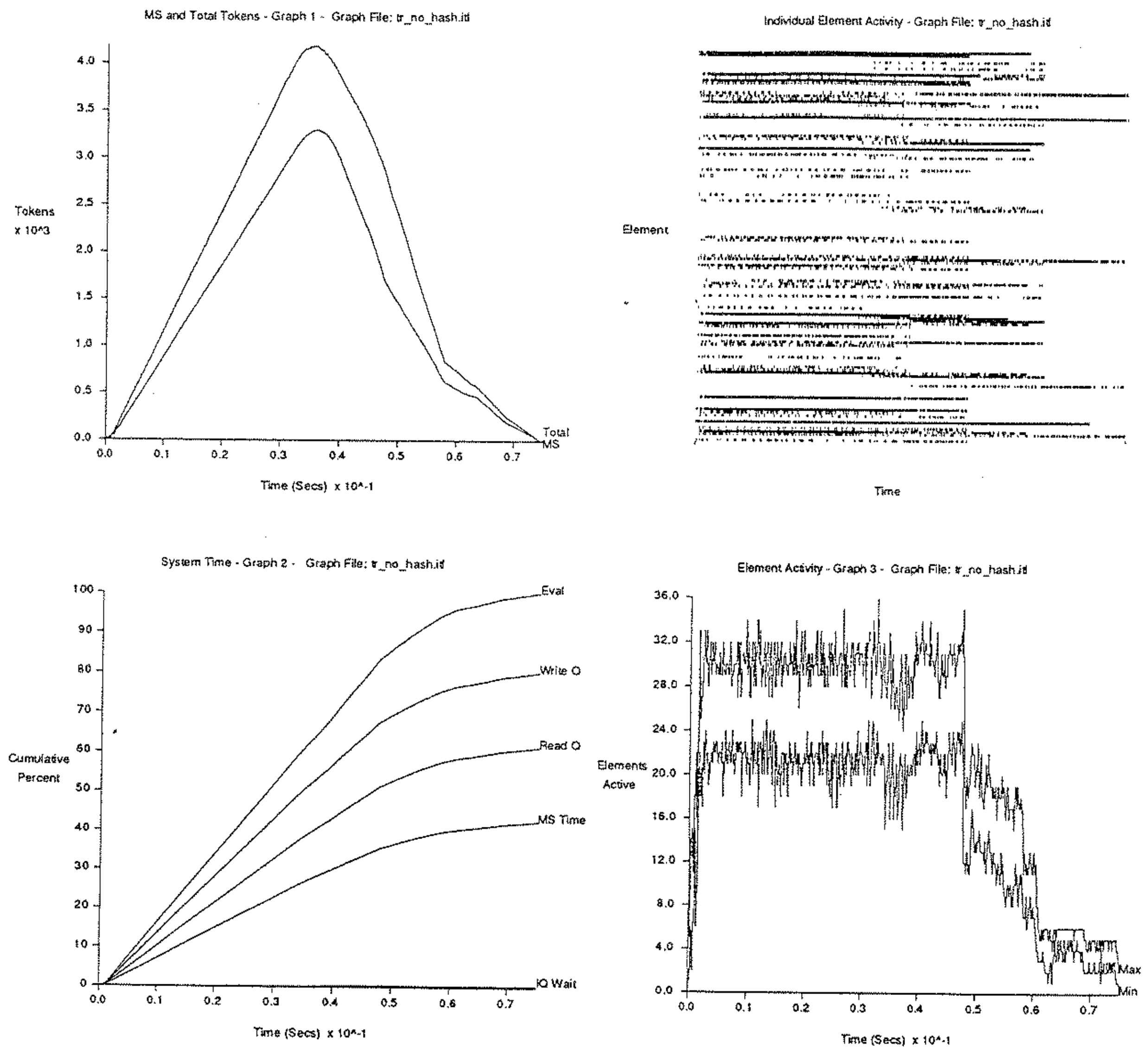
FIGURE 5.8 Changes in potential and actual execution times with node set

Figure 5.8 shows the execution time results graphically. There is little variation in the ratio of potential to actual execution times for these graphs, regardless of node set. In fact, this ratio depends greatly on graph allocation and machine utilisation. In this case, the machine utilisation is consistently low, due to the simulation using 128 processing elements, so that the potential to actual execution time ratio is rather high. A similar effect is seen in the concurrency graph of figure 5.7.

5.5 Unravelling

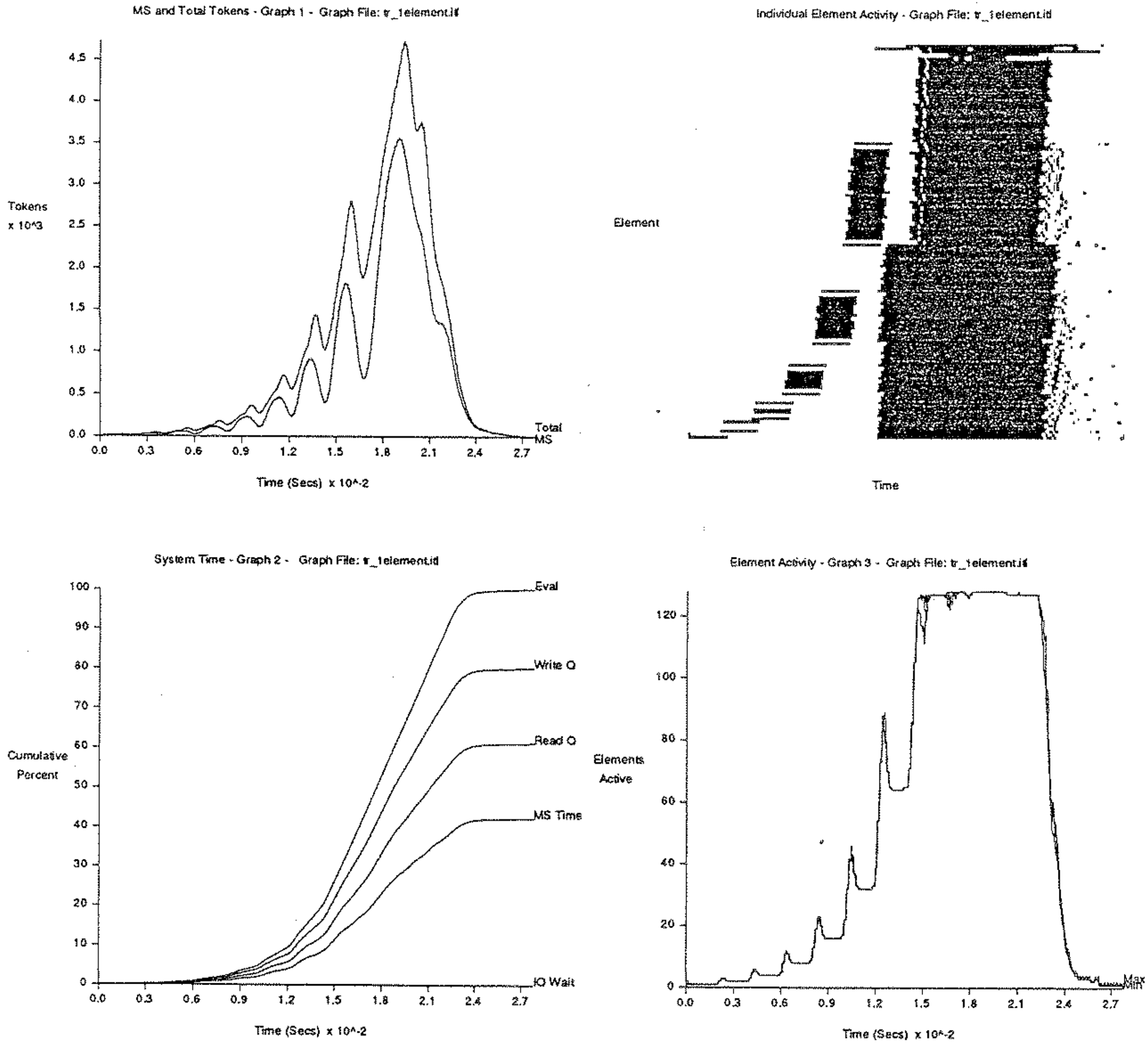
A modified trapezoidal integration program was used to study the effect of graph uravelling and related factors on run time performance. Several versions of the graph were used in order to analyse specific effects, selected listings are included in appendix A. Note that in most test graphs, output is done by directing tokens to a predefined console node (at address (1,-32,0), i.e. element 1, system node 32, input 0). This method has generally been preferred to the DL1 **write** statement since that statement would result in a resource sharer being planted which would tend to dominate graph size and obscure the effects under investigation.

Figure 5.9 shows the results of simulating the trapezoidal integration with dynamic machine addressing disabled (§4.3.7). This forces each instance of the recursive shared subgraph to execute in a randomly determined processing element distribution (the distribution being determined by the compiler). Graphs like this, which have fewer nodes than there are elements in the machine (81 vs 128 in this case), clearly rely on some form of dynamic distribution to increase their machine utilisation figure, although this requires extra copies of the graph to be sent to any processing elements which may be addressed at run time. The machine utilisation (MU) figure of 14.8% is correspondingly low in this simulation. As with the stream graphs of §5.4, the efficiency of this graph is limited by a rather low concurrency tail during the return sequence of the recursion.



Static size	(nodes/tokens)	81	4	
Dynamic size	(nodes/tokens)	30411	40377	
Processing elements		128		
Execution time (sec)	(pot/act)	0.005931	0.075085	
Concurrency	(pot/act)	240.2	19.0	(7.9%)
Instructions/Sec	(pot/act)	5127293	405020	
Machine Utilisation		14.8%		

FIGURE 5.9 Simulation of 'tr_no_hash.itl'



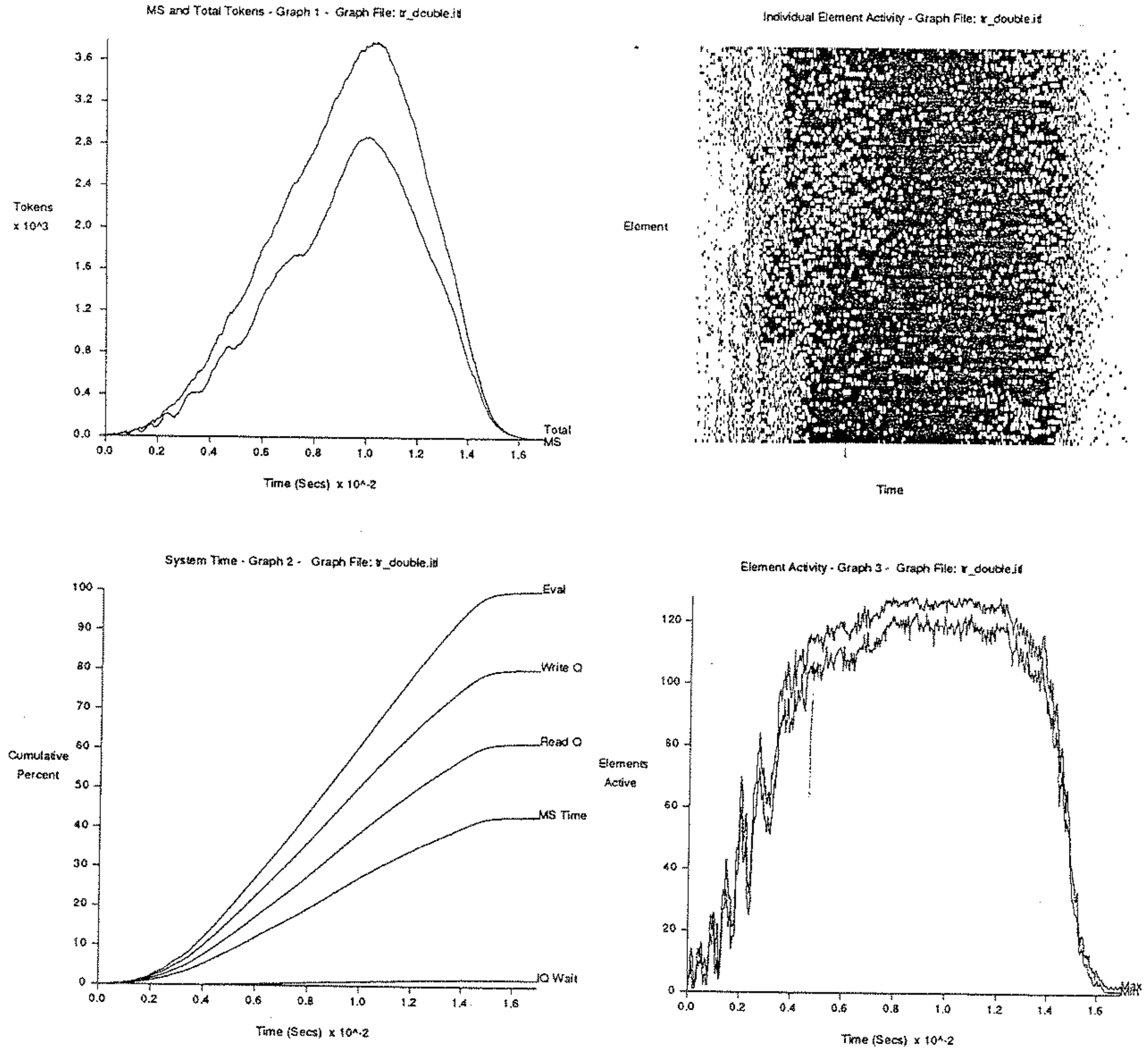
Static size	(nodes/tokens)	81	4	
Dynamic size	(nodes/tokens)	30411	40377	
Processing elements		128		
Execution time (sec)	(pot/act)	0.005931	0.027776	
Concurrency	(pot/act)	239.3	51.1	(21.4%)
Instructions/Sec	(pot/act)	5127293	1094874	
Machine Utilisation		39.9%		

FIGURE 5.10 Simulation of 'tr_1element.itl'

Figure 5.10 shows the effect of dynamic unravelling on this graph, this time with the random static PE distribution disabled by compiling the entire graph into one element. This makes the activity picture resemble a series of horizontal streaks spreading out over the PE range (each streak being one instantiation of the graph confined to a single processing element). The graph shows the 'tree like' effect of the computed copy number mechanism in generating tags or colours. The graph gradually expands its process space until it floods the entire machine; any extra activity (which can be virtually unbounded in a graph of this nature) merely generates higher queue occupancy while node evaluation catches up with the token traffic. The bands at the top and bottom of each processor group in the activity picture 'waves' are due to the exclusive-or used in the hashing algorithm for converting tags/static PE numbers into run time PE numbers (§4.3.7).

The graph generates exponentially increasing activity as it descends the doubly recursive call sequence. Its return sequence falls away exponentially but at a much faster rate since less computation is performed here. Each 'streak' exhibits 100% activity until a recursive call is made because destination nodes are in the same element. The usual case however, is that the compiler generates a random distribution for each call, this achieves a faster execution time and a greater machine utilisation due to higher concurrency within each instantiation.

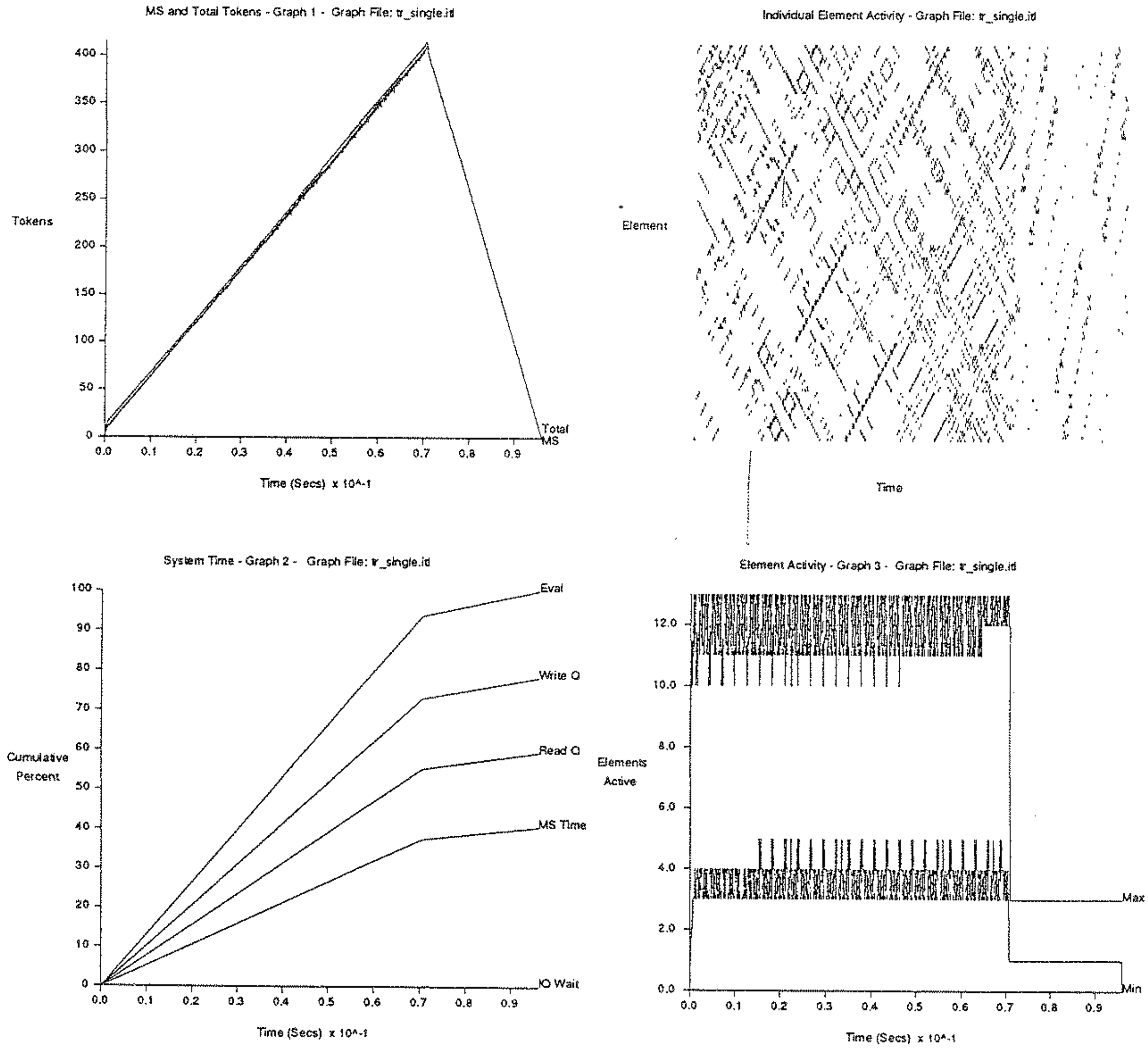
The figures for average actual concurrency (51.1) and machine utilisation (39.9%) are superior to those for the 'no hashing' case (19.0 and 14.8%), showing that for graphs of this nature, dynamic distribution (unravelling) between instantiations is more important than static distribution within instantiations. Note that the figures of 51.1 and 39.9% give a poor estimate of the true nature of this graph, since the machine was virtually 100% active for a large proportion of the time. It is common practice among dataflow researchers to use average figures like these, but they are really only a rough guide to true graph performance and should be used in conjunction with activity plots like those presented here where possible.



Static size	(nodes/tokens)	81	4	
Dynamic size	(nodes/tokens)	30411	40377	
Processing elements		128		
Execution time (sec)	(pot/act)	0.005931	0.016950	
Concurrency	(pot/act)	243.5	85.2	(35.0%)
Instructions/Sec	(pot/act)	5127293	1794138	
Machine Utilisation		66.6%		

FIGURE 5.11 Simulation of 'tr_double.itl', the unrestricted doubly recursive trapezoidal integration

In figure 5.11, the no hashing and one element restrictions are removed and the graph spreads out more randomly, evenly and quickly over the machine. The number of nodes fired, tokens used and the final result are all identical, but `tr_double` has a higher actual concurrency (85.2) and machine utilisation (66.6%). The 'bursty' activity of `tr_1element` has been smoothed out greatly in the unrestricted graph, although call and return sequences are still clear in the simulation.

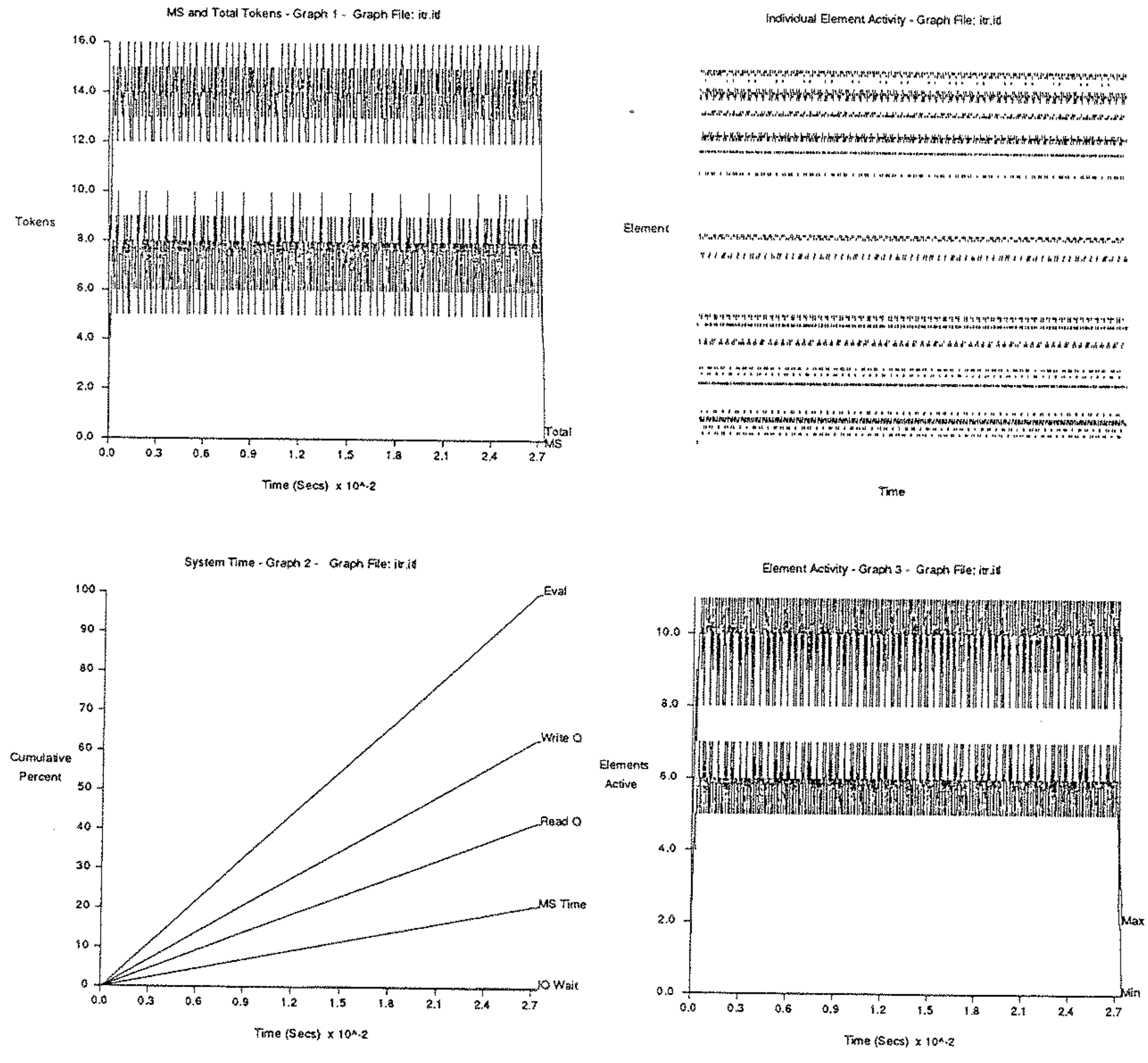


Static size	(nodes/tokens)	62	5	
Dynamic size	(nodes/tokens)	10835	14247	
Processing elements		128		
Execution time (sec)	(pot/act)	0.091999	0.096136	
Concurrency	(pot/act)	5.5	5.3	(95.7%)
Instructions/Sec	(pot/act)	117773	112705	
Machine Utilisation		4.1%		

FIGURE 5.12 Simulation of 'tr_single.itl', a singly recursive trapezoidal integration

Simulation of a singly recursive version of the trapezoidal integration using the same function, end points and `dx` interval, is shown in figure 5.12. The graph is tail recursive and sums its result in the call sequence. The call and return sequences are clearly visible in the execution profiles. Although many fewer nodes are executed (10835 vs 30411), the graph is still slower than the doubly recursive solution (0.09 vs 0.016 actual execution times) due to the much lower actual concurrency (5.3 vs 85.2).

The difference between these two solutions is even more obvious when one considers the potential concurrencies of 5.5 for the singly recursive solution, and 243.5 for the doubly recursive solution. The latter figure ensures that `tr_double` would execute even faster if more than 128 processing elements were available, whereas little speed up would be observed above 5 elements in the case of `tr_single`. The divide and conquer algorithm, with its $O(\log n)$ execution time is clearly superior to the singly recursive solution with an $O(n)$ characteristic, at least for this many integration intervals (200). To improve the poor potential concurrency of graphs like `tr_single` involves minimising the inefficiencies of shared subgraph calls and tagging overheads, this will be of prime concern in the next generation machine design.



Static size	(nodes/tokens)	32	4	
Dynamic size	(nodes/tokens)	6432	8442	
Processing elements		128		
Execution time (sec)	(pot/act)	0.025611	0.027423	
Concurrency	(pot/act)	7.6	7.1	(93.4%)
Instructions/Sec	(pot/act)	251138	234544	
Machine Utilisation		5.5%		

FIGURE 5.13 Simulation of 'itr.itl', an iterative trapezoidal integration

Figure 5.13 shows the simulation of `itr.itl`, a tagless, iterative version of the trapezoidal integration. Like `tr_no_hash`, `itr` does not unravel because no tagging is done, the matching store overhead (`MSTime`, graph 2) is correspondingly lower than in the previous cases. `Itr` has a low machine utilisation (5.5%), but executes far fewer nodes, with the result that the actual execution time is faster than `tr_single` (a 71% improvement) but still greater (by 62%) than `tr_double`. Again, there is a trade off between a simple, fairly efficient $O(n)$ graph (`itr.itl`) and a higher overhead $O(\log n)$ graph (`tr_double.itl`). There will be a break even point above which the 'divide and conquer' recursive solution is faster, but simpler iterative solutions must not be ruled out for certain problems, although this is often impossible to resolve at compile time. Clearly, the cost of token tagging must be weighed heavily against the efficiency of the final graph. In this study, `tr_single` has achieved little speed up due to unravelling, the pipelining effect being far more important. We suspect that this could be true for a large proportion of iterative/singly tail recursive graphs [2, 3].

As a further example, consider the results of simulating the shared FFT graph of §5.6 with and without unravelling, table 5.7 and figure 5.14. The actual execution time for `sffth.itl` (which is not unravelled) is more than double that of `sfft.itl` (which is unravelled). Achieved concurrency, execution rate and machine utilisation are all more than halved in `sffth.itl`. This suggests that the benefits of unravelling are applicable to a wider range of graphs than just the recursive ones seen so far. In fact, anywhere that tagging is used to achieve code sharing, or any other form of reentrancy, should benefit from unravelling. This result is discussed further in §5.9.3, where it is shown that any sequence of queued tokens can be uniquely tagged in an attempt to generate more concurrency.

5.6 Overheads of Tagging

In this section, the overheads involved with tagging are examined in more detail, by considering several versions of a 16 point complex Fast Fourier Transform graph.

Graph	Static Size		Dynamic Size		Concurrency	
	(nodes)	(tokens)	(nodes)	(tokens)	(potential)	(actual)
fft.itl	1760	144	1648	2416	41.2	33.3 (80.8%)
sfft.itl	706	128	3776	5056	76.7	49.8 (64.9%)
sffth.itl	706	128	3776	5056	77.3	21.9 (28.3%)

Graph	Execution Time (sec)		Instructions/Sec		Machine Utilisation
	(potential)	(actual)	(potential)	(actual)	
fft.itl	0.001434	0.001774	1149634	929236	26.0%
sfft.itl	0.002474	0.003809	1526582	991466	38.9%
sffth.itl	0.002450	0.008660	1541539	436033	17.1%

TABLE 5.7 Summary of the FFT simulations

Table 5.7 summarises the three FFT simulations, while figure 5.14 shows the token and element activity plots obtained. There are actually two complete data sets being piped through the graph in these simulations, so that two instantiations of the graph are running together. This randomises and increases activity to a more appropriate level for this study.

The first graph, `fft.itl`, is a straight forward 'flat' FFT with no shared code at all, existing entirely on the optimised zero copy number/tag level. `sfft.itl` is the same program but with all subgraphs defined as shared subgraphs. In the FFT program (appendix A), a basic FFT 'butterfly' operation is called both directly and indirectly a total of 32 times. By making all subgraphs shared, there is only one actual copy of the butterfly, so it is vital that the 32 instances be unravelled dynamically (in the unshared version, this is effectively done at compile time by randomly distributing 32 in line copies of this subgraph). The effect of inhibiting dynamic unravelling is shown by the simulations of `sffth.itl`, where the hashing of destination element addresses is disabled (see also §5.5).

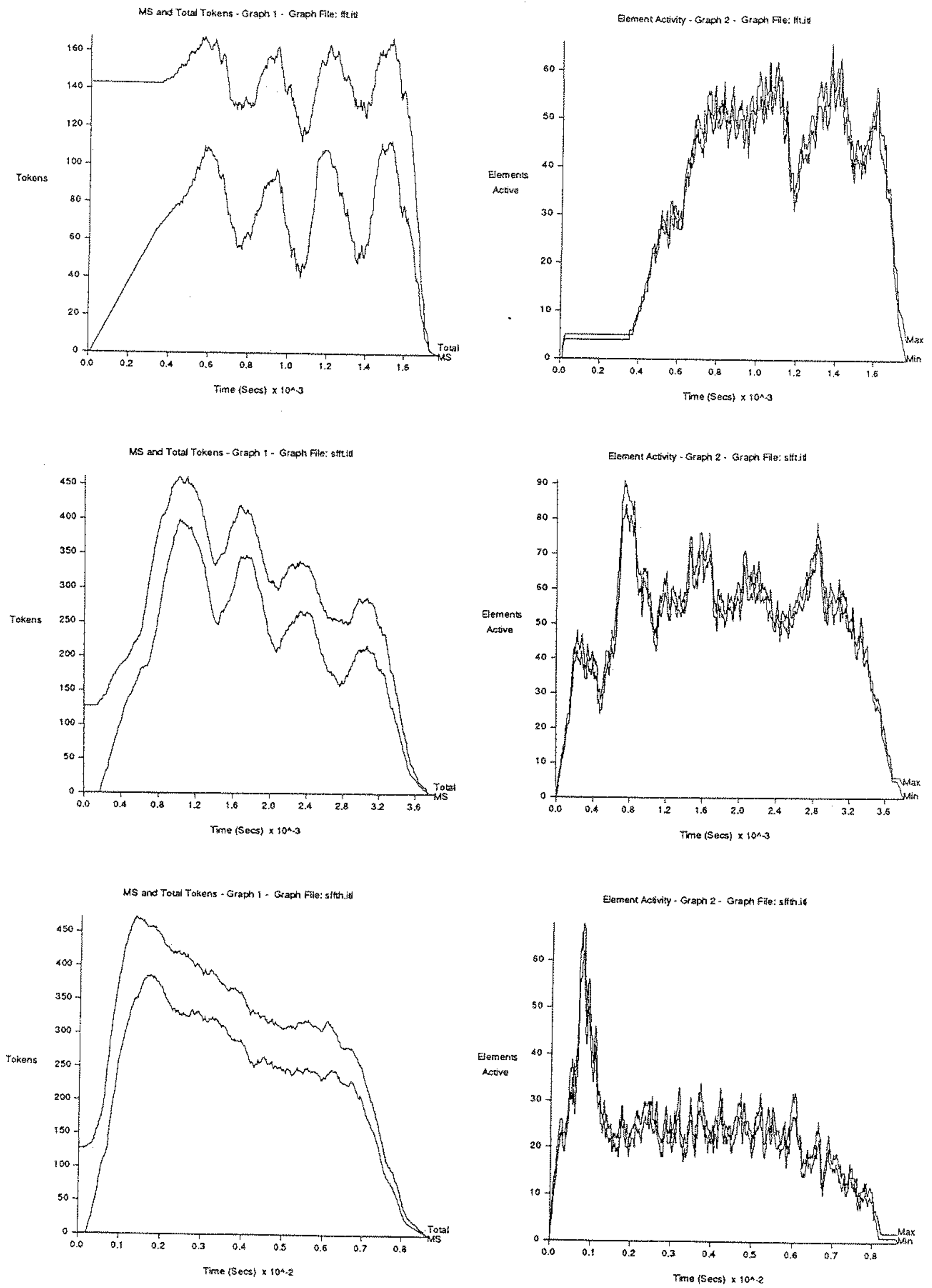


FIGURE 5.14 Simulations of the Fast Fourier Transform graphs

Note that `fft` executes fewer nodes than there are in the graph itself because the compiler generates multiple copies of priming tokens that are directed at duplicate trees, the duplicate trees are then bypassed by these tokens. `Fft` executes about twice as fast as `sfft`, while `sffth` is twice as slow as `sfft`. The shared versions both execute 3776 nodes (using 5056 tokens) compared to 1648 nodes (using 2416 tokens) for the unshared graph, these extra nodes and tokens (a 56% increase) are all involved in the shared subgraph call and return code. The static size of `sfft` is much smaller than `fft` because of the code sharing, but this has little or no influence on the actual number of nodes executed in this case.

The overheads of tagging are not just due to extra nodes being required to manipulate the tags, but also show up in the average token size used in the graph (tagged tokens are 32 bits longer then untagged tokens, although this may change in future implementations). In `fft`, all tokens are 5 words long (untagged, 32 bit reals), while the total traffic sent through the machine is 12080 words. For `sfft`, 2.5% of all tokens are 5 words long (untagged priming tokens), 87.3% are 7 words long (tagged, 32 bit reals), and the remaining 10.1% are 9 words long (tagged 64 bit environment tokens, used for shared subgraph argument returns). Of course, there is also a matching store overhead incurred by tagged matches (§2.9), which further degrades the performance of `sfft`. For `fft`, matching unit operations took 26.0% of the total time, whereas this figure rises to 40.5% for `sfft`.

Not all of these overheads are present in other architectures. For example, the Manchester dataflow machine uses fixed size tokens which always carry a tag field. Also, the Manchester matching store is not optimised in any way; it always performs a pseudo associative access based on node number and colour, which has been shown to be a limiting factor in the performance of that machine [29]. The figures presented here suggest that the shared subgraph call and return mechanisms used at RMIT to date, require careful attention in order to optimise them. A similar argument applies to dynamic loop translations (§4.3.6.2), since the same tagging mechanism is also used there.

5.7 Machine Utilisation

Machine utilisation was studied by varying the number of processing elements as well as the concurrency in the simulation of the graph `newtr.itl`. This is a version of the trapezoidal integration which had the number of x-axis intervals varied in order to control its potential concurrency. Figure 5.15 shows the machine utilisation curves obtained.

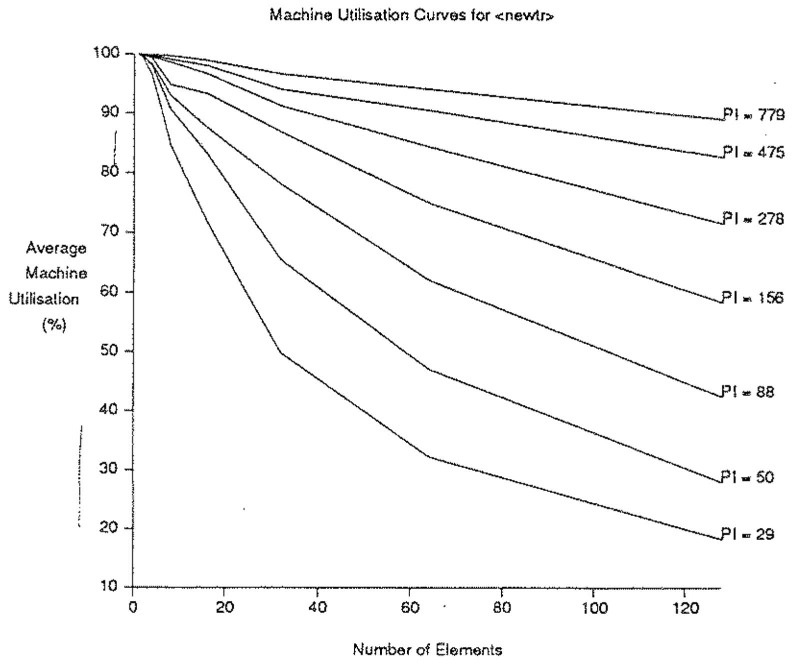


FIGURE 5.15 Machine utilisation vs number of PEs and potential concurrency (PI)

Results from these simulations have been used to compute an excess concurrency ratio required to give machine utilisations of 80% and 90% (for this graph) :-

Average Machine Utilisation	80%	90%
Excess Concurrency Ratio	3.0 ($\sigma = .3$)	6.7 ($\sigma = .5$)

where the Excess Concurrency Ratio = Potential Concurrency ÷ Number of PEs

The excess concurrency ratio is a useful figure of merit for a dataflow architecture, since it shows how much of the concurrency in a graph is absorbed by intra-processing element and intra-network activity, i.e., extra concurrency is required to keep these pipelines full. Actual concurrency is also degraded by time delays, such as tokens waiting longer than necessary in matching stores, or delays in network throughput.

The figures obtained can be accounted for by the length of the PE pipeline (three stages, being Input Queue, Processor, and Output Queue) which absorbs some graph concurrency (DFSIM does not model the communications network pipeline, but does account for delays due to destination clashes). The difference between PE pipeline stage times, with processing time dominating in these simulations, will expand the apparent size of this pipeline, so that an excess concurrency ratio greater than three would be expected for a high machine utilisation. Also, as the potential concurrency figure is an average over the entire run, then for a graph with any kind of variation over time (i.e., most graphs), a still higher excess concurrency ratio would be required. Finally, machine utilisation can never equal one for a graph which has a minimum activity less than `Number of PEs × Pipeline Stages` at any time in its execution.

5.8 Distribution of Machine Activity

This section shows how the various modules of the simulated machine distribute the run time work load under the simulation conditions described in §2.9.

	Percentage of Active Time
IQ Read	19.0 ($\sigma = 1.4$)
OQ Write	19.0 ($\sigma = 1.4$)
Matching Unit	37.7 ($\sigma = 9.8$)
Node Evaluation	23.6 ($\sigma = 7.2$)
Network Wait	0.6 ($\sigma = 0.2$)

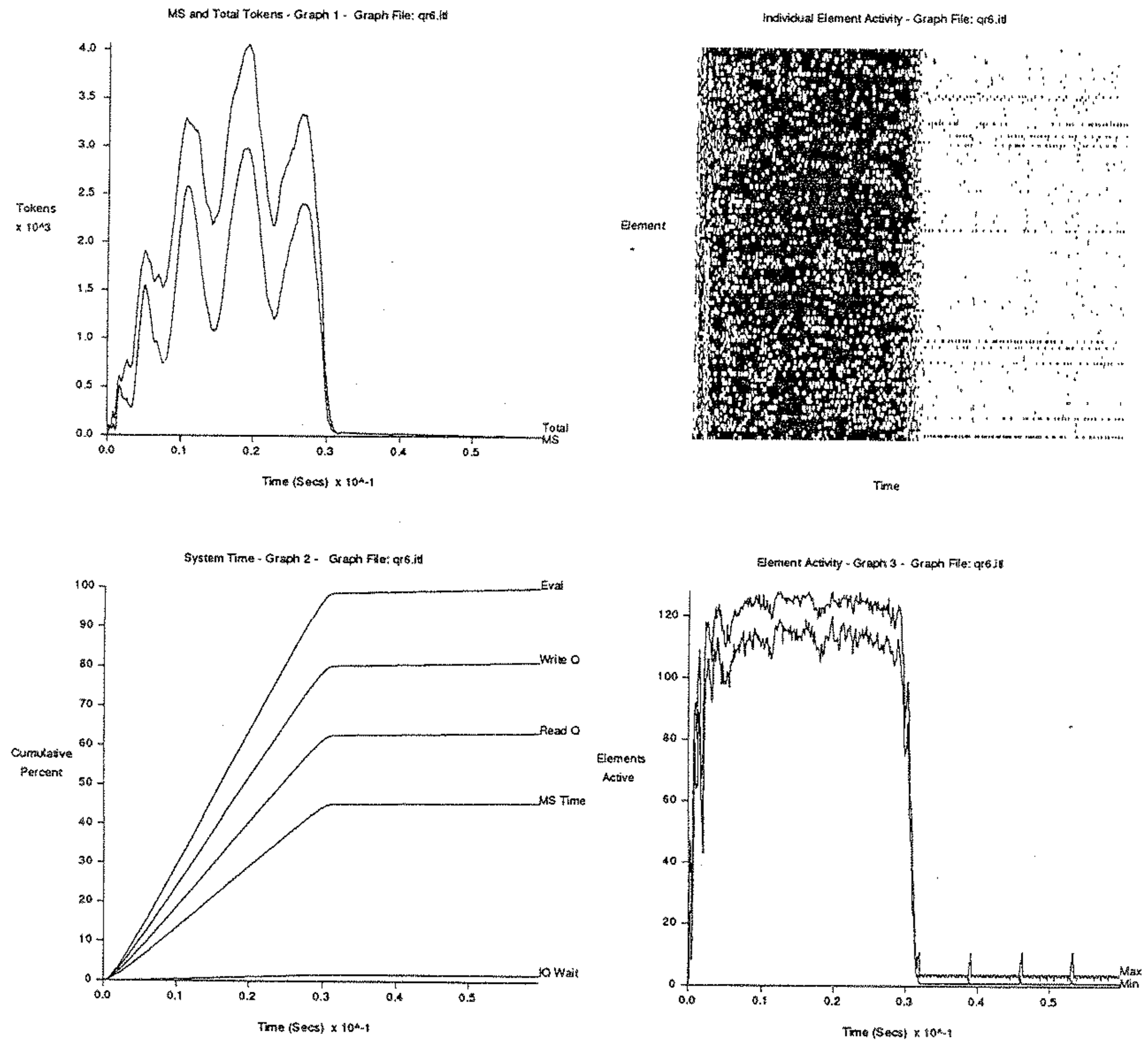
TABLE 5.8 The distribution of machine activity

Table 5.8 shows aggregate results from all the test graphs used in this chapter. Matching unit operations accounted for between 26% and 50% of total machine activity in all runs, while node evaluation accounted for between 16% and 32%. In graphs with a high proportion of tagged tokens, the matching store activity was also high (>40%); it was the number of tagged matches that affected this figure the most. Other factors, like the proportion of two input nodes executed (usually between 30 and 50%), had less influence.

Node execution was most expensive in graphs with a high proportion of floating point operations, e.g., 32% in the unshared FFT graphs, which also had the lowest matching overhead (26%) due to a total lack of tagged tokens and despite executing a relatively high proportion of two input nodes (46%). The other results were more consistent, e.g., network wait (the time lost due to destination clashes in the communications network) was always very low and directly proportional to the amount of traffic in the machine. No attempt was made to simulate the effects of localising graph segments, so that there was a consistent 127/128 probability that result tokens would be destined for external elements. This also shows up in the '`Locality of tokens`' graph (see figure 2.10). Input queue read and output queue write times were always roughly equal, which is consistent with the functional nature of the dataflow machine graphs used. In cases where there are many more priming tokens than final results, these figures could be expected to differ somewhat, although this effect is usually swamped by the number of tokens generated internally.

5.9 Other Effects

5.9.1 Effect of Sequential Code Segments



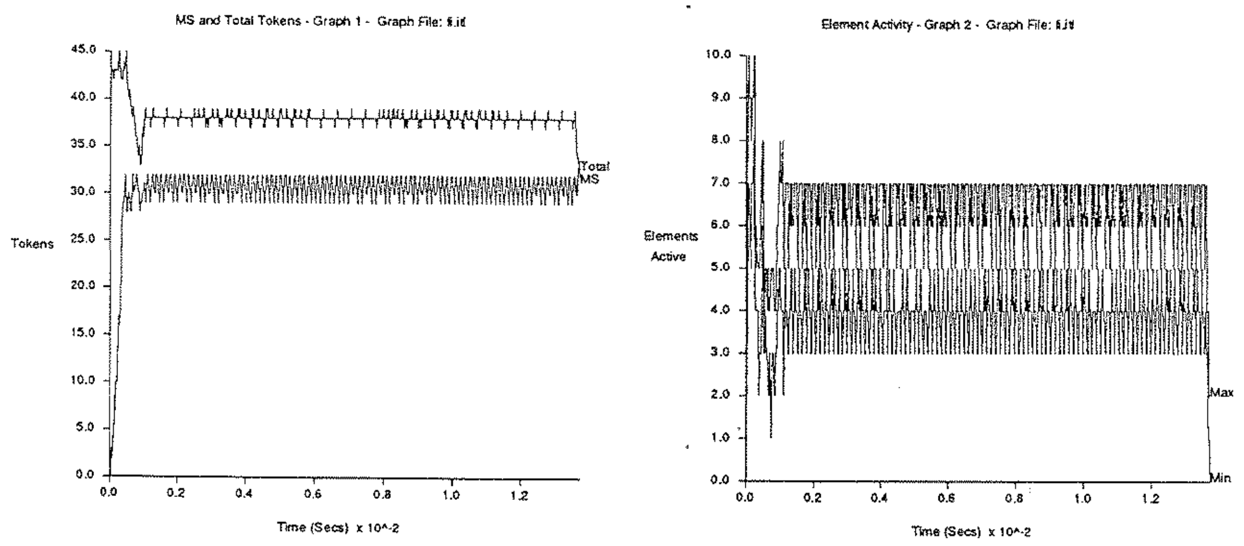
Static size	(nodes/tokens)	864	74	
Dynamic size	(nodes/tokens)	78834	105244	
Processing elements		128		
Execution time (sec)	(pot/act)	0.031188	0.059744	
Concurrency	(pot/act)	114.1	59.6	(52.2%)
Instructions/Sec	(pot/act)	2527735	1319532	
Machine Utilisation		46.5		

FIGURE 5.16 Simulation of 'qr6', a recursive solution to the 6 queens problem

In figure 5.16, the solution to the six queens problem is seen to generate large concurrency, but the execution time is dragged out by a factor of two due to the very slow, sequential resource sharer which sends results to the output stream. This must be done sequentially to prevent solutions from overwriting each other (the speed of the output device is not the limiting factor in this example, rather, it is the overhead of the calls to the resource sharer). In fact, there are only four solutions to this problem, but the eight queens case has 192 solutions and the output bottle-neck becomes a severe problem (as does any other largely sequential graph segment). Naturally, any traditional solution suffers the same difficulty in writing the results to an output file, but at the same time, a uniprocessor could take 120 times as long to compute the solutions (the graph sustains an average potential concurrency of ~120 during the recursion phase).

One approach that has been taken to sequential output is to output data with tags intact and do any sorting outside the machine. This relieves the machine of the burden of sorting tagged data and allows the data to be output as it is generated, but will not lead to greater concurrency/speed of output operations if the output device is the limiting factor. For example, the languages LAPSE and P5, on the prototype Manchester dataflow machine, perform sequential output by using the index field of the tag to hold the current sequence number for the output stream. Tokens destined for a given output stream have their index fields set to the current stream position by accessing a counter token; further serialising code exists between different write statements. Similar details apply to input operations, where tags on input tokens must be preset to appropriate (incrementing/unique) values.

5.9.2 Operations on Open Sequences



Static size	(nodes/tokens)	24	44	
Dynamic size	(nodes/tokens)	2421	3157	
Processing elements		128		
Execution time (sec)	(pot/act)	0.008659	0.013763	
Concurrency	(pot/act)	7.9	5.0	(62.9%)
Instructions/Sec	(pot/act)	279584	175905	
Machine Utilisation		3.9		

FIGURE 5.17 Simulation of 'fi.itl', a digital filter using sequences and queueing

The simulation of the digital filter `fi.itl`, figure 5.17, (listing in appendix A), shows how a very simple and efficient solution can be found to handling data *sequences*, such as encountered in signal processing applications, on the hybrid architecture. Although relatively low concurrency is present in this simple example, a typical signal processing graph would contain many such modules, thus ensuring a more even computation spread and higher concurrency and machine utilisation. The regularity of the graph and its inherently sequential algorithm make it well suited to pipelined execution (for real time or sequential data processing at least). One reason for this this is the high proportion of data to control tokens. In this case, 90.5% of all tokens (2857) were 32 bit reals, with only 9.5% (300) being used for control purposes.

An unravelled, tagged token approach would not improve the performance of this graph, indeed the extra overheads involved would degrade the performance, as shown by the shared FFT graph of §5.6. In general though, data dependencies may not be present at all between the elements of a queue and it should therefore be possible to process those elements concurrently. The next section outlines an approach that uses tagging to achieve this effect.

5.9.3 Unravelling of Queues

It is recognised that many dynamic dataflow implementations (e.g., the Manchester and MIT tagged token systems) have rejected the use of queueing in favour of tagging for reasons of hardware simplicity and performance efficiency. Tagging can be used to extract maximum concurrency from dataflow graphs by eliminating any unnecessary data dependencies implied by queued tokens, e.g., loop/recursion unfolding and run time distribution of shared code. However, it was shown in §5.5 that simple iterative solutions can easily outperform their tail recursive counterparts due to the efficiency of pipelined execution on the common ring structured dataflow processing element. Also, queueing fits in well with the sequential nature of operations like i/o and filtering, where extra tag manipulating code is not required.

Clearly, both the queued static and tagged dynamic dataflow systems have particular advantages and disadvantages, and the hybrid system should be able to combine the best features of both. To illustrate this ability, a method will be described that allows the application of a function to the elements of a token queue to be unravelled (i.e., each application is unfolded onto a separate machine region).

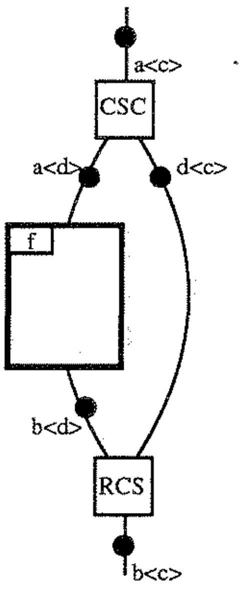


FIGURE 5.18 Unravelling the processing of a token queue

In figure 5.18, the machine graph macros `CSC (create-and-set-colour)` and `RCS (restore-colour-sequentially)` allow the token queue on arc `a`, with colour `c`, to be 'scattered' and the application of subgraph `f` to unravel. `CSC` tags each input token with a unique colour, `d`, in the same manner as a `create-colour/set-colour` pair does on shared subgraph entry (§4.3.5.2). A copy of the unique colour used, which is different for each token in the queue, is tagged with the original input's colour, and sent to the second output (`d<c>`). Subgraph `f` is now instantiated in distinct machine regions due to the dynamic addressing scheme used (assume that each application of `f` is independent in the scope presented here, i.e., there are no data dependencies). `RCS` receives an ordered queue of colours on its second input that indicates the order in which the outputs of `f` must be requeued. `RCS` performs this requeueing and restores the original input colour.

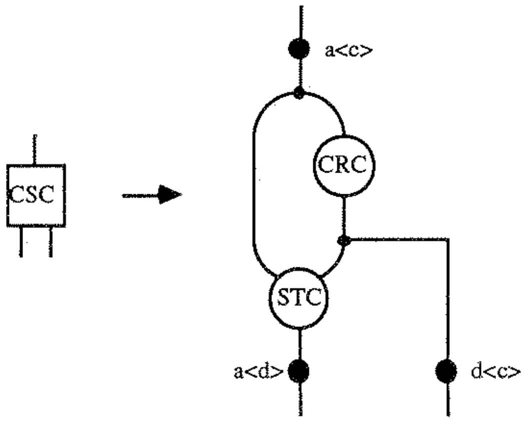


FIGURE 5.19 (a) CSC using the current node set

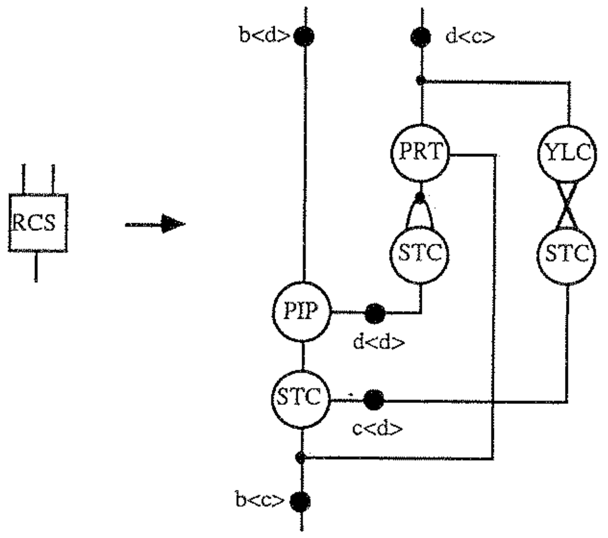
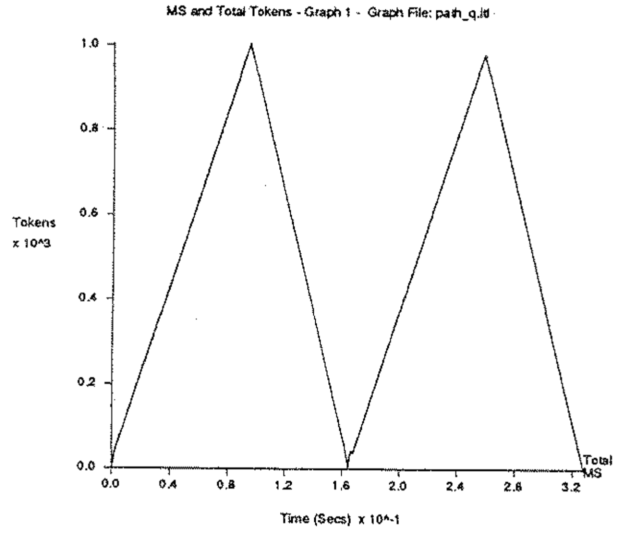
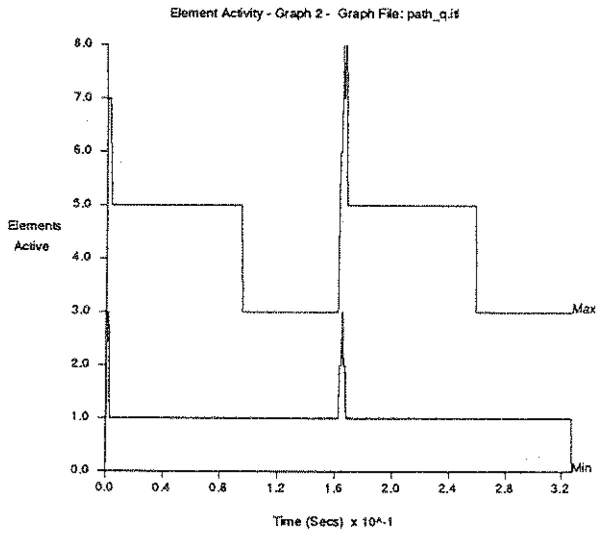


FIGURE 5.19 (b) RCS using the current node set

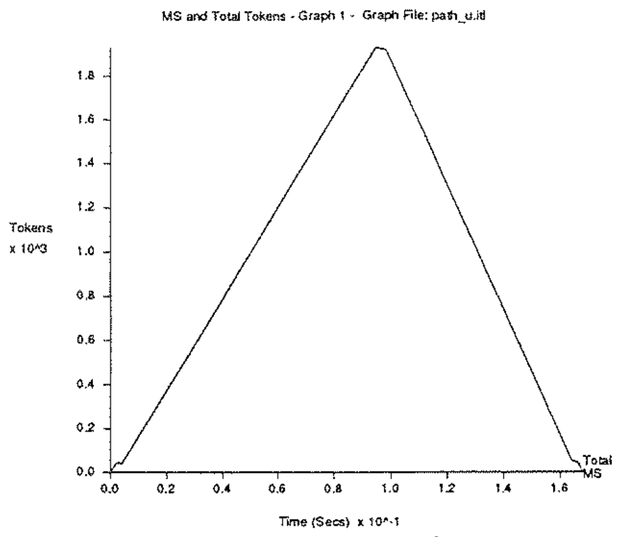
Figure 5.19 shows implementations of `CSC` and `RCS` using the existing node set that are clean, non-strict and reentrant. An ordered queue corresponding to the unique colour sequence generated by `CSC` is maintained on the data input of the `protect (PRT)` node in `RCS`. This node provides non-strict operation by allowing the first colour through, before any output has been produced. The uniquely coloured outputs of `f` wait to be selected on the data input of the `pass-if-present (PIP)` node. They are then tagged with their original colour by a `set-colour (STC) node`. As each output of `RCS` is produced, the `protect` node is retriggered, allowing the selector for the next output colour through. All other nodes are involved in the manipulation of colours, e.g., the `yield-colour (YLC)/set-colour (STC)` pair performs a swap colour with value operation (this should be implemented as one node).

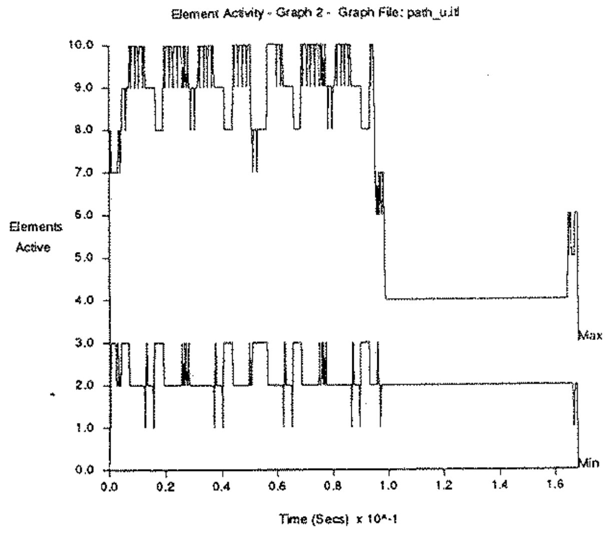




Static size	(nodes/tokens)	42	2	
Dynamic size	(nodes/tokens)	17064	21078	
Processing elements		128		
Execution time (sec)	(pot/act)	0.327225	0.327522	
Concurrency	(pot/act)	2.0	2.0	(99.9%)
Instructions/Sec	(pot/act)	52148	52100	
Machine Utilisation		1.6%		

FIGURE 5.20 (a) Simulation of path_q.itl





Static size	(nodes/tokens)	55	2	
Dynamic size	(nodes/tokens)	17098	21112	
Processing elements		128		
Execution time (sec)	(pot/act)	0.167866	0.168474	
Concurrency	(pot/act)	3.9	3.9	(99.8%)
Instructions/Sec	(pot/act)	101807	101440	
Machine Utilisation		3.0%		

FIGURE 5.20 (b) Simulation of path_u.itl

Figure 5.20 shows the simulation of a ‘pathological case’, in which the function `f()` is defined as `id(500 - id(n))`, where `id()` is a recursive identity function that returns the value of its input, `n`, in time $O(n)$. Two priming tokens are sent into this graph to make up the input queue, their values being 490 and 10 respectively. The queued outputs of the graph are therefore 10 and 490 in that order. When no unravelling shell is used around `f()` (`path_q.itl`), the time taken is almost twice as long as in the unravelled case (`path_u.itl`). The concurrency and instruction execution rate (both potential and achieved), and machine utilisation of `path_u` are all twice those of `path_q`. The activity plots show that the results are effectively computed serially for `path_q` and concurrently for `path_u`. The first result (490) emerges half way through the `path_q` simulation, whereas both results are returned in the same minimal time for `path_u`. The reason for the difference is that the results of the first call to `id()` for each input token must be kept ordered by `path_q`, but not by `path_u`. Thus, in `path_q`, the input to `id(500 - 10)`, which is the output of `id(10)`, is delayed until the output of `id(490)` is computed. The numbers are such that the total execution time should be the same for both results, however this is prevented by the artificial data dependency between the intermediate results in the queued graph.

If `CSC` and `RCS` were implemented as single nodes (`CSC` is trivial, whereas `RCS` would require a new matching function), then there would be no problem at all in switching between tagged and untagged code in a hybrid graph. The possibilities of this mode of operation are still under investigation, a process made simple by the unique nature of the hybrid architecture. One important use of `CSC/RCS` is in the protection of non-reentrant code. By ensuring that graph segments like `f()` in the above example are never instantiated more than once in a given context, it is possible to implement them using non-reentrant code. Thus, the lazy and eager merge macros of ch. 4 could be left in the simplified, non-reentrant form (i.e., using a simple nondeterministic merge) if they were always surrounded by a `CSC/RCS` shell.

Chapter 6
SUMMARY AND CONCLUSIONS

6.1 Simulation and the Design Process

Care must be taken when interpreting the results of simulations like the ones presented in this thesis, since the results are only as good as the fundamental figures that the simulation is based on. In particular, simulation based on timing figures from the prototype emulator is questionable. These figures are largely inappropriate for a dedicated, discrete processing element design because of the radically different architectures envisioned. Never the less, they provide a feel for the factors most likely to influence over all performance before any serious design decisions are made. With this in mind, development of the next generation processing element will involve use of the new sixteen element emulator hardware to more accurately simulate the precise architectures of interest.

One of the more important factors yet to be fully analysed, is the effect of imbalance between the execution rates of different stages in the PE pipeline. While it seems reasonable to make average throughput equal for all stages, the effects of different node evaluation times, different matching rates (due to matching functions, tagging, etc.), different result distribution times (due to variable numbers of outputs) and so on, must all be considered in new designs. The most advanced work done in this area to date is by the Manchester group, who have demonstrated the need for extra buffering at the output of their matching unit, and who are currently studying the advantages of an optimised input queue that should improve the utilisation of the matching store resources [29].

6.2 Recommended Modifications to DL1

Although compilers for new languages are being developed for the RMIT system (§2.1), DL1 will remain as a very useful system development tool. Its unique combination of high and low level features allows its use as a powerful, arbitrary graph construction tool. The newer languages, with greater support for complex data structures, higher level functions, functional and logic programming, etc., are expected to become the preferred user development tools; they will provide a means of benchmarking and source level compatibility already accepted by other researchers in the field. As the DL1 code generator provides a rather direct and efficient translation from program graph to machine code, the language should be suitable as an intermediate form for other compilers and preprocessors. A cut down version of the DL1 compiler is being adapted for this purpose.

In order to facilitate this use of DL1 as an intermediate language for other compilers, and to allow even greater control over machine graph generation, it is recommended that a node() function call be added to the language. This function would accept as inputs the type of node to be laid and a list of arcs or identifiers to be linked to the inputs of the node. It should also be possible to specify special matching functions, literal data, absolute physical location (to override the compiler generated graph address), etc.. This style of graph description is actually a symbolic assembly language for the dataflow machine, that is superior to the lower level ITL format described in appendix D.

Currently, identifiers can only be 'typed' in the parameter lists of subgraphs, but a more satisfactory approach would be to allow the type of an identifier to be specified anywhere that the identifier is used or defined. This would allow statements such as

```
a: int + b: int -> c: real;
```

Since DL1 allows identifiers to be used before they are defined, type specification is sensible even inside expressions, as in this example.

To specify result types statically, it would be possible to have a *cast* operator, as in the C programming language. This can be done using a similar format by allowing entire expressions, not just identifiers, to be 'typed', e.g.,

```
(a: int + b: int): real -> c;
```

In this example, consistency checks can be applied to 'c' to check any following uses or declaration that may occur. To aid program brevity and readability type specification should be optional in all cases, with ambiguities being flagged by the compiler and resolved by the programmer with appropriate declarations. When a declaration is made, it should only be necessary on the first occurrence of an identifier, be that a definition or a use.

With this scheme type coercion can be more easily controlled by the programmer, removing much of the need for run time checking and coercion. This would result in a very significant saving in the current PE emulator code, since for many nodes type checking and coercion are more expensive than the node function itself. However, run time type checking also serves as an important error detection mechanism and should not be abandoned lightly. The best solution would be to have an extra class of nodes that perform type checking without coercion. Such an instruction set is used by the MIT group, where arithmetic nodes are available with and without automatic type coercion [9].

Although subgraph parameters must currently be 'typed' in DL1, this is not actually essential to the compiler, since parameter source and destination nodes can be checked for consistency when the subgraph is called. Even if a program has no type specifications at all, the compiler can still deduce the types of many identifiers by their definitions; any remaining checking must be deferred to run time.

The compiler is currently lacking in its ability to handle complex data types. This may be acceptable if DL1 is to be used solely as an intermediate form for other languages, but a facility for arrays and records is considered a minimum requirement for general program development. Records in particular would be very helpful in simplifying software design, since the current informal record/expression list constructions can lead to complex and verbose coding techniques. Arrays are more of a problem because of their unique implementation requirements, the current level of support being limited to direct access to a single indexed storage node per PE. The next generation machine will include distributed deferred access structure store units for efficient, random access data storage and retrieval. Any implementation of arrays or records should be made compatible with the DL1 'no unnecessary typing' philosophy. Therefore, arrays and records should be declared without the need to specify component type(s); it is only the *structure* of the data that is necessary for program graph generation, and not the base *types* of the structure's components.

The advantages of unravelling have been discussed in §5.5, however there is also the disadvantage of needing copies of dynamically addressable graph segments in all processing elements participating in their execution. Thus, shared subgraphs and other unfolded code blocks are not really shared at all during loading and at run time, although they do reduce the static graph size. In the original FLO system specifications no provision was made for unravelling, so that shared subgraphs were loaded just once and actually were shared at run time with a corresponding reduction in concurrency and performance (see §5.6, and refs [2, 41]).

6.3 Suggestions for Further Research

The dataflow project at RMIT is expected to provide benefits in many related research areas. For example, the 16 element emulation facility designed around the dual M68020 PE prototype emulator and a high speed 16 way buffered delta switch network (§2.1, [52]), will provide an ideal test bed for general multiprocessor research. To this end, a global bus with shared memory access would be a worthwhile addition to this hardware.

Areas of immediate concern to the dataflow project will be the development of a multiprocessor operating system, establishment of on line mass storage, and completion of a high speed interface to the CSIRO host computer. Also, the development of the next generation processing element will require substantial research into ways of building a practical, very high performance dataflow machine using state of the art hardware technology.

One of the interesting possibilities for future processing element designs is the prospect of combining the matching store and structure store units. This is suggested by the similarity of their operations, particularly in the area of deferred access queueing. It is especially relevant to the hybrid implementation because the hybrid matching store already supports such queues in the form of tokens queueing on the same input of a two input node. The difference between the queued matching operation and deferred structure store accessing is that the latter provides indexing, data persistence, and satisfies all deferred accesses with one data item (as opposed to matching with just one waiting token). In fact, the hybrid matching unit will also satisfy several waiting tokens with one matching data item since this is precisely the operation of the proposed *complex protect* matching function (§3.4). The matching unit could also be made to support indexed structure storage with suitable modifications to its memory organisation.

Of particular relevance to the efficiency of the emulation rig is research into combined dataflow/control flow machines, which have the potential of combining the best features of both architectures [32]. In a combined machine, the granularity and functionality of the node set is entirely arbitrary, unlike most dataflow node sets that are fine grain for reasons of graph stability (for even distribution of activity) and high concurrency. Large grain node functions would be ideal for execution on the emulator, since they can exploit the speed of the von Neumann processors in each processing element, free of the significant overheads currently involved in the communications of data tokens. The combined model actors, or nodes, also communicate through matched tokens (both data and control tokens are available), but the ratio of computation to communication is usually far higher than in a low level dataflow model. Control tokens take the form of messages and semaphores that enable the synchronisation of communicating processes for determinate shared memory accessing, etc.. Data tokens have their usual dataflow interpretation.

Consideration must also be given to means of very low level optimisation of dataflow based execution. Can the significant performance improvements achieved in von Neumann architectures through the use of caches, instruction prefetching, 'RISC' concepts, etc., be applied to dataflow computers? Many of these techniques rely on the predictability of program counter control flow, even to the extent that some state of the art processors attempt to predict the outcome of instructions that alter this flow (e.g., Branch Target Lookahead). Although the highly asynchronous nature of dataflow computing appears to preclude many optimisations of this nature, research is certainly warranted in this area. In addition, the potential for highly concurrent computing that dataflow machines provide opens up a completely new field of research into specialised optimisation techniques which requires much attention.

6.4 Conclusions

A hybrid dataflow implementation has been described, together with machine and language features designed to take advantage of its unique characteristics. In particular, ways to optimise the use of token queueing have been outlined, including the use of hybrid matching functions and corresponding features in the programming language DL1 (e.g., reentrant code generation, queues and streams, etc.). Additional features such as an informal record mechanism (generalised expression lists), improved code templates, and eager and lazy conditional evaluation have made DL1 a practical dataflow language for the purposes of this and future research. Also, the principles of tagged token dataflow evaluation, including machine graph unravelling, have been successfully applied to the RMIT system with quantitative simulation results confirming the expected performance improvements generated by these techniques.

Appendix A
TEST PROGRAMS

A.1 DL1 Source Listings

This appendix includes source listings of selected test programs used in this thesis. Several limitations of DL1 are clear in these examples, e.g., the lack of arrays and named records is immediately obvious from the listings of Recursive_Queens and Fast_Fourier. Also note that DL1 sacrifices some elegance for low level control, e.g., the numerous compiler options (see also appendix B), and primitive statements like **prime**, **switch**, etc.. Never the less, it has proven to be an excellent tool for research at both program and machine graph levels, since it combines the convenience of structured programming with a very necessary degree of low level machine graph control. However, use of DL1 as a development language is expected to decline when the compilers described in the introduction of chapter 2 are commissioned, although a simplified derivative of the DL1 compiler will be used as an intermediate form for the new languages.

```
{ A program to find all solutions to the six queens problem using multiple recursion. }

[w+]

program recursive_6queens;

{
        Note:-
                        This program is based on a solution found in ref [18].

                        All subgraphs are functional (some return dummy results), this is
                        a hangover from the sequential solution which keeps a running sum
                        of the results found and the number of placements tried. This is not
                        possible in the recursive parallel search solution. Never the less, the
                        results of all function calls are still used as signals to maintain clean code.

                        This code suffers from the lack of formal, named records and arrays,
                        e.g. 'mark' uses a bitstring as an array of booleans and 'write_array'
                        uses an informal record of 6 integers as an array.
}

        shared subgraph try(rdn,ldn,cols: boolean; r0,r1,r2,r3,r4,r5,row,col: integer): boolean;

                subgraph next_try(rdn,ldn,cols: boolean; r0,r1,r2,r3,r4,r5,row,col: integer): boolean;

                        subgraph write_board(r0,r1,r2,r3,r4,r5: integer): boolean;

                                subgraph row(pos: integer): char;
                                begin { row }
                                        if pos = 0 then 'Q . . . . .' else
                                        if pos = 1 then '. Q . . . .' else
                                        if pos = 2 then '. . Q . . .' else
                                        if pos = 3 then '. . . Q . .' else
                                        if pos = 4 then '. . . . Q .' else
                                        if pos = 5 then '. . . . . Q' else '? ? ? ? ? ?'
                                        -> row;
                                end { row };

                                subgraph ww(r0,r1,r2,r3,r4,r5: char): boolean;
                                begin
                                        { A resource sharer to protect the output stream from garbled solutions }
                                        on when(r0,r1,r2,r3,r4,r5) then label(enable) -> set;
                                        { 'enable' grants the resource }
                                        on enable then r0,r1,r2,r3,r4,r5 -> r0g,r1g,r2g,r3g,r4g,r5g;

                                        { now strip the solution of its colour }
                                        yield(r0g) -> r0d,null;
                                        yield(r1g) -> r1d,null;
                                        yield(r2g) -> r2d,null;
                                        yield(r3g) -> r3d,null;
                                        yield(r4g) -> r4d,null;
                                        yield(r5g) -> r5d,null;

                                        writeln(output,r0d,chr(10),r1d,chr(10),r2d,chr(10),
                                                r3d,chr(10),r4d,chr(10),r5d,chr(10),chr(10)) -> ack;
                        yield(set) -> setd,setc;
                        { 'setd' releases the resources }
                        protect setd with ack;
                        setcopy(setd,setc) -> ddd;
                        setdest(ddd,ddd) -> null; { return the release signal }
                  end;

            begin { write_board }
                  { Print the solution, discard the result }
                  ww(row(r0),row(r1),row(r2),row(r3),row(r4),row(r5)) -> null;
            end { write_board };

      begin { next_try }

            { the code branches here }
            switch row < 5
            then rdn,ldn,cols,r0,r1,r2,r3,r4,r5,row,col
            -> rdnt,ldnt,colst,r0t,r1t,r2t,r3t,r4t,r5t,rowt,colt
            else null,null,null,r0f,r1f,r2f,r3f,r4f,r5f,null,null;

            { true branch… go to the next row }
            rowt + 1 -> nrow;
            try(rdnt,ldnt,colst,r0t,r1t,r2t,r3t,r4t,r5t,nrow,0) -> null;
            try(rdnt,ldnt,colst,r0t,r1t,r2t,r3t,r4t,r5t,nrow,1) -> null;
            try(rdnt,ldnt,colst,r0t,r1t,r2t,r3t,r4t,r5t,nrow,2) -> null;
            try(rdnt,ldnt,colst,r0t,r1t,r2t,r3t,r4t,r5t,nrow,3) -> null;
            try(rdnt,ldnt,colst,r0t,r1t,r2t,r3t,r4t,r5t,nrow,4) -> null;
            try(rdnt,ldnt,colst,r0t,r1t,r2t,r3t,r4t,r5t,nrow,5) -> null;

            { false branch… a solution has been found, print it }
            write_board(r0f,r1f,r2f,r3f,r4f,r5f) -> null;

            on row then false -> next_try;
      end { next_try };

      subgraph mark (rdn,ldn,cols: boolean; row,col: integer; val: boolean)
                        -> (rdno,ldno,colso: boolean);
      begin { mark }
            if val
            then setbit(rdn,row-col+5)
            else clearbit(rdn,row-col+5)
            -> rdno;

            if val
            then setbit(ldn,row+col)
            else clearbit(ldn,row+col)
            -> ldno;

            if val
            then setbit(cols,col)
            else clearbit(cols,col)
            -> colso;
      end { mark };

      subgraph advance(rdn,ldn,cols: boolean; r0,r1,r2,r3,r4,r5,row,col: integer): boolean;

            subgraph write_array(r0,r1,r2,r3,r4,r5,i: integer; val: integer)
                                    -> (r0o,r1o,r2o,r3o,r4o,r5o: integer);
            begin { write_array }
                  { r0 to r5 are treated as an array of integers }
                  if i=0 then val else r0 -> r0o;
                  if i=1 then val else r1 -> r1o;
                  if i=2 then val else r2 -> r2o;
                  if i=3 then val else r3 -> r3o;
                  if i=4 then val else r4 -> r4o;
                  if i=5 then val else r5 -> r5o;
            end { write_array };

      begin { advance }
            { place the queen and recurse }
            write_array(r0,r1,r2,r3,r4,r5,row,col) -> r0x,r1x,r2x,r3x,r4x,r5x;
            mark(rdn,ldn,cols,row,col,false) -> rdni,ldni,colsi;
            next_try(rdni,ldni,colsi,r0x,r1x,r2x,r3x,r4x,r5x,row,col) -> advance;
      end { advance };

      subgraph test(rdn,ldn,cols: boolean; row,col: integer): boolean;
      begin { test }
            testbit(cols,col) and
            testbit(ldn,row+col) and
            testbit(rdn,row-col+5) -> test;
      end { test };

begin { try }
      { test if safe to place queen }
      if test(rdn,ldn,cols,row,col)
      then advance(rdn, ldn, cols, r0,r1,r2,r3,r4,r5, row, col)
      else false -> try;
```

```
        end { try };

begin { 6queens }
        { initialise board (no queens placed) }
        prime true -> rdn, ldn, cols;
        prime 0 -> r0,r1,r2,r3,r4,r5;
        { recurses 6 ways at every level, solutions die as soon as an unsafe placement is found }
        try(rdn, ldn, cols, r0,r1,r2,r3,r4,r5, 0, 0) -> null;
        try(rdn, ldn, cols, r0,r1,r2,r3,r4,r5, 0, 1) -> null;
        try(rdn, ldn, cols, r0,r1,r2,r3,r4,r5, 0, 2) -> null;
        try(rdn, ldn, cols, r0,r1,r2,r3,r4,r5, 0, 3) -> null;
        try(rdn, ldn, cols, r0,r1,r2,r3,r4,r5, 0, 4) -> null;
        try(rdn, ldn, cols, r0,r1,r2,r3,r4,r5, 0, 5) -> null;
end { 6queens }.
```

```
{
        A 16 point complex FFT program. Derivations of this graph have
        been used extensively in testing the new 16 element emulator hardware.
        The FFT connectivity is established in the parameter lists.
}
[w + ]

program Fast_Fourier_Transform;

constant
        ZERO            =       0.0;
        PION8           =       0.39269908;
        PION4           =       0.78539816;
        PI3ON8                  =       1.17809725;
        PION2           =       1.57079633;
        PI5ON8                  =       1.96349541;
        PI3ON4                  =       2.35619449;
        PI7ON8                  =       2.74889357;

        subgraph fft(alpha, a, b, c, d: real) -> (ao, bo, co, do: real);
        begin
                sin(alpha)              -> sina;
                cos(alpha)              -> cosa;
                d * sina                -> dsina;
                c * cosa                -> ccosa;
                d * cosa                -> dcosa;
                c * sina                -> csina;
                a + ccosa + dsina       -> ao;
                b - csina + dcosa       -> bo;
                a - ccosa - dsina       -> co;
                b - dcosa + csina       -> do;
        end;

        subgraph fft2(f0, fi0, f1, fi1: real) -> (fo0, fio0, fo1, fio1: real);
        begin
                fft(ZERO, f0, fi0, f1, fi1) -> fo0, fio0, fo1, fio1;
        end;

        subgraph fft4(f0, fi0, f1, fi1, f2, fi2, f3, fi3: real)
                        -> (fo0, fio0, fo1, fio1, fo2, fio2, fo3, fio3: real);
        begin
                fft2(f0, fi0, f1, fi1)          -> z0, zi0, z1, zi1;
                fft2(f2, fi2, f3, fi3)          -> z2, zi2, z3, zi3;
                fft(ZERO, z0, zi0, z2, zi2)     -> fo0, fio0, fo2, fio2;
                fft(PION2, z1, zi1, z3, zi3)    -> fo1, fio1, fo3, fio3;
        end;

        subgraph fft8(f0, fi0, f1, fi1, f2, fi2, f3, fi3, f4, fi4, f5, fi5, f6, fi6, f7, fi7: real)
        -> (fo0, fio0, fo1, fio1, fo2, fio2, fo3, fio3, fo4, fio4, fo5, fio5, fo6, fio6, fo7, fio7: real);
        begin
                fft4(f0, fi0, f1, fi1, f2, fi2, f3, fi3)        -> z0, zi0, z1, zi1, z2, zi2, z3, zi3;
                fft4(f4, fi4, f5, fi5, f6, fi6, f7, fi7)        -> z4, zi4, z5, zi5, z6, zi6, z7, zi7;
                fft(ZERO, z0, zi0, z4, zi4)             -> fo0, fio0, fo4, fio4;
                fft(PION4, z1, zi1, z5, zi5)            -> fo1, fio1, fo5, fio5;
                fft(PION2, z2, zi2, z6, zi6)            -> fo2, fio2, fo6, fio6;
                fft(PI3ON4, z3, zi3, z7, zi7)           -> fo3, fio3, fo7, fio7;
        end;

        subgraph fft16(f0, fi0, f1, fi1, f2, fi2, f3, fi3, f4, fi4, f5, fi5, f6, fi6, f7, fi7,
                        f8, fi8, f9, fi9, f10, fi10, f11, fi11, f12, fi12, f13, fi13, f14, fi14, f15, fi15: real)
                -> (fo0, fio0, fo1, fio1, fo2, fio2, fo3, fio3, fo4, fio4, fo5, fio5, fo6, fio6, fo7, fio7, fo8,
                        fio8, fo9, fio9, fo10, fio10, fo11, fio11, fo12, fio12,
                        fo13, fio13, fo14, fio14, fo15, fio15: real);
        begin
                fft8(f0, fi0, f1, fi1, f2, fi2, f3, fi3, f4, fi4, f5, fi5, f6, fi6, f7, fi7)
                        -> z0, zi0, z1, zi1, z2, zi2, z3, zi3, z4, zi4, z5, zi5, z6, zi6, z7, zi7;
                fft8(f8, fi8, f9, fi9, f10, fi10, f11, fi11, f12, fi12, f13, fi13, f14, fi14, f15, fi15)
                        -> z8, zi8, z9, zi9, z10, zi10, z11, zi11, z12, zi12, z13, zi13, z14, zi14, z15, zi15;
                fft(ZERO, z0, zi0, z8, zi8)     -> fo0, fio0, fo8, fio8;
                fft(PION8, z1, zi1, z9, zi9)    -> fo1, fio1, fo9, fio9;
                fft(PION4, z2, zi2, z10, zi10)  -> fo2, fio2, fo10, fio10;
                fft(PI3ON8, z3, zi3, z11, zi11) -> fo3, fio3, fo11, fio11;
                fft(PION2, z4, zi4, z12, zi12)  -> fo4, fio4, fo12, fio12;
                fft(PI5ON8, z5, zi5, z13, zi13) -> fo5, fio5, fo13, fio13;
                fft(PI3ON4, z6, zi6, z14, zi14) -> fo6, fio6, fo14, fio14;
                fft(PI7ON8, z7, zi7, z15, zi15) -> fo7, fio7, fo15, fio15;
        end;

begin { Fast_Fourier_Transform }

        { test input is a unit step function }

        { the input imaginary part }
        on start
        then 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
        -> ai0, ai1, ai2, ai3, ai4, ai5, ai6, ai7, ai8, ai9, ai10, ai11, ai12, ai13, ai14, ai15;

        { the input real part }
        on start
        then 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
        -> a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15;

        { now call fft, the test results are discarded }
        fft16(a0, ai0, a8, ai8, a4, ai4, a12, ai12, a2, ai2, a10, ai10, a6, ai6, a14, ai14,
                a1, ai1, a9, ai9, a5, ai5, a13, ai13, a3, ai3, a11, ai11, a7, ai7, a15, ai15)
        -> null, … ; { 16 nulls }

        prime true -> start;
end.
```

```
{ A 'Z-domain' bandpass filter program that uses tokens as state variables }

[0:15]          { put this graph in 16 elements }

program BandPassFilter;

constant
        a =  2.56920E-01;
        b =  3.00000E+00;
        c =  5.77264E-01;
        d =  4.21794E-01;
        e =  5.62998E-02;

begin
        { these definitions generate unit time delays }
        x -> z2;
        z2 -> z4;
        z4 -> z6;
        z6 -> z8;
        z8 -> z10;
        z10 -> z12;
        c*z2-d*z4+e*z6+x -> u;
        u*a-b*z8+b*z10-z12 -> (1,-32,0); { result to console }

        { initialise the state variables, assume input was zero for negative time }
        prime 0.0,0.0 -> z2,z4,z6,z8,z10,z12;

        { feed the input with 100 points (i) of a unit step (x) }
        prime 0 -> i;
        switch i < 100
        then i+1, 1.0 -> i,x;
end.
```

```
{ A doubly recursive trapezoidal integration program }

{o2,2}          { set occurrences for double recursion }
{w+}            { extended code generation turned on }

program tr_double;

        shared subgraph area(a, b, dx: real): real;

                subgraph f(x: real): real;
                constant
                        pi = 3.141592654;
                        mean = 0.0;
                        sd = 1.0;
                begin
                        1.0/(sd*(sqrt(2.0*pi)))*exp(-0.5*sqr(x-mean)/sqr(sd)) -> f;
                end;

        begin { area }
                { NB.           Lazy evaluation of 'true' conditional branch is necessary for recursion,
                                also used in 'false' branch to eliminate unnecessary calls to 'f' }
                if (b-a) > dx
                then `area(a, (b + a) / 2.0, dx) + area((b + a) / 2.0, b, dx)
```

```
          else `f(a) * (b - a)
          -> area;
     end;

begin { main }
     area(a, b, dx) -> (1,-32,0);
     prime
     begin
          0.0 -> a;
          2.00 -> b;
          0.01 -> dx;
     end
end.
```

```
{ A singly (tail) recursive trapezoidal integration program }

[o2,2]        { set occurrences for double recursion }
[w+]          { extended code generation turned on }

program tr_single;

     shared subgraph area(x, b, dx, sum: real): real;

          subgraph f(x: real): real;
          constant
               pi = 3.141592654;
               mean = 0.0;
               sd = 1.0;
          begin
               1.0/(sd*(sqrt(2.0*pi)))*exp(-0.5*sqr(x-mean)/sqr(sd)) -> f;
          end;

     begin { area }
          { define the tail recursion }
          if x > b
          then sum
          else `area(x + dx, b, dx, sum + f(x) * dx)
          -> area;
     end;

begin { main }
     area(a, b, dx, sum) -> (1,-32,0);
     prime
     begin
          0.0 -> a, sum;
          2.00 -> b;
          0.01 -> dx;
     end
end.
```

```
{ An iterative trapezoidal integration program }

program itr;

     subgraph area(inita, initb, initdx: real): real;

          subgraph f(x: real) -> (y: real);
          begin
               sqr(x) -> y
          end;

     begin { int }
          { this loop is not (safely) reentrant (could be 'protected') }
          merge(newx, inita) -> x;
          merge(newsum, initsum) -> sum;
          merge(newb, initb) -> b;
          merge(newdx, initdx) -> dx;
          switch x < b
          then sum + f(x) * dx, x + dx, b, dx
          -> newsum, newx, newb, newdx
          else area, null, null, null;
          prime 0.0 -> initsum;
          end;

begin { main }
     write(output, int(a, b, dx)) -> null;
     prime
     begin
          0.0 -> a; 2.0 -> b; 0.01 -> dx;
     end
end.
```

```
{ Stream reversal program }

[a+,w+,o1,1]

program stream_reversal;

        shared subgraph _reverse(_in: any): any;

                shared subgraph _revl(_in1, _in2: any): any;
                begin
                        { moves the head of _in1 to _in2 }
                        get(_in1) -> h, _t;
                        cons(h, _in2) -> _x;
                        if empty(_t) then _x else `_revl(_t, _x) -> _revl;
                end;

        begin { _reverse }
                if not empty(_in) then `_revl(_in, ]) else `] -> _reverse;
        end;

begin
        _reverse(_a) -> (1,-32,0); { output result to the predefined console node }
        prime 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,] -> _a;
end.
```

```
{ A program that serially sums the elements of a stream }

[a+,w+,o1,1] { turn on advanced and extended code generation }

program serial_sum;

        shared subgraph serial(_input: integer): integer;
        begin
                if empty(_input) then 0 else `head(_input) + serial(tail(_input)) -> serial;
        end;

begin
        serial(_a) -> (1,-32,0);
        prime 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,] -> _a;
end.
```

```
{ A tail recursive equivalent of serial_sum (not optimised) }

[a+,w+,o1,1]

program serial_stream_sum;

        shared subgraph serial(partial_sum, _input: integer): integer;
        begin
                if empty(_input)
                then `partial_sum
                else `serial(partial_sum + head(_input), tail(_input))
                -> serial;
        end;

begin
        serial(a, _a) -> (1,-32,0);    { serial(0, _a) is slightly more expensive, since it requires a trigger }
        prime 0 -> a;
        prime 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,] -> _a;
end.
```

```
{
        Binary stream element addition using a recursive divide and conquer algorithm and
        lazy evaluation for protection against indefinite recursion and empty streams.
}

[w+,a+,o2,2]

program Binary_stream_sum;

        shared subgraph alternate(_x: integer) -> (_x1,_x2: integer);

                subgraph alt1(_t: integer) -> (_t1,_t2: integer);
                begin
                        get(_t) -> thead, _ttail;
                        alternate(_ttail) -> _t2,_t3;
                        cons(thead, _t3) -> _t1;
                end;

        begin { alternate }
                if empty(_x) then `], ] else `alt1(_x) -> _x1,_x2;
        end;
        shared subgraph binary_add(_input: integer) -> (sum: integer);

                subgraph bin1(_input: integer): integer;

                        subgraph bin2(_input: integer): integer;
                        begin
                                alternate(_input) -> _x,_y;
                                binary_add(_x) + binary_add(_y) -> bin2;
                        end;

                begin { bin1 }
                        get(_input) -> hhead,_tail;
                        if empty(_tail) then hhead else `bin2(_input) -> bin1;
                end;

        begin { binary_add }
                if empty(_input) then 0 else `bin1(_input) -> sum;
        end;

begin { Binary_stream_sum }
        prime
                1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,] -> _a;
        binary_add(_a) -> (1,-32,0);
end.
```

```
{
        Binary stream element addition using switch and join are used for conditional execution
without needing the auxiliary functions 'bin1' and 'bin2'.
}

[w+,a+,o2,2]

program Binary_stream_sum;

        shared subgraph alternate(_x: integer) -> (_x1,_x2: integer);

                subgraph alt1(_t: integer) -> (_t1,_t2: integer);
                begin
                        get(_t) -> hhead, _ttail;
                        alternate(_ttail) -> _t2,_t3;
                        cons(hhead, _t3) -> _t1;
                end;

        begin { alternate }
                if empty(_x) then ], ] else `alt1(_x) -> _x1,_x2;
        end;

        shared subgraph binary_add(_input: integer): integer;
        begin
                empty(_input) -> c1;
                switch c1 then _input -> else _input1;
                        get(_input1) -> h1,_t1;
                        empty(_t1) -> c2;
                        switch c2 then h1, _input1 -> h2,null else null,_input2;
                                alternate(_input2) -> _x,_y;
                        join c2 then h2 else binary_add(_x) + binary_add(_y) -> sum;
                join c1 then 0 else sum -> binary_add;
        end;

begin { Binary_stream_sum }
        prime 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,] -> _a;
        binary_add(_a) -> (1,-32,0);
end.
```

Appendix B
DL1 REFERENCE

B.1 Introduction

This appendix gives technical information on the DL1 dataflow programming language. Included are syntax diagrams, compiler options, reserved words and predefined functions. The information presented here supersedes all previous versions.

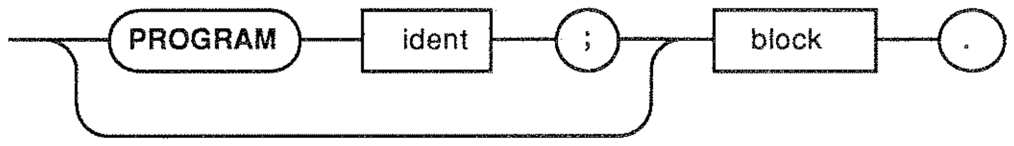
B.2 Using DL1

The compiler accepts source in a '.dl1' file (the extension is supplied by the compiler) and generates a '.lis' file with compile time statistics and an optional compiler generated listing. In the case of a successful compilation, a '.itl' file is produced which contains the intermediate target language (ITL), textual machine graph and priming tokens description. The '.itl' file is used as input to the simulator, DFSIM, or to the ITL to machine binary translator, ITL68K. Errors are reported to standard output as well as being logged in the list file.

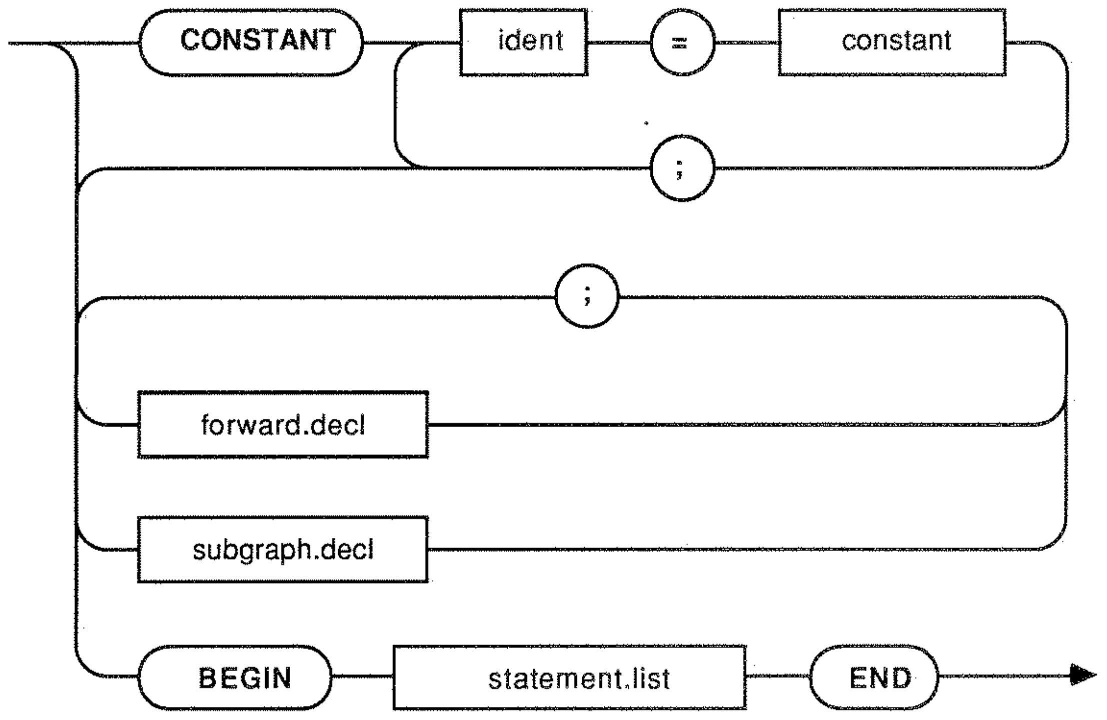
B.3 Syntax Diagrams

An extensive user's manual exists, that gives many examples and explanations of the features of the language [39]. The syntax of DL1 is simple 'LL', the compiler being recursive decent, one symbol/one character look ahead, with no backtracking. Block structure is fully supported where sensible, but reference can not be made to non local variables due to possible implications of non-functionality.

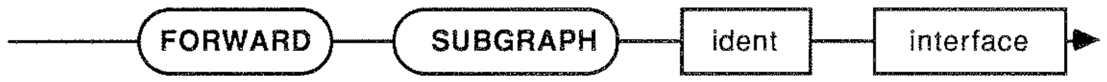
program



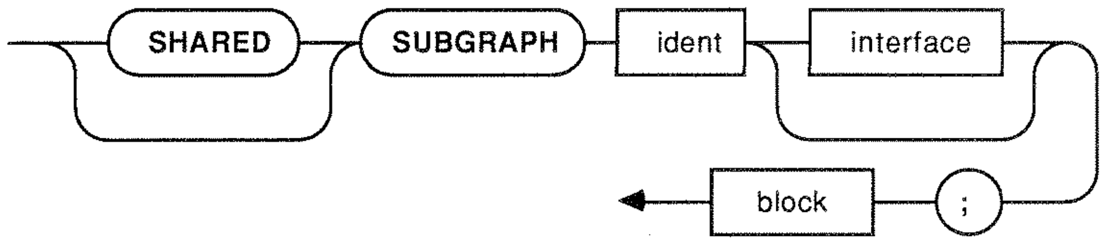
block



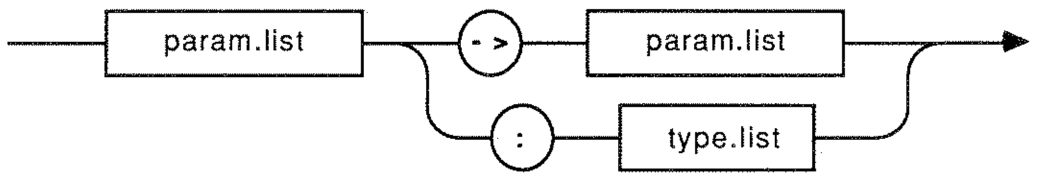
forward.decl



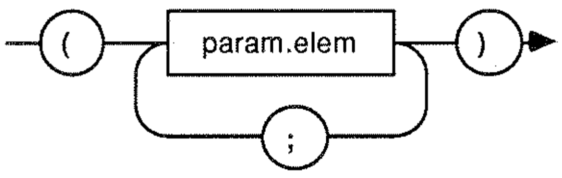
subgraph.decl



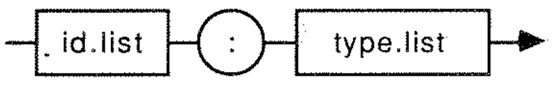
interface



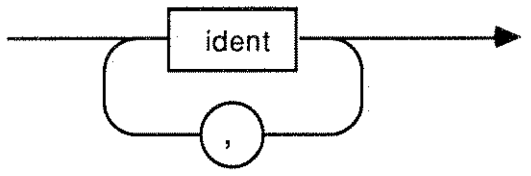
param.list



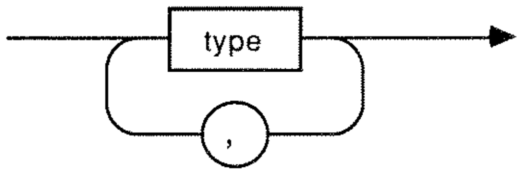
param.elem



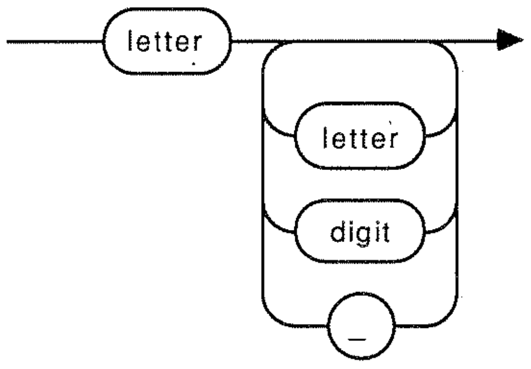
id.list



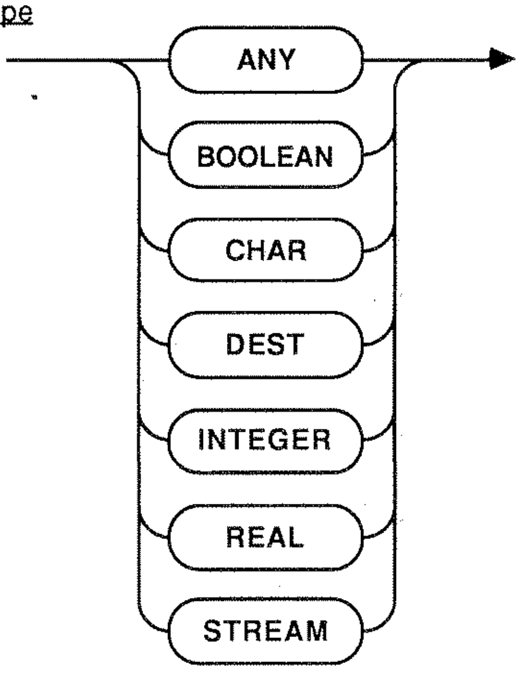
type.list



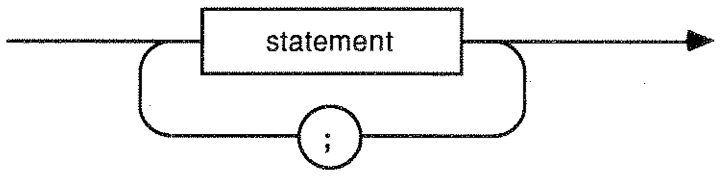
ident



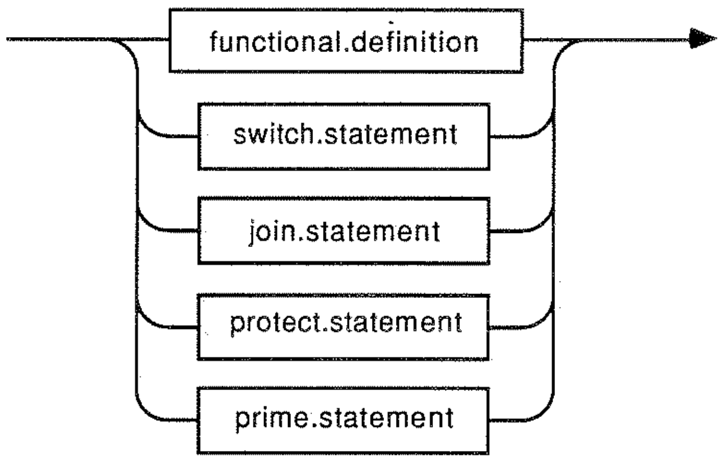
type



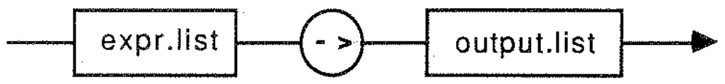
statement.list



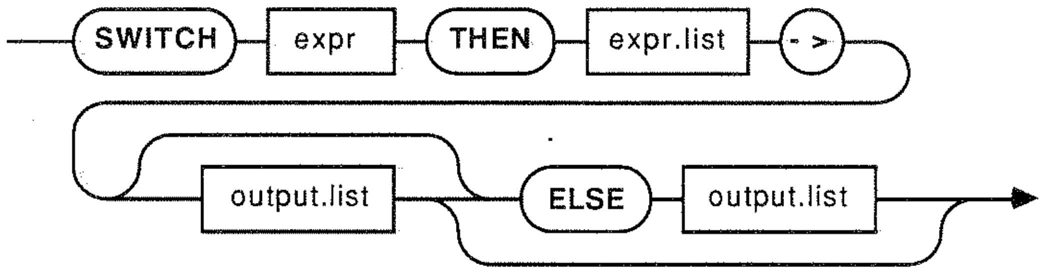
statement



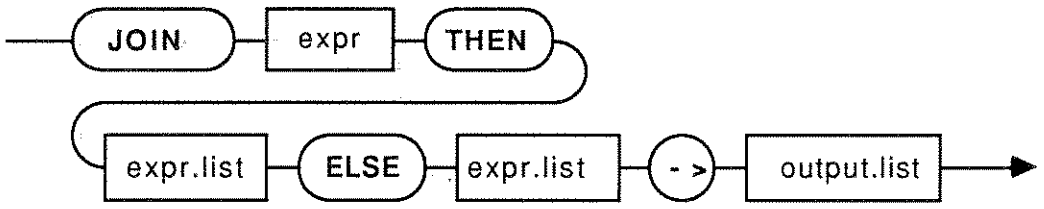
functional.definition



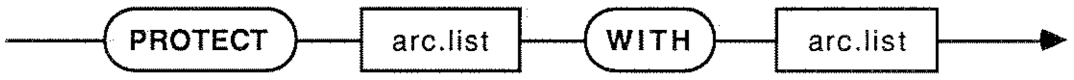
switch.statement



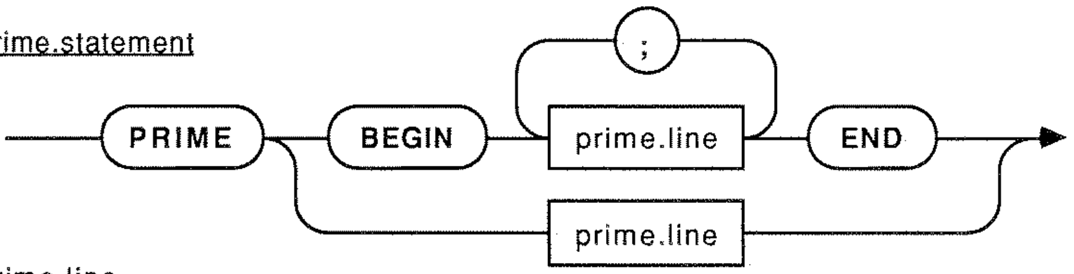
join.statement



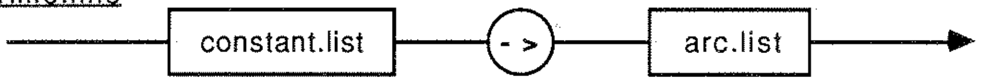
protect.statement



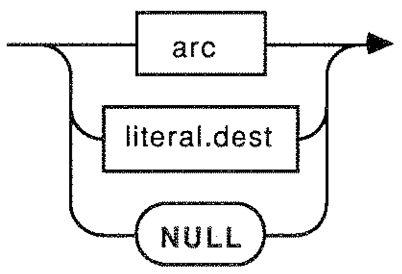
prime.statement



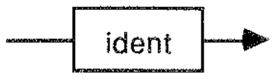
prime.line



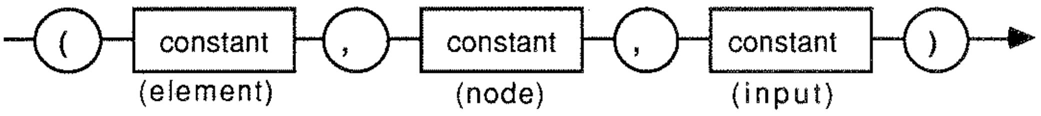
output



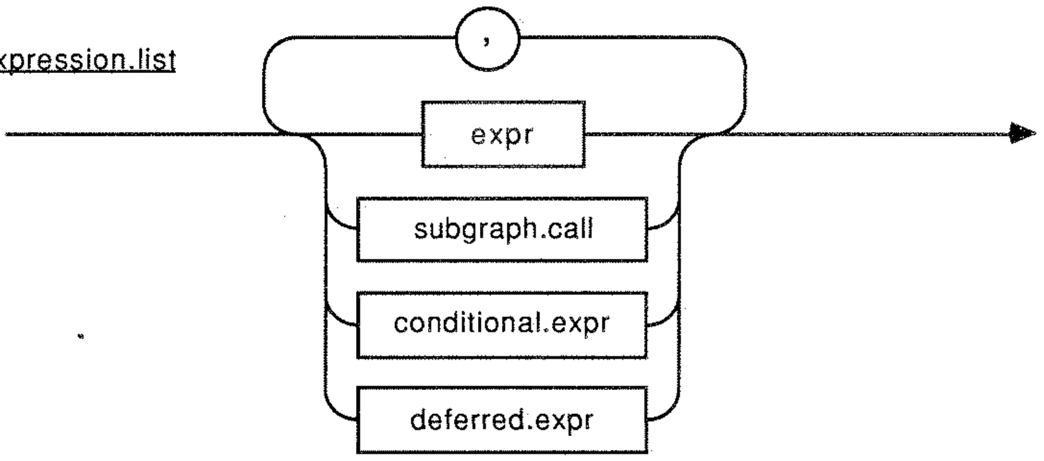
arc



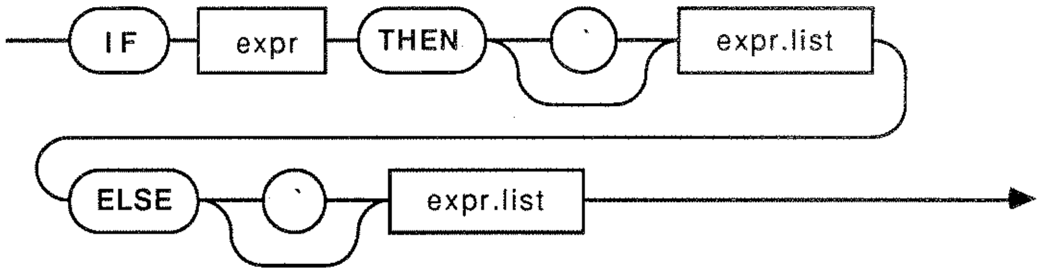
literal.dest



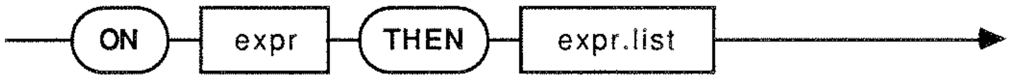
expression.list



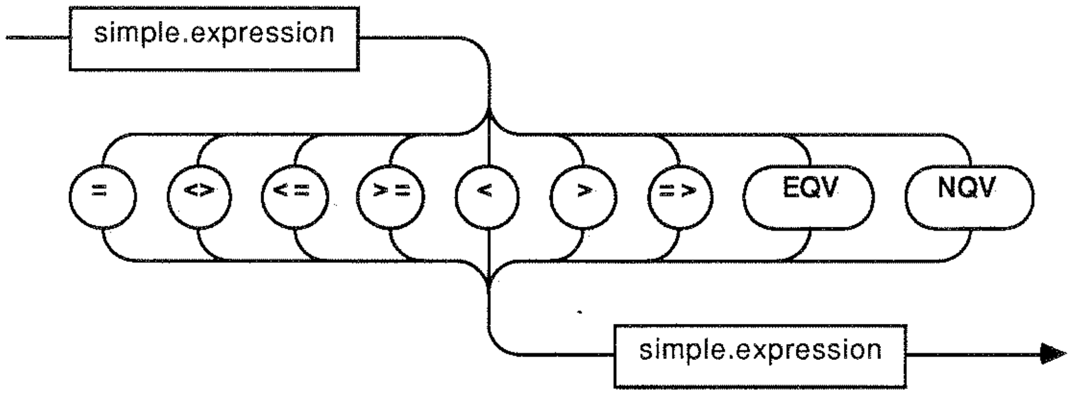
conditional.expr



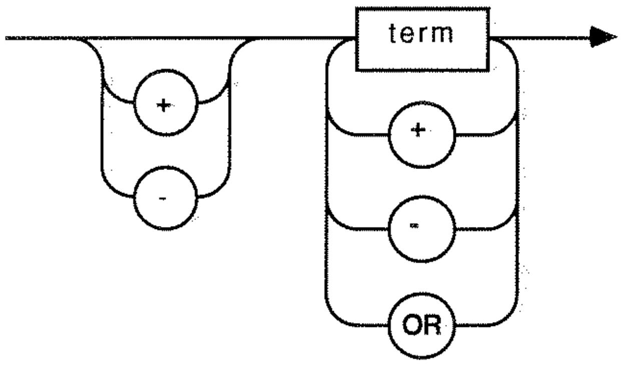
deferred.expr



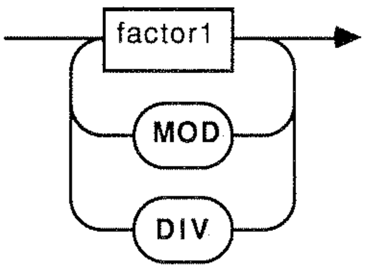
expression



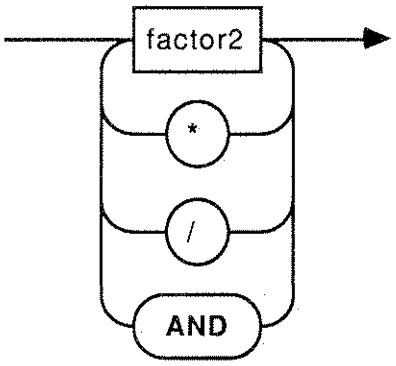
simple.expression



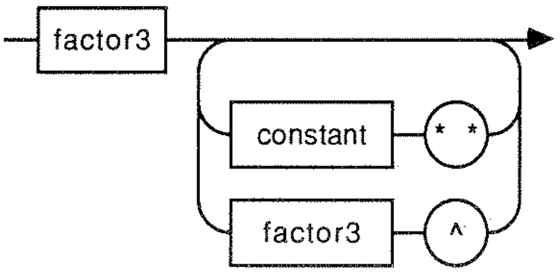
term



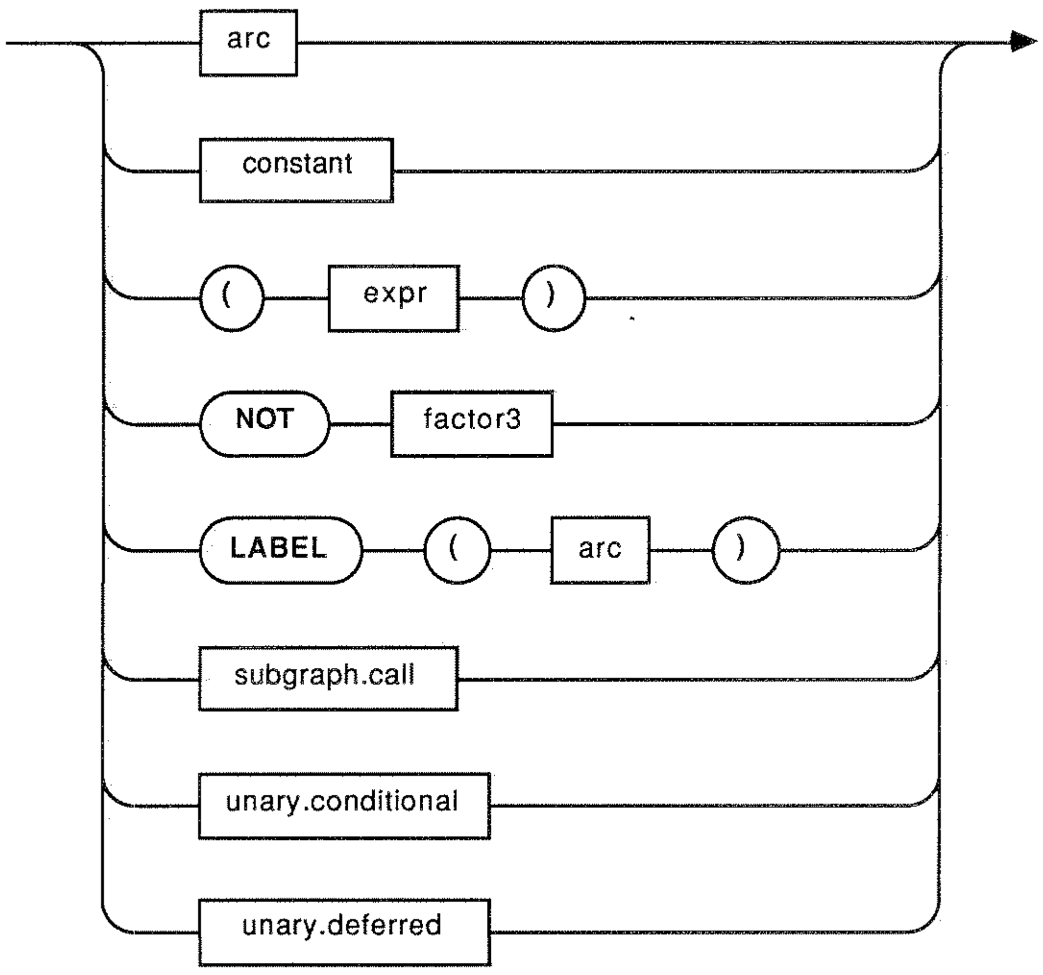
factor1



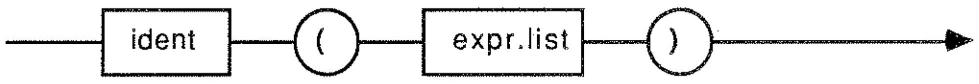
factor2



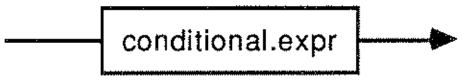
factor3



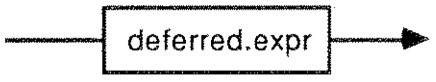
subgraph.call



unary.conditional



unary.deferred



B.4 Compiler Options

DL1 allows several options to be present in the source file, which provide control over code production, compile time statistics, listings, etc.. Options are imbedded between square brackets, e.g., to provide run time node tracing and an informative but brief compile time display, the option string [x+,k+,i+,l-] could be used. The options have been used extensively in gathering the results of chapter 5; they have proved very useful as a way to vary compile time parameters to generate comparative results.

Option/Default	**Effect**
a -	advanced node set for code production
b -	byte node addressing
c +	compressed ITL listing
d -	determinate/reentrant code production
e +	verbose error monitoring
f -	fine grain for exponentiation (LNE/MUL vs PWR)
h -	heap checking for debugging
i -	display of arc usage
k -	compile time information
l -	compiler generated listing
m +	use new *occurrence* data format (*<maxocc.occ>*)
o 8,8	top and max occurrence for shared subgraphs
r +	constant reduction for literals
s -	scanner output for debugging
t -	type checking
w -	extended node set for code production
x -	run time trace between [x+] and [x-]
[0 : 127]	element range for following node assignments

B.5 Reserved Words

Graph Delimiters
program, constant, forward, shared, subgraph, begin, end
Statement Delimiters
switch, join, oldif, protect, prime
Expression Delimiters
if, then , else, either, endif, on, endon, with
Predefined Constants
true, false, eos (or ']', end of stream), null (the token bucket)
Type Identifiers
any, boolean, char, copy, dest, integer, real, stream
Operators
and, or, eqv, xor, not, mod, div
I/O Streams
input, output, bend, elbow, twist, waist, grip, swivel, flooutl-, floinpl-

B.6 Predefined Functions

Arithmetic
abs, ln, exp, log, pwr, round, trunc, sqr, sqrt, sin, cos, tan, arccos, arcsin, arctan
Bit string
clearbit, setbit, testbit
Character
chr, ord
I/O
read, current, write, writeln, las
Stream
head, tail, get, empty, cons, bracket, unbracket
Misc
compare, first, issue, merge, pred, succ, store, when, setdest, setcopy, yield, label

Appendix C
ADVANCED NODE SET DEFINITION

C.1 Introduction

This node set definition describes all of the node functions currently defined in the RMIT dataflow variant and used in this thesis. This includes all original 'FLO' nodes still in use, as well as the extended and advanced nodes mentioned in this thesis. The format includes a description of the matching functions and their operation. Also given are timing figures for the emulator hardware. A new node-set is currently being defined which will incorporate the suggestions made in this thesis and other work at RMIT [25]. The node set definition given here should therefore be taken as a guide only.

The entry for each node function consists of the Intermediate Target Language (ITL) mnemonic followed by an informal description of the function. The matching function or firing rule of the node is indicated by entries of the form **diadic**, **protected**, etc.. Future node sets will have matching functions specified independently of the node function, as different matching functions may make sense on the one node.

C.2 Multiple Output Destinations

Most nodes allow their results to be distributed to a maximum of two destination nodes. In the node definition table, nodes with two outputs have their output arcs referred to explicitly, i.e. **out.0** and **out.1**. For one output nodes the output arc is simply referred to as **out**, but will occupy the position of **out.0** in the actual node description.

The simulator has a special mode to allow trees of `duplicate` nodes to be collapsed into a single `replicate`, with multiple outputs. In future node sets, all general purpose nodes will be able to send multiple copies of their output(s), specified by a destination list. This will handled by the Result Distribution Unit (figure 2.1).

C.3 Transparent Transmission of End of Stream Tokens

To reduce special case processing, end of stream tokens (**eos**, or]) are passed transparently for several functions. In the case of diadic functions, both argument tokens must be end of stream, unless one input is a literal. This allows efficient scalar operations on the elements of a stream.

C.4 Exceptions

Exceptions are generally handled by propagating the reason, operands and the name of the node at which the exception occurred as an 'exception' or **?** token to successor nodes which, in most cases, will propagate the ? token further. In cases where the successor node is not known or is ambiguous, e.g. an attempted use of a non-boolean (**bits**) token as the selector on a path control node, the resulting ? token is sent directly to the processing element's exception node (see system nodes).

C.5 Node Definitions

The following notes are referred to in the node definition table:

(1)	Applies for the **real int** case also
(2)	Always gives **real** result
(3)	Integer division
(4)	Exponentiation using log and multiply
(5)	Unary minus
(6)	Automatic type coercion where sensible
(7)	Exponentiation base **e**
(8)	Angular arguments and results are in radians
(9)	Same operation as XOR
(10)	Empty storage node fetch generates an exception

(11)	Special node with multiple (>2) outputs
(12)	Type should be **bits**
(13)	Exception if inp.0 is not protected, no queueing on inp.1 (yet)
(14)	Sends several results to same destination
(15)	Uses the literal as inp.1, else inp.1 will clear the node
(16)	Returns two outputs in this case
(17)	Destination not coded as part of node description
(18)	The clock generates boolean true tokens (i.e., **bits** $FFFF)
(19)	Δ = size difference, m = longest, n = shortest (all in words)
(20)	No evaluation is performed for this operation
(21)	Not yet implemented by the emulator
(22)	A second pair indicates timing with AMD9511 h/ware support
(23)	The number of cycles is given for an 8MHz clock (4MHz 9511)
(24)	The time for this operation is not included in the execution
(25)	Time depends upon length and type of resulting token

C.5.1 Arithmetic

Mnemonic	Op	Input Types		Output Types	Timing (µsec (#cycles))	
diadic						
ADD	+	**int**	**int**	**int**	22.75 (182) (22)	
		int	**real**	**real** (1)	248 (1984)	152 (1215)
		real	**real**	**real**	152 (1214)	129 (1035)
SUB	–	**int**	**int**	**int**	22.75 (182)	
		int	**real**	**real** (1)	246 (1965)	154 (1233)
		real	**real**	**real**	149 (1195)	132 (1053)
MUL	*	**int**	**int**	**int**	22.75 (182)	
		int	**real**	**real** (1)	274 (2189)	138 (1107)
		real	**real**	**real**	177 (1419)	116 (927)
DVD	÷	**int**	**int**	**real** (2)	423 (3387)	201 (1605)
		int	**real**	**real** (1)	290 (2321)	141 (1131)
		real	**real**	**real**	194 (1551)	119 (951)
DIV	div (3)	**int**	**int**	**int**	25 (197)	
MOD	mod	**int**	**int**	**int**	25 (201)	
PWR	^ (4)	**int**	**int**	**real**	5061 (40485)	2699 (21589)
		int	**real**	**real**	4964 (39715)	2639 (21115)
		real	**real**	**real**	4868 (38945)	2617 (20935)
monadic						
NEG	– (5)	**int**	-	**int**	6.5 (52)	
		real	-	**real**	10 (79)	11 (87)
ABS	abs	**int**	-	**int**	7.75 (62)	
		real	-	**real**	10 (79)	10 (79)
LNE	$\log_e$	**int**	-	**real** (6)	1969 (15754)	1497 (11972)
		real	-	**real**	1853 (14826)	1454 (11635)
LOG	$\log_{10}$	**int**	-	**real** (6)	2113 (16907)	1541 (12324)
		real	-	**real**	1997 (15979)	1498 (11987)
EXP	e^x (7)	**int**	-	**real** (6)	2978 (23824)	1174 (9390)
		real	-	**real**	2862 (22896)	1132 (9053)
SQT	√	**int**	-	**real** (6)	1612 (12894)	296 (2370)
		real	-	**real**	1496 (11966)	254 (2033)
SQR	square	**int**	-	**real** (6)	_ (21)	
		real	-	**real**	_ (21)	
SIN	sin (8)	**int**	-	**real** (6)	4112 (32894)	1165 (9322)
		real	-	**real**	3996 (31966)	1123 (8985)
COS	cos	**int**	-	**real** (6)	4049 (32394)	1180 (9436)
		real	-	**real**	3933 (31466)	1137 (9099)
TAN	tan	**int**	-	**real** (6)	8163 (65307)	1437 (11498)
		real	-	**real**	8047 (64379)	1395 (11161)
ASN	arsin	**int**	-	**real** (6)	3725 (29798)	1861 (14886)
		real	-	**real**	3609 (28870)	1819 (14549)
ACS	arcos	**int**	-	**real** (6)	3745 (29960)	1913 (15306)
		real	-	**real**	3629 (29032)	1871 (14969)
ATN	artan	**int**	-	**real** (6)	1836 (14684)	1531 (12246)
		real	-	**real**	1720 (13756)	1489 (11909)

C.5.2 Logical and Set

Mnemonic	Op	Input Types		Output Types	Timing (μsec (#cycles))
diadic					
AND	∧	**bits**	**bits**	**bits**	27+3Δ+4m (216+22Δ+34m) (19)
IOR	∨	**bits**	**bits**	**bits**	25+4n (206+34n)
XOR	⊕	**bits**	**bits**	**bits**	25+4n (206+34n)
IMP	⇒	**bits**	**bits**	**bits**	28+3Δ+4m (224+22Δ+34m)
EQV	⇔	**bits**	**bits**	**bits**	27+3Δ+4m (216+22Δ+34m)
NQV	≠ (9)	**bits**	**bits**	**bits**	25+4n (206+34n)
TSB	test bit	**int**	**bits**	**bits**	26 (205)
STB	set bit	**int**	**bits**	**bits**	26 (205)
CLB	clear bit	**int**	**bits**	**bits**	26 (205)
monadic					
NOT	¬	**bits**	-	**bits**	9+4.5n (72+36n)

C.5.3 Relational

diadic						
EQ	=					
NE	≠					
GE	≥					
GT	>					
LE	≤					
LT	<	**int**	**int**	**bits**	16.9 (135)	
		int	**real**	**bits** (1)	212 (1699)	64.9 (519)
		real	**real**	**bits**	86.7 (694)	26.1 (209)
		char	**char**	**bits**	27.4 (219)	
CPT	compare types					
		any	**any**	**bits**	5.6 (45)	

C.5.4 Sequence

monadic					
SUC	succ				
PRE	pred	**char**	-	**char**	6.6 (53)
		int	-	**int**	10.4 (83)

C.5.5 Stream

monadic					
BRA	bracket	<inp.0 -> out; **eos** -> out>			
		any	-	**any, eos**	5.8 (46)
UNB	unbracket	<if type_of (inp.0) ≠ eos then inp.0 -> out>			
		any	-	**any** (¬ **eos**)	5.5 (44)
head					
HD	head	<head of stream(inp.0) -> out>			
		any	-	**any** (¬ **eos**)	_ (21)
tail					
TL	tail	<tail of stream(inp.0) -> out>			
		any	-	**any** (¬ **eos**)	_ (21)
stream					
STS	stream store	<for each element of stream(inp.0) copy of inp.1 -> out>			
		any	-	**any** (¬ **eos**)	_ (21)
cons					
CON	cons	stream cons(inp.0, inp.1) -> out			
		any	**any**	**any**	_ (21)

C.5.6 Type Coercion

Mnemonic	**Op**	**Input Types**		**Output Types**		**Timing** (µsec (#cycles))	
monadic							
ORD	ord	**int**	-	**int**		3.3 (26)	
		char	-	**int**		12.5 (100)	
CHR	chr	**int**	-	**char**		13.1 (105)	
		char	-	**char**		3.3 (26)	
RND	round	**int**	-	**int**		3.3 (26)	
		real	-	**int**		110 (880)	155 (1243)
TRN	trunc	**int**	-	**int**		3.3 (26)	
		real	-	**int**		117 (939)	78.5 (628)
FLT	float	**int**	-	**real**		121 (966)	61.9 (495)
		real	-	**real**		7.8 (62)	

C.5.7 Storage

storage							
S	storage						
	store	<inp.0 is stored>					
		any	-	no output		* (20)	
	retrieve	<copy_of_last inp.0 -> out (10)>					
		-	**any**	**any**		1.3 (10)	

C.5.8 Replicate and Identity

monadic							
DUP	dup	**any**	-	**any**	**any**	3.8 (30)	
REP	rep	**any**	-	**any...** (11)		_ (21)	
merge							
ID	identity	<merge(inp.0, inp.1) -> out>					
		any	**any**	**any**		_ (21)	

The identity function may be used to enforce explicit merging of two arcs onto one arc without using the communications network to do so. This merge is non-strict and non-reentrant.

C.5.9 Path Control

diadic							
PRS	presence	<on inp.0, inp.1 then true -> out>					
		any	**any**	**bits**			2.3 (18)
PIT	pass if true	<if inp.1 then inp.0 -> out>					
		any	**bits**	**any**			17 (136)
PIF	pass if false	<if ¬inp.1 then inp.0 -> out>					
		any	**bits**	**any**			17 (136)
PIP	pass if present	<on inp.1 then inp.0 -> out>					
		any	**any**	**any**			1.3 (10)
SWI	switch	<if inp.1 then inp.0 -> out.1 else inp.0 -> out.0>					
		any	**bits**	**any**	**any**		16 (128)
SYN	synchronise	<on inp.0, inp.1 then inp.0 -> out.0; inp.1 -> out.1>					
		any	**any**	**any**	**any**		_ (21)
non-strict diadic							
ARB	arbitrate	<earlier of inp.0, inp.1 -> out.0, latter -> out.1>					
		any	**any**	**any**	**any**		_ (21)

Mnemonic	**Op**	**Input Types**		**Output Types**		**Timing** (µsec (#cycles))
protected	(initially unprotected)					
PRT	protection					
	passing	<if unprotected(inp.0) then inp.0 -> out; protect(inp.0)>				
		any	-	**any**		_ (21)
	blocking	<if protected(inp.0) then queue(inp.0)>				
		any	-	no output		_ (21)
	clearing	<on inp.1 then unprotect(inp.0) (13)>				
		-	**any**	no output		_ (21)
	retriggering	<on inp.1, protected(inp.0) then inp.0 -> out (13)>				
		-	**any**	**any**		_ (21)
EMC	eager merge control					
	passing	<if unprotected(inp.0) then inp.0 -> out.0, out.1; protect(inp.0)>				
		any (12)	-	**any** (12)	**any** (12)	_ (21)
	blocking	<if protected(inp.0) then queue(inp.0)>				
		any (12)	-	no output		_ (21)
	clearing	<on inp.1 then unprotect (inp.0); (13)>				
		-	**any**	no output		_ (21)
	retriggering	<on inp.1, protected(inp.0) then inp.0 -> out.0, out.1 (13)>				
		-	**any**	**any** (12)	**any** (12)	_ (21)
LMC	lazy merge control					
	passing	<if unprotected(inp.0) then if inp.0 then inp.0 -> out.0 else out.1; protect(inp.0)>				
		any (12)	-	**any** or	**any** (12)	_ (21)
	blocking	<if protected(inp.0) then queue(inp.0)>				
		any (12)	-	no output		_ (21)
	clearing	<on inp.1 then unprotect (inp.0); (13)>				
		-	**any**	no output		_ (21)
	retriggering	<on inp.1, protected(inp.0) then if inp.0 then inp.0 -> out.0 else out.1; (13)>				
		-	**any**	**any** or	**any** (12)	_ (21)

Switching and passing of tokens is conditional on the least significant bit of the bit-string (**bits**) token on inp.1. The internal convention adopted for generated true and false values is all bits set for **true** results and all bits cleared for **false** results.

C.5.10 Token Structure

monadic					
D	decode	<decode(inp.0) -> out>			
		any	-	**any...** (14)	_ (21)

The fields of the input token are returned as a stream of tokens. A specific use of this node is to decode ? tokens. The inverse function, which may be used when forming graphs, has not yet been implemented.

C.5.11 Priming

first monadic					
FIR	first				
	passing	<if not seen(inp.0) then inp.0 -> out; seen(inp.0)>			
		any	-	**any**	1.3 (10)
	blocking	<if seen(inp.0) then discard(inp.0)>			
		any	-	no output	* (20)
	clearing	<on inp.1 then not seen(inp.0)>			
		-	**any**	no output	* (20)

Mnemonic	**Op**	**Input Types**		**Output Types**		**Timing** (µsec (#cycles))
PRM	prime					
	priming	<if not seen(inp.0)				
			then literal, inp.0 -> out; seen(inp.0)>			
		any	**any** (15)	**any...** (16)		7.3 (58)
	passing		else inp.0 -> out>			
		any	-	**any**		4.3 (30)
	clearing	<on inp.1 then not seen(inp.0)>				
		-	**any**	no output		* (20)

The first and prime functions may be used for priming shared subgraphs. Both of these functions may be reset by a token of any type on inp.1. If not reset, state information (indicating seen(inp.0)) will be retained in the matching store, i.e., an unclean graph will result.

C.5.12 Destination

diadic						
STD	set destination <inp.0 -> [inp.1]>					
		any	**dest**	**any** (17)		6 (48)

As this node and the nodes of C.3.3 change the connectivity of the graph dynamically and thus introduce nondeterminism, they should be used with some care.

C.5.13 Colour

monadic						
YLC	yield colour	<inp.0 with colour 0 -> out.0; colour of inp.0 -> out.1>				
		any	-	**any**	**colour**	4.8 (38)
diadic						
STC	set colour	<inp.0 with colour inp.1 -> out>				
		any	**colour**	**any**		6 (48)

C.5.14 Shared Subgraphs (1)

monadic	(always with literal occurrence/link)					
A	arg entry	<inp.0 with copy newcopy -> out>				
		any	**occur**	**any**		10.8 (86)
R	return entry	<on inp.1 then link <occurrence, dest> -> out>				
		link	**any**	**link**		12 (96)
diadic						
E	exit	<inp.0 with copy newcopy -> [inp.1.dest]>				
		any	**link**	**any**		10.5 (84)

Argument tokens for the subgraph are provided by arg-entry (A) nodes. Appropriately triggered (by a subgraph argument for example) return-entry (R) nodes provide the destinations to which exit (E) nodes send subgraph result tokens. Arg-entry and return-entry nodes are associated with the invoking context. Exit nodes are associated with the body of the shared subgraph itself.

On entering a shared subgraph the token's copy number is computed as:
newcopy = (*oldcopy* * maxoccurrence) + occurrence

On exiting, the copy is computed as:
newcopy = (*oldcopy* – link.occurrence) div maxoccurrence
(if E receives a **link**)

If the new copy (or colour) is not zero, then the copy presentt/coloured bit is set in the result token's destination fields and *newcopy*/*newcolour* is appended. Occurrence is the number of the static graph instance at the calling graph level.

C.5.15 Shared Subgraphs (2)

Mnemonic	**Op**	**Input Types**		**Output Types**	**Timing** (µsec (#cycles))
monadic					
CRC	create colour		<<u>on</u> inp.1 <u>then</u> unique colour qualified by optional inp.0 -> out>		
		occur	**any**	**colour**	_ (21)
SRL	set return link		<<u>on</u> inp.1 <u>then</u> environment <colour inp.1, literal dest> -> out>		
		link	**colour**	**env**	_ (21)
diadic					
STC	set colour		<inp.0 with colour inp.1 -> out>		
		any	**colour**	**any**	_ (21)
E	exit		<inp.0 with colour env.colour -> [inp.1.dest]>		
		any	**env**	**any**	10.5 (84)

These functions form an alternate context mechanism that relies upon unique (nonrecurring) colours as opposed to computed (recurring) copy numbers. CRC generates a unique colour which is appended to arguments by STC nodes and used by SRL nodes to generate environment (return address) tokens. The advantage of using nonrecurring tags is that some code templates may be greatly simplified and run much more efficiently, e.g., lazy merge macros can be eliminated altogether, to be replaced by a simple merge operation. In addition, graph unravelling is more effective with nonrecurring tags.

C.5.16 Predefined Nodes

A number of system nodes are predefined in every processing element. In addition, input and output nodes are also predefined, but are associated with specific devices which are in turn associated with specific processing elements; this association varies between installations and ensures that resource managers can handle i/o streams without fear of garbling them through multiple accessing.

C.5.17 System Nodes

Non i/o system node names are reserved, with their node descriptions existing in all processing elements; e is the processing element number.

Although a particular system node may be referred to at a number of places in the graph, it represents a single resource. Multiple referencing therefore implies indeterminate merging on the node's input points. Unless this is intentional, the node should be referenced once only within an encapsulating resource manager.

monadic					
e.-1	Node Store	<inp.0 -> node store>			
		node	-	no output	_ (24)
e.-2	Set max occur		<inp.0 -> max occurrence>		
		occur	-	no output	_ (24)
e.-3	Exception Node				
	setting	<set destination for exception tokens to inp.1>			
		-	**dest**	no output	_ (24)
	firing	<exception tokens -> [last inp.1]>			
		-	-	?	_ (25)
e.-4	Trace Node				
	setting	<set destination for trace tokens to inp.1>			
		-	**dest**	no output	_ (24)
	firing	<trace tokens -> [last inp.1]>			
		-	-	?	_ (25)
e.-5 Clock Node					
	setting	<set destination for timing tokens to inp.1>			
		-	**dest**	no output	_ (24)
	firing	<timing tokens -> [last inp.1]>			
		-	-	**bits** (18)	_ (21)

C.5.18 Input and Output

As input and output nodes have physical devices associated with their node names, there will be restrictions on the type of tokens produced or accepted by these nodes. Type and length information is preserved in all input and output operations. Node n is the i/o stream identifier while element e has the appropriate device driver.

Name	**Op**	**Input Types**		**Output Types**	**Timing**
monadic					
e.n	INPUT				
	setting	<set destination for device tokens to inp.1>			
		-	**dest**	no output	_ (21)
	reading	<on inp.0 then device.token -> [last inp.1]>			
		any	-	**any** (17)	_ (21)
e.n	OUTPUT				
	setting	<set destination for acknowledge to inp.1>			
		-	**dest**	no output	_ (21)
writing	<inp.0 -> device (data); inp.0 -> [last inp.1] (ack)>				
		any	-	**any** (17)	_ (21)

In the case of output nodes, provision of an acknowledgement destination on inp.1 is optional.

Appendix D
EMULATION DETAILS

D.1 Emulator Hardware

This appendix includes descriptions of the prototype emulator hardware and the binary token and node formats that it uses [40, 26]. Also included is a description of the textual intermediate target language (ITL) format which also reflects a format designed around the 16 bit emulator hardware.

The prototype emulator board includes two Motorola M68000 CPUs (referred to as CPU1 and CPU2) with 128k RAM and 4k ROM each. The RAM is directly expandable to 512k for each CPU simply by replacing the 64k DRAM chips with 256k chips. CPU2 has access to an AMD9511 floating point arithmetic chip which may be used to evaluate many of the primitive node functions. The four, 2k x 16 bit FIFOs are implemented using Signetics 8X60N Fifo Ram Controller (FRC) chips with two 2k x 8 static RAM chips each. CPU1 has control over the input FIFO (for resetting, interrupt handling, etc.) and CPU2 has control over the output FIFO. Both CPUs have access to the pipe and local FIFOs. In addition to the input and output fifos, both CPUs have extra external interfaces consisting of one 8 bit input port and one 8 bit output port each.

As part of an exercise to evaluate the potential of a stand alone CPU module, CPU2 was programmed to handle all of the defined nodes by itself, including floating point functions, so that the 9511 can be done without if so desired. A timing analysis of the nodes implemented to date on the emulator is presented in appendix C.

The hardware and software of the emulator have been designed to allow either polled or interrupt driven operation but it was quickly discovered in programming the emulator that polled operation of the CPU/FIFO interface would out-perform interrupt driven operation quite easily. It was found that the FIFOs were running near empty (between tokens) a lot of the time and the cost of servicing FIFO empty interrupts was very high. In the time taken to service one interrupt, the CPU could poll the FIFO several times over. Another, more subtle problem was found with interrupt driven operation in that when the last word of a token was read out of a FIFO, so that the FIFO went empty, then the CPU was stalled from processing that token while it waited for the FIFO to go non-empty (an event which would never occur at all if there were no more tokens). Again, this situation arose because the FIFOs ran much closer to empty than was anticipated.

D.2 Token Format

Tokens carry all intermediate results around the dataflow machine, as well as the initial graph descriptions in the form of *node* type tokens. The currently defined token formats are:-

<token>	::=	<destination> [<tag>] <data>
<destination>	::=	<element#> <tag present> <input point> <node#>
<tag>	::=	<copy> | <colour> (see D.3)
<data>	::=	<data header> <data value>
<element#>	::=	**16 bit number**, (can include an 8 bit *Context* field)
<tag present>	::=	**1 bit boolean: (false |true)**
<input point>	::=	**1 bit scalar: (0 |1)**
<node#>	::=	**14 bit number**
<data>	::=	see below

D.3 Data Types

Tokens carry a <data header> field which describes the type and length of the data value which follows. The <data type> field allows for dynamic type checking and coercion, in addition it greatly simplifies debugging and tracing of graph execution. The <data length> field gives the number of 16 bit words taken up by the data value itself. The currently defined data types are:-

<data>	::=	<data header> <data value>
<data header>	::=	<data type> <data length>
<data type>	::=	**8 bit scalar** (see <data value>)
<data length>	::=	**8 bit number** (see <data value>)
<data value>	::=	<real> | <integer> | <bits> | <chars> | <eos> | <dest> | <colour> | <occur> | <link> | <env> | <?> | <node>
<real>	::=	**32 bit real**
<integer>	::=	**16 bit integer**
<bits>	::=	**variable length bit vector** (for sets and boolean arrays)
<chars>	::=	**variable length char vector** (for chars and strings)
<eos>	::=	**zero length end-of-stream**
<dest>	::=	**32 bit untagged token destination fields**
<colour>	::=	**32 bit tag** (copy or colour)
<occur>	::=	**8 bit occurrence, 8 bit maxoccurrence**
<link>	::=	<occur> <dest> (for old tagging scheme)
<env>	::=	<colour> <dest> (for new tagging scheme)
<?>	::=	**variable length exception** (see D.4)
<node>	::=	**variable length node description** (see D.5)

Character strings (**chars**) with an odd number of characters are terminated by a null character whereas even sized strings have no terminator, their finish being implied by the data length itself. A single ascii character is represented as a null filled string of length one (word).

Bit strings (**bits**) are used for sets, packed boolean arrays and simple booleans. The least significant bit is used as the value of a boolean. When two bit strings of unequal length arrive at a diadic node, the shorter bit string is padded to the length of the longer string with leading zeros. The output of such a node will have a length equal to that of the longest input.

D.4 Exceptions

Run time exceptions that occur as a result of a nodes execution or some other condition (errors, tracing, etc.) are handled by generating a token of type **exception** or **?**. The variable length data field allows information as to the source of the exception, the location of the error, the node function, and the data involved to all be carried on the one token. The format for exception tokens is:-

<?>	::=	<reason> <location> [<node header>] [<arguments>]
<reason>	::=	**16 bit scalar** (notdest, notcolour, notenv, notlink, notbits, argtype, arglength, overflow, underflow, emptystore, nonode, msover, nsover) (see reference [25] for further details)
<location>	::=	<dest> [<tag>] (the original location of the exception)
<node header>	::=	(the header of the exception generating node, if any, see D.5)
<arguments>	::=	<data value> [<data value>] (the original input(s), if any)

D.5 Nodes

The loading of a dataflow graph into the multiprocessor is achieved by directing binary tokens of type *node* to the predefined `node-store` node in each processing element which uses that node. The binary node format is:-

<node>	::=	<node#> <node header> [<literal>] [<dest*>] [<dest*>] [<link>]
<node#>	::=	**14 bit number** (bits 14 and 15 not used)
<node header>	::=	<one input> <data present> <trace> <function>
<one input>	::=	**1 bit flag**
<data present>	::=	**1 bit flag**
<trace>	::=	**1 bit flag**
<extended>	::=	**1 bit flag** (indicates an extension, see <link>)
<function>	::=	**11 bit scalar** (see appendix C for defined nodes)
<literal>	::=	<data> (optional literal data saved in node store)
<dest*>	::=	<dest> (with <tag present> = **0**)
<link>	::=	**16 bit pointer** (for nodes longer than 5 words)

D.6 ITL Format

The intermediate target language, or ITL, is an ascii representation of dataflow machine graphs and priming tokens. Each line in an ITL file describes either a node or a priming token, special entries to set simulation parameters will not be described here but can be found in [23].

<ITLNODE>	::=	**N** <element#> <node#> <TR> <IN> <DP> <MNEM> [<ITLDATA>] [<ITLDEST>] [<ITLDEST>]
<TR>	::=	**: | *** (trace this node)
<IN>	::=	**F | T** (this node has one input, (matching function))
<DP>	::=	**F | T** (this node has literal data present)
<MNEM>	::=	**1-3 char string** (see appendix C for defied mnemonics)
<ITLDATA>	::=	**ascii type and value** (of the literal data)
<ITLDEST>	::=	<element#> <node#> <input point>
<ITLPRIME>	::=	**T** <ITLDEST> <ITLDATA>

REFERENCES

[1] D. Abramson and G.K. Egan, "The RMIT Data Flow Computer: A Hybrid Architecture", Dep. Comm. and Elecn. Eng., Royal Melbourne Institute of Technology, Tech. Rep. TR-112-057R, 1987.

[2] D. Abramson, G.K. Egan, M.W. Rawling, and C. Baharis, "The RMIT Data Flow Computer: Benchmarks", Dep. Comm. and Elecn. Eng., Royal Melbourne Institute of Technology, Tech. Report TR-112-058R, 1987.

[3] D. Abramson, G.K. Egan, M.W. Rawling, and A. Young, "The RMIT Data Flow Computer: The Architecture", Dep. Comm. and Elecn. Eng., Royal Melbourne Institute of Technology, Tech. Report TR-112-061R, 1987.

[4] W.B. Ackerman, "Dataflow Languages", *IEEE Computer*, vol. 15, no. 2, Feb. 1982, pp. 50-69.

[5] M. Amamiya and R. Hasegawa, "Dataflow Computing and Eager and Lazy Evaluations", *New Generation Computing*, 2, 1984, pp. 105-129.

[6] Arvind and D.E. Culler, "Data-flow Architectures", Laboratory for Computer Science, Massachusetts Institute of Technology, Tech. Memo TM-294, 1986.

[7] Arvind and K.P. Gostelow, "The U-Interpreter", *IEEE Computer*, vol. 15, no. 2, Feb. 1982, pp. 42-50.

[8] Arvind, K.P. Gostelow, and W. Plouffe, "An Asynchronous Programming Language and Computing Machine", Dep. of Information and Computer Science, University of California, Irvine, Dec. 1978.

[9] Arvind and R.A. Iannucci, "Instruction Set Definition for a Tagged-Token Data Flow Machine", Computation Structures Group Memo 212-3, Laboratory for Computer Science, Massachusetts Institute of Technology, Sep. 1982.

[10] Arvind and R.A. Iannucci, "A Critique of Multiprocessing von Neumann Style", in *Proc. 10th Ann. Symp. Computer Architecture*, Stockholm, June 1983, pp. 426-436.

[11] Arvind and R.E. Thomas, "I-structures: An Efficient Data Structure for Functional Languages", Laboratory for Computer Science, Massachusetts Institute of Technology, Tech. Memo TM-178, 1981.

[12] P. Barahona and J.R. Gurd, "Simulated Performance of the Manchester Multi-Ring Dataflow Machine", in *Parallel Computing '85*, M. Feilmeier et al (Eds.), pp. 419-424.

[13] A.J. Catto and J.R. Gurd, "Nondeterministic Dataflow Graphs", Information Processing 80, S.H. Lavington (Ed.), North Holland Publishing Co., 1980, pp. 251-6.

[14] M. Chase, "Data Flow Processor Speeds Imaging Tasks", *Electronic Imaging*, Dec. 1984, pp. 42-46.

[15] M. Cornish, D.W. Hogan, and J.C. Jensen, "The Texas Instruments Distributed Data Processor", in *Proc. Louisianna Computer Exposition*, Lafayette, La., March 1979, pp. 189-193.

[16] J.G.D da Silva and I. Watson, "Pseudo-associative Store with Hardware Hahing", *IEE Proc.*, Vol. 130, Pt E, no. 1, Jan. 1983.

[17] A.L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", in *Proc. 5th Ann. Symp. Computer Architecture*, New York, 1978, pp. 210-215.

[18] A.L. Davis, "A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems", Tech. Report UUCS-78-108, Dep. Comp. Science, University of Utah, July 1978.

[19] J.B. Dennis, G.A. Broughton, and C.K.C. Leung, "Building Blocks for Data Flow Prototypes", in *Proc. 7th Ann. Symp. Computer Architecture*, LaBoule, France, May 1980.

[20] J.B. Dennis, G.R. Gao, and K.W. Todd, "Modeling the Weather with a Data Flow Supercomputer", *IEEE Trans. Computers*, vol. C-33, no. 7, July 1984, pp. 592-603.

[21] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", in *Proc. 2nd Ann. Symp. Computer Architecture*, New York, May 1975.

[22] D.M. Dias and J.R. Jump, "Packet Switching Interconnection Networks", *IEEE Computer*, Dec. 1986, pp. 43-53.

[23] G.K. Egan, "FLO: A Decentralised Data-flow System, Parts 1 & 2", Internal Document, Dep. Computer Science, University of Manchester, 1980.

[24] G.K. Egan, "Data-flow: Its Application to Decentralised Control", Ph.D. dissertation, Dep. Computer Science, University of Manchester, 1979.

[25] G.K. Egan, "The RMIT Data Flow Computer: Token and Node Set Definition", Dep. Comm. and Elecn. Eng., Royal Melbourne Institute of Technology, 1987.

[26] G.K. Egan, C. Baharis, M.W. Rawling, and A. Young, "Royal Melbourne Institute of Technology Data-flow Project", in *IREE Proc. 2nd Australian Computer Engineering Conference*, Sydney, May 1986.

[27] D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn, "A Second Opinion on Dataflow Machines and Languages", *IEEE Computer*, Feb. 1982, pp. 58-69.

[28] J.R.W. Glauert, "A Single-Assignment Language for Data Flow Computing", M.Sc. dissertation, Dep. Computer Science, University of Manchester, Jan. 1978.

[29] J.R. Gurd, C.C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer", *CACM*, 1985, vol. 28, no. 1, Jan. 1985, p. 34.

[30] J.R. Gurd, I. Watson, and J.R.W. Glauert, "A Multi-Layered Dataflow Computer Architecture", Manchester Dataflow Group Report, July 1978.

[31] J.R. Gurd, P.C. Treleaven, and I. Watson, "A Data Flow Computer Architecture", Manchester Dataflow Group Draft Report, 1977.

[32] R.P. Hopkins, P.W. Rautenbach, and P.C. Trelevean, "A Computer Supporting Data Flow, Control Flow and Updateable Memory", University of Newcastle upon Tyne, Tech. Report No 144, Sep. 1979.

[33] P.E.C. Jones, B.P. Kidman, and R. Morello, "Code Generation from Expressions in a Dataflow Language", In *Proc. ACSC8, Aust. Comp. Science Conf.*, 1985.

[34] P.E.C. Jones and B.P. Kidman, "Common Subexpression Detection in Conditional Expressions in a Dataflow Language", In *Proc. ACSC9, Aust. Comp. Science Conf.*, 1986.

[35] J. McGraw, S. Skedzielewski, et al, "SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual", Lawrence Livermore National Laboratory, Livermore, California.

[36] D.P. Misunas, "Deadlock Avoidance in a Data-Flow Architecture", Computation Structures Group Memo 116, Massachusetts Institute of Technology, Feb. 1975.

[37] R.S. Nikhil, K. Pingali, and Arvind, "Id Nouveau", Computation Structures Group Memo 265, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1986.

[38] A. Plas et al, "LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment", in *Proc. 1976 Int. Conf. on Parallel Processing*, pp. 293-302.

[39] M.W. Rawling and C.P. Richardson, "The RMIT Data Flow Computer: DL1 User's Manual", Dep. Comm. and Elecn. Eng., Royal Melbourne Institute of Technology, Tech. Report TR-112-059R, 1987.

[40] M.W. Rawling and E.A. Zuk, "A Dataflow Processing Element Emulator", 4th Year Design Manual, Dep. Comm. and Elecn. Eng., Royal Melbourne Institute of Technology, 1982.

[41] C.P. Richardson, "Object Recognition using a Data-Flow Machine: Algorithms for a Laser Range-finder", M.Sc. dissertation, Dep. Computer Science, University of Manchester, 1979.

[42] C.P. Richardson, "Manipulator Control Using a Dataflow Machine", Ph.D. dissertation, Dep. Computer Science, University of Manchester, 1981.

[43] J. Sargeant, "Efficient Stored Data Structures for Dataflow Computing", Ph.D. Thesis, University of Manchester, April 1985.

[44] T. Shimada, K. Hiraki, K. Nishida, and S. Sekigucki, "Evaluation of a prototype data-flow processor of the SIGMA-1 for scientific computations", in *Proc. 13th Ann. Symp. Computer Architecture*, pp. 226-234.

[45] S. Skedzielewski and J.R.W. Glauert, "IF1, an Intermediate Form for Applicative Languages", Lawrence Livermore National Laboratory, Livermore, California, June 1984.

[46] V.P. Srini, "A Fault-Tolerant Dataflow System", *IEEE Computer*, March 1985, pp. 54-68.

[47] A. Takeuchi and K. Furukawa, "Parallel Logic Program Languages", in *Lecture Notes in Comp. Science no. 225*, G. Goos and J. Hartmanis (Eds.), 1986, pp. 242-254.

[48] K.R. Traub, "A Compiler for the MIT Tagged-Token Dataflow Architecture", M.Sc. dissertation, Dep. Elec. Eng. and Comput. Sci., Massachusetts Institute of Technology, Aug. 1986.

[49] K. Ueda, "Guarded Horn Clauses", D. Eng. dissertation, University of Tokyo, Graduate School, 1986.

[50] K.S. Weng, "Stream Oriented Computation in Recursive Data-Flow Schemas", Laboratory for Computer Science, Massachusetts Institute of Technology, Tech. Memo 68, Oct. 1975.

[51] P.J. Whitelock, "A Conventionl Language for Dataflow Computing", M.Sc. dissertation, Dep. Computer Science, University of Manchester, 1978.

[52] A. Young, "Interconnecting Dataflow Machines", M.Eng. thesis (to be submitted), Dep. Comm. and Elecn. Eng. Royal Melbourne Institute of Technology, 1988.

[53] T. Yuba and H. Kashiwagi, "Japanese Project for Supercomputing Systems", in *Parallel Computing 4*, Elsevier Science Publishers, North-Holland, 1987.