Data Flow on a Queue Machine

by

Bruno R. Preiss

A Thesis
Submitted in Conformity
with the Requirements
for the Degree of
Doctor of Philosophy
at the
University of Toronto
Abstract

Data-flow computer architectures have been suggested as a means to achieve increased computational throughput via the automatic detection and exploitation of potential parallelism. In this thesis, a new approach to data-flow, called pseudo-static data flow, is described.

One way of realizing the pseudo-static data-flow approach is to use a queue machine. A queue machine is a processing element that uses a queue as the underlying mechanism for the manipulation of operands. It can be shown that instruction sequences on a queue machine are a natural execution model for computation tasks specified by acyclic data-flow graphs. A compiler for the OCCAM programming language that decomposes source programs into acyclic data-flow graphs for execution on a queue machine is described.

The architecture of a queue-machine processing element intended for use in a multiprocessor system is proposed. A multiprocessor system architecture based on a partitioned bus configured in a ring topology that uses the queue machine processing element has been simulated in order to characterize its performance potential. The system has been simulated with up to eight processing elements and it exhibits better than linear speed-up as the number of processing elements is increased.
Acknowledgements

I would like to express my deepest gratitude to my thesis supervisor and friend, Dr. V. Carl Hamacher. He gave me the freedom to work on my own, yet he was always willing to provide assistance and encouragement.

I am indebted to the Natural Sciences and Engineering Research Council of Canada (NSERC) for its support in the form of a post-graduate scholarship. I must also acknowledge the support from the University of Toronto in the form of an Open Doctoral Fellowship. This research was also supported in part by NSERC (Canada) grant A5192.

I would also like to extend my gratitude to my friends and colleagues of the staff and students of the computer group.

It goes without saying that my family has always been very supportive and encouraging and for this I thank them. My deepest, heart-felt thanks go to my wife, Patty. She has always believed in me and supported me without question. The sacrifices she has made I cannot repay.

Finally, I dedicate this thesis to the memory of my Godfather, Edgars Vitols.
# Table of Contents

Abstract .................................................................................................................................. i
Acknowledgements .............................................................................................................. iii
Table of Contents ................................................................................................................. iv
List of Figures ..................................................................................................................... vi
List of Tables ..................................................................................................................... viii
List of Symbols ................................................................................................................... ix

1. Introduction ........................................................................................................................... 1

2. Data-Flow Computer Architecture ..................................................................................... 4
   2.1. Introduction ............................................................................................................... 4
   2.2. The Genesis of Data-Flow ....................................................................................... 5
   2.3. Interpreting Data-Flow Graphs ................................................................................. 8
   2.4. Static and Dynamic Data-Flow Architectures ........................................................ 12
   2.5. Data-Flow Machines ............................................................................................... 15
       2.5.1 MIT Static Data-Flow Architecture ............................................................ 15
       2.5.2. The LAUS System Architecture.................................................................... 17
       2.5.3. Manchester Data-Flow Architecture ........................................................... 19
       2.5.4. MIT Dynamic Data-Flow Architecture....................................................... 21
   2.6. Criticisms of Data Flow ............................................................................................ 22
   2.7. Thesis ...................................................................................................................... 25
   2.8. Outline of Research ............................................................................................... 28
   2.9. Summary ................................................................................................................. 30

3. Queue Machines .................................................................................................................. 34
   3.1. Introduction ............................................................................................................. 34
   3.2. Simple Queue Machine Execution Model .............................................................. 34
   3.3. Evaluating Expressions on a Queue Machine.......................................................... 35
   3.4. On the Pipelined Execution of Queue-Based Instruction Sequences ...................... 49
   3.5. Indexed Queue Machine Execution Model ............................................................. 52
   3.6. Executing Data-Flow Graphs on a Queue Machine............................................... 54
   3.7. Conclusions ............................................................................................................. 58

4. Data Flow on a Queue Machine .......................................................................................... 60
   4.1. Introduction ............................................................................................................. 60
   4.2. Dynamic Data-Flow Graph Splicing....................................................................... 61
   4.3. Partitioning OCCAM Programs into Acyclic Graphs............................................. 69
   4.4. Data-Flow Analysis ............................................................................................... 75
   4.5. Intercontext Communication .................................................................................. 80
   4.6. Dealing with Side-Effects ....................................................................................... 87
## List of Figures

2.1. Sample program fragment and corresponding dependence graph ...........................................7
2.2. Generic $m$-input, $n$-output data-flow actor ........................................................................9
2.3. Binary addition data-flow actor ..........................................................................................10
2.4. *Switch* and *merge* data-flow actors .............................................................................10
2.5. MIT static data-flow architecture ......................................................................................16
2.6. LAU system architecture ....................................................................................................18
2.7. Manchester data-flow architecture ....................................................................................20
2.8. MIT dynamic data-flow processor system architecture .....................................................22
2.9. MIT dynamic data-flow processing element architecture ..................................................23
2.10. Pseudo-static data-flow activity template layout ...............................................................26
3.1. The (binary) expression parse tree, level order of nodes, and level-order conjugate tree for the statement $f \leftarrow ab + \frac{(c - d)}{e}$ ..........................................................................................................................37
3.2. Trivial binary tree illustrating $a \pi_T a, a \pi_T c, a \pi_T b$ ..........................................................39
3.3. Turing algorithm for computing the level-order conjugate tree of a binary tree ...............46
3.4. Flow of operands and results in queue-based execution model and stack-based execution model ..........................................................................................................................49
3.5. The four binary expression parse trees with exactly four nodes: $-(x - y), -(x \times y), (-x) \times y,$ and $x \times (-y)$ ........................................................................................................................................51
3.6. Binary expression parse tree and binary acyclic data-flow graph for the statement $d \leftarrow \frac{a}{a + b} + (a + b)c$ ..........................................................................................................................55
4.1. Data-flow actors for intercontext communications: *Send* and *Receive* .................................................62
4.2. Augmented data-flow actors for intercontext communication: *Send* and *Receive* ..........................................................................................................................63
4.3. Sequencing communication on a channel ...........................................................................64
4.4. Data-flow actors for context generation: *rfork* and *ifork* ..................................................64
4.5. Function calls on a Queue Machine ....................................................................................66
4.6. Iteration on a Queue Machine ............................................................................................68
4.7. OCCAM program fragment illustrating dynamic process creation ..................................71
4.8. OCCAM primitives and example data-flow graphs ............................................................72
4.9. Combining data-flow graphs with the *seq* and *par* constructs .........................................73
4.10. Dynamic process creation on a queue machine .................................................................75
4.11. Algorithm for generating Use and Definition sets .............................................................80
4.12. Algorithm for live-value analysis ......................................................................................................................... 81
4.13. Algorithm for constructing a depth-first list of the nodes of a DAG ........................................................................ 84
4.14. Data-flow graph for $e \leftarrow ((a + b) \times (-e)) \div d$
and the depth-first order of its nodes ............................................................................................................................ 85
4.15. Algorithm to compute $P^{\ast}(v), I^{\ast}(v),$ and $C(v)$ for the nodes $v \in V$
of an acyclic data-flow graph....................................................................................................................................... 85
4.16. Algorithm to form the sequence of the inputs to an acyclic data-flow graph satisfying the relation $\pi$ ...................................................................................................................................................... 86
4.17. Data-flow actors with side-effects ............................................................................................................................. 88
4.18. Data-flow graphs with and without control-token arcs ................................................................................................. 90
4.19. Sequencing array accesses ......................................................................................................................................... 91
4.20. Algorithm to generate a queue machine instruction sequence from an acyclic data-flow graph .................................................. 92
4.21. Software system components ..................................................................................................................................... 95
5.1. Relationship between the operand queue, the queue pointer, the page base address and the page offset, and the register window ......................................................................................................................... 102
5.2. Queue machine processing element register set ........................................................................................................ 105
5.3. Virtual register number to physical register number translation ......................................................................................... 106
5.4. Format of the queue pointer ........................................................................................................................................ 107
5.5. Virtual register number to memory address translation .................................................................................................... 108
5.6. Basic instruction format .............................................................................................................................................. 109
5.7. Instruction format for $dup$ instructions ......................................................................................................................... 111
5.8. Contents of the queue page after $trap$ and $ftrap$ instructions ........................................................................................ 116
5.9. Three-stage instruction pipelining (simple instructions) .................................................................................................. 117
5.10. Instruction cycle usage for various instruction types ................................................................................................... 118
5.11. Queue machine processing element data path architecture .......................................................................................... 119
5.12. Decoding $QP, IR_1,$ and $IR_2$ to select window register operands and queue addresses .................................................. 121
5.13. Queue machine processing element/memory interface ................................................................................................ 123
5.14. An approach to implementing channels using dedicated message processor and cache hardware .......................................................... 125
5.15. Message cache entry format ........................................................................................................................................ 126
5.16. State transition diagram for interprocessor channel transfers .......................................................................................... 129
5.17. State transition diagram for intraprocessor channel transfers .......................................................................................... 131
5.18. 4-processor, 2-partition queue machine multiprocessor system architecture ................................................................................. 132
6.1. Organization of the queue machine multiprocessor simulator .......................................................................................... 139
6.2. Concurrent Euclid program fragment defining the state of a processing element .................................................................. 142
6.3. Queue machine multiprocessing kernel memory map.................................................................145
6.4. State transition diagram for contexts ..........................................................................................146
6.5. Queue machine multiprocessor kernel channel address map.....................................................147
6.6. Amdahl’s law ($f = 0.93$) .........................................................................................................152
6.7. Modified Amdahl’s law ($f = 0.63$, $g = 0.3$) ..........................................................................153
6.8. Graph of system throughput ratio vs. number of processors 
 for the matrix multiplication program ..........................................................................................155
6.9. Binary Recursive OCCAM procedure and the corresponding 
 non-recursive OCCAM procedure ...............................................................................................156
6.10. Graph of system throughput ratio vs. number of processors 
 for the Fast Fourier Transform program .........................................................................................158
6.11. Graph of system throughput ratio vs. number of processors 
 for the Cholesky decomposition program .......................................................................................159
6.12. Graph of system throughput ratio vs. number of processors 
 for the congruence transformation program .....................................................................................160
6.13. State transition diagram for accessible message processor states ............................................165
List of Tables

3.1 Queue machine and Stack machine instruction sequences for evaluating $f \leftarrow ab + \frac{(c-d)}{e}$ .......................................................... 36

3.2 Speed-up figures as a function of the number of nodes in the parse tree for a two-stage pipelined ALU ................................................................. 52

3.3 Speed-up as a function of the number of pipeline stages for a parse tree with 11 nodes ..................................................................................... 52

3.4 Indexed queue machine instruction sequence for evaluating $d \leftarrow \frac{a}{a+b} + (a+b)c$ .................................................................................. 54

4.1 Non-interface Intermediate Form Table (IFT) entries ........................................... 77

4.2 Interface Intermediate Form Table (IFT) entries .................................................. 78

4.3 Sample OCCAM program fragment and corresponding Intermediate Form Table ...................................................................................... 79

4.4 $P^*(v)$, $I^*(v)$, and $C(v)$ for the nodes $v$ of the data-flow graph of Figure 4.14.(a). .................................................................................. 86

4.5 $W(v)$ for the input nodes $v$ of the data-flow graph of Figure 4.14.(a). .................. 87

5.1 Table of source operand modes .............................................................................. 109

5.2 Queue machine processing element instruction set ............................................... 113

5.3 Message cache entry state transition tables for $fetch - and - \phi_1$ and $fetch - and - \phi_2$ operations .............................................................. 127

5.4 Message cache entry state transition tables for $send$ and $receive$ operations .............................................................. 128

6.1 Table of queue machine multiprocessing kernel entry points ................................ 148

6.2 Statistics for the matrix multiplication program .................................................... 154

6.3 Statistics for the Fast Fourier Transform program ................................................ 157

6.4 Statistics for the Cholesky decomposition program .............................................. 158

6.5 Statistics for the congruence transformation program ......................................... 160

6.6 Table of compiler optimization speed-up factors .................................................. 162

6.7 State transition table for accessible message processor states ......................... 164
List of Symbols

\( T \) a binary tree ................................................................. 38
\( N(T) \) the set of nodes of tree \( T \) ...................................................... 38
\( S(T) \) the set of all subtrees of tree \( T \) .................................................. 38
\( \Gamma_T(n) \) the level of node \( n \) in tree \( T \) .................................................. 38
\( L_T(n) \) the set of nodes of the left subtree of tree \( T \in S(T) \) whose root is \( n \) ................................................................. 38
\( R_T(n) \) the set of nodes of the right subtree of tree \( T \in S(T) \) whose root is \( n \) ................................................................. 38
\( D_T(n) \) the set of descendent nodes of node \( n \) in tree \( T \) ................................................................. 38
\( \pi_T \) a level-order precedence relation on the nodes of a binary tree ................................................................. 38
\( \Pi(T) \) the level-order traversal of binary tree \( T \) ................................................................. 41
\( O_0 \) the set of nullary operators ................................................................. 41
\( O_1 \) the set of unary operators ................................................................. 41
\( O_2 \) the set of binary operators ................................................................. 41
\( A(n) \) the arity of operator \( n \) ................................................................. 41
\( P \) a binary expression parse tree ................................................................. 42
\( I \) a sequence of operators ................................................................. 42
\( E(I) \) the evaluation of an operator sequence on a queue machine ................................................................. 42
\( \lambda_T(n) \) the left immediate descendent of node \( n \) in tree \( T \) ................................................................. 42
\( \rho_T(n) \) the right immediate descendent of node \( n \) in tree \( T \) ................................................................. 42
\( \delta(T) \) the level-order conjugate of binary tree \( T \) ................................................................. 42
\( i(T) \) the in-order traversal of a binary tree \( T \) ................................................................. 44
\( G \) a directed graph ................................................................. 54
\( p \) a path in a directed graph ................................................................. 54
\( DAG \) directed acyclic graph ................................................................. 55
\( G \) a binary, acyclic data-flow graph ................................................................. 55
\( \pi_G \) a relation on the vertices of a data-flow graph ................................................................. 56
\( P(v) \) the set of immediate predecessors of node \( v \) of a \( DAG \) ................................................................. 82
\( S(v) \) the set of immediate successors of node \( v \) of a \( DAG \) ................................................................. 82
\( P^*(v) \) the set of all predecessors of node \( v \) of a \( DAG \) ................................................................. 82
\( C(v) \) the cost of computation of node \( v \) ................................................................. 83
\( I^*(v) \) the required input set of node \( v \) of a \( DAG \) ................................................................. 83
\( \pi_I \) a relation on the inputs to a data-flow graph ................................................................. 83

Chapter 1
Introduction

Data-flow architectures have been proposed as a way to build faster and more powerful computers. The main attraction of data-flow architectures is their potential for automatically detecting and exploiting parallelism in a computation. However, data-flow research has had only qualified success and limited acceptance. It is believed that this is because data-flow architectures are a radical departure from conventional, Von Neumann architectures. As a result, data-flow architectures are not buoyed by the same body of knowledge as conventional architectures. Furthermore, many issues in the design of data-flow systems remain open questions.

The main research goal of this thesis is the development of a realistic and efficient multiprocessor computer architecture based on the data-flow execution model. It is implicitly
recognized in this thesis that those architectures that directly implement the data-flow execution model have achieved limited success in attaining the goals of efficiency and maximum throughput. For this reason, the processing element and multiprocessor system architectures proposed in this thesis resemble in many ways conventional, Von Neumann systems. Nonetheless, the system proposed herein differs from a conventional architecture in that it has a formal, theoretical underpinning — the data-flow execution model.

The thesis begins in Chapter 2 with a brief survey of data-flow research to date. Data-flow architectures are based on the execution of computations represented by data-flow graphs. Such graphical representations were first used in certain optimizing compilers.

Data-flow architectures are classified as either static or dynamic. Each class has its advantages and limitations. A new approach, called pseudo-static data-flow, is proposed in Chapter 2. This approach attempts to incorporate the benefits of both static and dynamic data-flow, without the associated costs.

Finally, it is observed that all data-flow architectures have a common architectural feature — a circular pipeline with queueing of operands and results. Thus, a processing element architecture that directly incorporates an operand queue is proposed. Such processing elements will be called queue machines.

The purpose of Chapter 3 is to provide a mathematically sound definition of the queue machine execution model. First, a simple queue machine execution model is defined. It will be shown that such a machine is equally as powerful as a stack machine for the evaluation of arbitrary expressions. An algorithm for the generation of instruction sequences for a simple queue machine will be presented.

Next, a more powerful queue machine model, called an indexed queue machine, will be described. It will be shown that instruction sequences on an indexed queue machine are the natural execution model for a computation task expressed by an acyclic data-flow graph.

In Chapter 4, the issues associated with programming an indexed queue machine will be discussed. The basis of the approach is the partitioning of programs into a collection of acyclic data-flow graphs that are connected together at execution time via a dynamic data-flow graph splicing mechanism.

A compiler that translates OCCAM source programs into indexed queue machine instruction sequences will be described. In particular, the method of partitioning programs and constructing data-flow graphs will be presented. In addition, certain heuristic optimization algorithms will be discussed.

In Chapter 5, the architecture of a processing element that implements the indexed queue machine execution model is proposed. The instruction set of the processing element is based on the novel use of virtual register numbers rather than physical register numbers to refer to registers. This allows the efficient execution of indexed queue machine instruction sequences.

Next, a multiprocessor architecture that uses the indexed queue machine as its basic processing element is proposed. This architecture is based on a shared, segmented bus connected in a ring topology. An efficient approach to implementing interprocessor communications using dedicated message handler hardware will be described.

Finally, a simulation study of the proposed multiprocessor system is presented in Chapter 6. This simulation study involved the implementation of a detailed simulation of the proposed system that can be used to execute programs generated by the OCCAM compiler described above. Thus, by simulating the execution of actual test programs, the behaviour of the proposed multiprocessor system can be characterized.
Chapter 2
Data-Flow Computer Architecture

0.1. Introduction

The purpose of this chapter is to describe the background and motivation for the research presented in this thesis. The main research goal is the development of a realistic and efficient multiprocessor computer architecture and system based on the data-flow execution model.

First, the genesis of the data-flow execution model will be described. This model was developed with the goal of automatically exploiting the parallelism inherent in arbitrary computer programs. The roots of the design of the data-flow execution model are related to procedures used in compilers to facilitate code optimization. These procedures will be briefly described.

Data-flow architectures are designed to directly execute programs written in a graphical base language. A data-flow program graph is a directed graph whose nodes represent operators and whose arcs represent the flow of operands. The basic mechanisms involved in the interpretation of such graphs will be described.

Data-flow architectures can be categorized as either static or dynamic depending on the program representation scheme and the details of the execution mechanisms. The characteristics of these two categories will be compared.

Various different data-flow computers have been proposed and built by other researchers. The architectures of four such machines will be described. These data-flow systems have achieved limited success in attaining the goal of high-speed or high-throughput computation. Some of the reasons for this limited success will be presented together with criticisms of the data-flow approach to execution.

Finally, some new ideas and proposals for a data-flow-based system will be presented. These ideas constitute the essential part of this thesis. The research work done to develop and confirm these ideas will be outlined in this chapter and presented in detail in the following chapters.

0.2. The Genesis of Data-Flow

A computer program can be described as a partially-ordered set of high-level functions [Gajski 1981]. This partial order implies that the execution sequence of certain subsets of instructions can be altered without affecting the function computed by the program. In particular, those subsets of instructions for which no ordering relation holds can be executed in parallel.

High performance computer architectures can exploit such parallelism by using multiple operation units, executing in parallel, to increase computation throughput and/or decrease the total computation time.

Parallelism can be classified by scope from microscopic to macroscopic and by type as regular or irregular. For example, vector and array (SIMD) machines take advantage of the regular, microscopic parallelism of numerical matrix computation. Such algorithms contain non-ordered subsets of instructions whose elements all have the same operator but different operands. Multi-processor (MIMD) architectures take advantage of irregular, macroscopic parallelism by performing concurrent tasks or processes. Such programs are partitioned (by the programmer) into partially-ordered subsets of instructions such that no ordering relation holds between elements of the partition. Finally, both regular and irregular, microscopic parallelism is exploited in pipelined architectures that can execute several machine instructions in parallel.

Data-flow architectures have evolved in an attempt to efficiently exploit more general forms of parallelism. Whereas SIMD and MIMD architectures each execute a certain class of problems efficiently, data-flow systems are intended to take advantage of all forms of parallelism. Such
architectures are intended to automatically recognize and exploit microscopic to macroscopic and both regular and irregular parallelism, without suffering serious performance degradation when executing programs exhibiting the specialized forms of parallelism discussed above.

The genesis of data-flow systems can be traced to the notion of data- and control-flow analysis — procedures used in compilers to perform optimizing code transformations and to minimize the usage of temporary memory locations by optimizing the usage of CPU registers. Data-flow computers either directly or indirectly execute programs expressed in a graphical base language of data-flow graphs that are derived from high-level source programs by data- and control-flow analysis.

Data- and control-flow analysis is used to determine the instruction sequencing constraints or partial order implied by the source program. The analysis procedure constructs a directed graph which represents the source program. The nodes of the graph correspond to atomic actions and the arcs correspond to sequencing constraints (or dependences). Such graphs reveal concurrency at the operator, function, and program module levels [McGraw 1980].

There are two types of dependence that arise in programs — data dependences and control dependences [Gajski 1982]. Data dependences ensure the integrity of values and results and guarantee that the correct computation is performed. Control dependences are used to sequence operations with side effects and to facilitate conditional execution, iteration, and procedure calls. Data dependences are further divided into three categories — flow dependences (also called data dependences [Heuft 1982]), (data) output dependences or (memory) space dependences, and (data) antidependences.

As an example, Figure 0.1 shows a simple program fragment and the corresponding dependence graph. The nodes of the graph correspond to atomic operations and the arcs represent dependences. Flow dependence arcs are constructed by placing directed arcs from nodes in which a value is assigned to a variable to nodes in which that value is used. These arcs represent the flow of data from the point at which it is computed to the point where it is used. Antidependence arcs are constructed by placing arcs from nodes in which the value of a variable is used to nodes in which the variable is assigned a new value. These arcs represent the requirement that a value must not be destroyed (overwritten) before completing the operations that require the value. Data output dependence arcs are constructed from nodes in which a value is assigned to a variable to the next node in which a new value is assigned to the same variable. These arcs are important in the cyclic graphs that are used to represent loops. They ensure that values are assigned to variables in the correct sequence. Finally, control dependence arcs are constructed by placing a directed arc between nodes having side-effects in order to guarantee the sequencing implied by the program code.

Whereas some of the arcs of the dependence graph represent the computation itself, others are required by the underlying execution mechanism. In particular, output dependence and antidependence arcs result from the Von Neumann execution model in which variables are stored in memory locations. Assigning values to variables has the (side) effect of altering the state of memory. The approach taken in data-flow architectures is to prohibit operations having side effects — all operations are purely functional. As a result, the dependence graphs contain only flow dependences. Such graphs form the base language representation of programs and are called data-flow graphs.

0.3. Interpreting Data-Flow Graphs

Data-flow architectures are designed to execute programs specified in a graphical base language. Graphical base languages formalize the syntax and semantics of data-flow graphs. In effect, well-formed data-flow graphs constitute the equivalent of assembly language programs for data-flow architectures. The details of the syntax and semantics of the graphical base language
\textbf{Figure 0.1.}
Sample program fragment and corresponding dependence graph.

and the rules for constructing well-formed data-flow graphs depend on the architecture of the machine for which the language is intended. Nevertheless, all data-flow program graph schemes are based on a number of common concepts. These concepts will be described in this section.

A data-flow graph is a graphical representation of a computation. In general, such graphs are directed graphs whose nodes represent atomic operations and whose arcs represent the flow of information [Dennis 1974]. The nodes of the data-flow graph are called \textit{actors}. The arcs connecting the actors represent data paths along which flow data items called \textit{tokens}. The manner in which the computation proceeds is governed by a set of rules called \textit{firing rules}. For each actor,
the firing rules describe the conditions under which the actor may execute. An actor whose firing
rule predicate is satisfied executes by absorbing tokens from (some or all of) its input arcs and
producing tokens on (some or all of) its output arcs. The concepts of actor, token, and firing rule
are the basic elements of the data-flow execution model. This model is based on two underlying
principles:
1.  *Asynchrony* — an operation may (but need not) proceed when its firing rules are satisfied;
i.e., when its operands are available.
2.  *Functionality* — all operations are functional (in the mathematical sense); i.e., they are free
from side effects [Gajski 1982].

A generic $m$-input, $n$-output data-flow actor is shown in Figure 0.2. In general, an actor can
have any finite number of inputs and can compute any finite number of functions. As a rule, most
data-flow architectures limit the number of inputs and outputs that any given actor may have.

![Figure 0.2: Generic $m$-input, $n$-output data-flow actor.](image)

The most common actors are unary and binary arithmetic and logic actors. As an example,
the binary addition actor is shown in Figure 0.3. This actor has two inputs, $x_1$ and $x_2$, and one
output, $f_1(x_1, x_2) = x_1 + x_2$. The firing rule for this actor is the predicate $p(x_1) \land p(x_2)$, where
$p(x_i)$ is true if and only if a token is present on input arc $x_i$.

In general, a data-flow actor does not have to consume all its input tokens; nor does it have
to produce a result on all its output arcs. Two such actors, *switch* and *merge*, arise when implement-
ing conditional execution and iteration. The switch actor, shown in Figure 0.4,(a), places a
token on only one of its outputs. The firing rule predicate for the switch actor is $p(x_1) \land p(x_2)$.
The actor duplicates the input $x_1$ on output $f_1$ if the control input $x_2$ has the Boolean value true.
The input $x_1$ is duplicated on output $f_2$ if the control input $x_2$ has the Boolean value false.

The merge actor, shown in Figure 0.4.(b), has the firing rule predicate $(p(x_1) \land p(x_3) \land x_3) \lor (p(x_2) \land p(x_3) \land \neg x_3)$. The result of this actor is a duplicate of input token
$x_1$ or $x_2$ on output arc $f_1$ depending on whether the Boolean-valued control input $x_3$ has value
true or false, respectively. Note that this actor does not require tokens on all its inputs in order to
fire.
In order to simplify the execution mechanism needed to interpret data-flow graphs, all actors should obey the same firing rules and consume all their input tokens. Since the merge actor can be simulated using switch actors, it is possible to construct an instruction (actor) set having uniform firing rules [Arvind 1978].

The execution control mechanism for interpreting data-flow graphs is classified as data-driven execution. That is, the execution proceeds solely on the basis of availability of operands (data). However, there are two ways in which an actor can interact with its predecessors and successors (collectively called the environment) based on whether the arcs have a finite capacity of one or conceptually infinite capacity.

According to the simplest mechanism, an actor fires when its operands are made available by the environment. The results of the computation are passed to the environment on the output arcs. No throttling mechanism exists to prevent data tokens produced by a fast node from piling up on the input of a slow node. Such pile-ups can occur in cyclic data-flow graphs containing
paths of different lengths. An implementation of this execution mechanism must either provide for arcs with conceptually infinite capacity or it must prevent pile-ups. Pile-ups can be prevented in cyclic graphs by performing certain graph transformations that do not change the function computed by the graph but cause all paths to be of equal length [Brock 1979].

According to the second method of data-driven computation, an actor fires when its operands are made available by the environment and it has received a signal from the environment that its output arcs are empty. The actor fires; places the results on its output arcs; and informs the environment that the data on its input arcs have been consumed. The advantage of this execution mechanism is the property that the arcs only require a token-carrying capacity of one. The disadvantage of this mechanism is the need for signalling the environment that input data have been consumed. In effect, each arc in the graph has associated with it an arc in the reverse direction which carries the data-taken acknowledgement signals. Thus, the number of arcs and tokens has been effectively doubled.

0.4. Static and Dynamic Data-Flow Architectures

Data-flow systems can be classified as either static architectures or dynamic architectures. Static and dynamic architectures differ in the following ways: First, in static architectures, the program graph is loaded into memory in completed form before the program begins execution (i.e., no run-time loader); whereas in dynamic architectures, nodes can be created at run time (e.g., to support loop unravelling and recursion). Second, in static architectures, at most one instance of an actor may be enabled for firing at a time. Dynamic architectures support several instances of an actor firing simultaneously [Srini 1986]. Finally, static architectures use the same storage space for instructions (actors) and data (tokens) (i.e., impure code). Dynamic architectures use physically separate memories for instructions (i.e., pure code) and data [Todd 1982].

In static data-flow architectures, the data-flow program graph is represented as a multiply-linked collection of items called activity templates. Each activity template corresponds to an actor of the data-flow program graph. An activity template is a triple consisting of i) an operation code, ii) a set of operand slots, and iii) a (destination) pointer list [Dennis 1980b]. The operand slots correspond to the input arcs of the actor. Each slot is a reserved memory location into which the actor’s predecessor stores a token. Thus, arcs have a maximum token-carrying capacity of one. The elements of the destination pointer list correspond to the output arcs of the actor. These pointers indicate the operand slots in other activity templates into which the result of this actor is to be stored. The operation code field of the actor specifies the function computed by the actor. It also directly or implicitly specifies the number of operand slots, the number of destination pointers, and the firing rule for the given actor. The basic execution cycle involves: identifying an activity template whose firing rule is satisfied (i.e., all its operand slots are full and the operand slots of its successors to which it is connected are empty); computing the function specified by the operation code; and storing the result in the operand slots specified by the destination pointers.

The finite arc capacity and impurity of code in the static data-flow systems result in data-flow program graphs that are not reentrant. This constraint affects both iteration and function (subroutine) calling mechanisms. Automatic run-time loop unravelling is not supported by static data-flow systems. Functions can be made reentrant by either code-duplication or code-sharing mechanisms.

Data-flow programs containing function calls consist of a hierarchy of data-flow graphs. The hierarchy is constructed by associating a data-flow program graph with a node in another data-flow graph. This node can then be expanded (i.e., the associated graph is duplicated) either at compilation time or at execution time [Davis 1982]. In order to support recursive functions, execution-time expansion must be used. This is accomplished by introducing a special actor
which takes a pointer to a function and an argument list. This actor has the (controlled) side effect of duplicating the activity templates of the specified function, binding the arguments to the input arcs of the function, and linking the output arcs of the function to the calling domain [Miranker 1976].

The code-sharing approach to function reentrancy in static architectures is based on pipelined execution of data-flow graphs. This mechanism relies on the fact that data-flow graphs in static architectures have the first-in, first-out (FIFO) property. A sequence of arguments can be submitted to the function, one set at a time. The function will return a sequence of results in the same order. An implementation of this mechanism is described in [Todd 1982].

In dynamic data-flow architectures, data-flow program graphs are also represented as a collection of multiply-linked activity templates. In this case, activity templates consist of i) an operation code and ii) a (destination) pointer list. No space is reserved in the activity template for storing (input) data tokens. Instead, tokens are stored elsewhere. Each token is tagged with information that identifies the arc on which the token conceptually resides. Thus, a token consists of a data field and a tag field. As a result, the token-carrying capacity of the arcs is not constrained. As in static architectures, the elements of the destination pointer list correspond to the output arcs of the actor. These pointers specify the address of the successor activity template and also indicate to which input of the actor the token is to be sent. (In general, the functions computed by the actors need not be commutative). In addition, the pointer may also specify how many tokens are required by the successor actor in order to fire (to facilitate efficient evaluation of its firing rules). The basic execution cycle involves locating actors whose input arcs are full (by matching tag fields of tokens); computing the function specified by the operation code (using the value fields of tokens); and forming the value parts of the output tokens from the result of the computation and the tag parts of the output tokens from the elements of the destination list.

Since there are no operand slots in the activity templates, the code is pure. Furthermore, the token-carrying capacity of arcs is not constrained by the representation of the data-flow graph. As a result, loop unravelling and reentrant functions are easily implemented in dynamic data-flow architectures.

To support reentrancy and unravelling, the tag field of tokens is augmented with information that identifies the context in which the token is executing. The augmented tag field is called an activity name. The activity name specifies i) the actor (and which of its inputs) to which the token is destined, ii) the iteration number of the loop to which the token belongs (if it is inside a loop), and iii) the activity name of the calling function if the token belongs to a called function [Arvind 1980b]. Note that the activity name is recursively defined. In effect, every token carries with it information that is analogous to the processor stack in conventional, Von Neumann architectures!

Iteration and function calls are accomplished in dynamic architectures by using actors that manipulate the tag fields of tokens [Watson 1979]. For example, iteration is accomplished using an actor that takes tokens from the bottom of a loop, increments the iteration number field of the activity name of the token, and injects the token back into the top of the loop. Arguments are transmitted to functions and results are received from functions using actors that, in effect, push and pop contexts from activity names.

0.5. Data-Flow Machines

A number of data-flow architectures have been proposed or are under development in Britain, Canada, France, Japan, and the United States. Recent articles provide an excellent summary and overview of the various approaches to implementing the data-flow execution model [Sriini 1986] and [Treleaven 1982]. In this section, the architectures of four data-flow machines — two static and two dynamic — will be described.
0.5.1. MIT Static Data-Flow Architecture

The MIT static data-flow architecture is a packet communication system based essentially on two hardware elements — routing networks and autonomous processing/memory elements [Dennis 1974], [Dennis 1979a], [Dennis 1980a], [Dennis 1980b], and [Dennis 1984a]. A block diagram of the architecture is shown in Figure 0.5.

![Figure 0.5](image)

MIT static data-flow architecture.

The instruction memory of this machine consists of a number of units called cell blocks. Each cell block contains many instruction cells (activity templates). A program is loaded from a host computer into the instruction cells prior to execution. The host injects into the system any
data tokens needed to initiate execution. When a cell block receives a token, it stores the token in the appropriate instruction cell. When all the operands have arrived in an instruction cell, the cell block containing that cell forms an instruction packet containing an operation code, the operands, and destination pointers. This packet is then submitted to the arbitration network.

The arbitration network routes the instruction packet to an appropriate idle processing element (they need not all be identical). If all processing elements are busy, the instruction packet is queued within the arbitration network. The processing element evaluates the function specified in the instruction packet and forms a number of data (token) packets (one for each destination pointer). Each data packet consists of the result token and a destination pointer. The data packets are routed by the distribution network to the appropriate cell block. The distribution and arbitration networks are pipelined networks containing FIFO queues. It is assumed that they have enough capacity to hold all data and instruction packets so that blockage will not occur.

A small, four-processor prototype has been built. The processing elements were implemented using bit-slice technology. The routing elements were constructed using custom, LSI 2×2 routers to implement a Banyan network [Dennis 1980a].

0.5.2. The LAU System Architecture

The LAU system is a static data-flow architecture based on a single assignment language (language à assignment unique [Plas 1976]). An interesting consequence of the single-assignment approach is that the resulting data-flow program graphs are acyclic. Special machine instructions are used to implement iteration, conditional execution, and function calls. A block diagram of the LAU system architecture is shown in Figure 0.6.

Source programs are compiled into data-flow graphs in the host which then loads the program into the memory unit. For each instruction in the memory unit there is a control part stored in the control unit (at the same address). The control part contains bits that specify whether the corresponding instruction’s operands are available and whether the context in which the instruction is found is enabled for execution. The control unit performs a (pseudo) associative search of the control parts of instructions to determine which instructions are able to execute. The addresses of such instructions are sent to the memory unit.

The memory unit retrieves the instruction and places it in the instruction queue. An instruction consists of an operation code, the addresses of source operands, and the address of the result. Instructions are removed from the queue by the processing elements. The processing elements retrieve the operands from the memory, compute the result, and store the result back into memory. The processing element then sets the appropriate bits in the control unit to enable subsequent instructions.

Due to the static nature of the architecture and the restriction to strictly acyclic graphs, special methods are used to implement iteration and function calls. Iteration is accomplished using a special instruction that injects tokens into the top of the acyclic graph corresponding to the loop body and an instruction that waits for the computation of the loop body to be completed. Then, if the loop is to be repeated, the values are injected back into the top of the graph. In this way, exactly one instance of a loop body is active at a time. Thus, loop unravelling is impossible. Function calls are implemented using a similar inject-and-wait scheme. Parallel calls to the same function can be achieved by having the programmer specify at compile time that several copies of the function are to be loaded. Clearly, recursion is impossible.

A prototype system consisting of 32 processing elements has been implemented. In the prototype, the processing elements share a common bus to access the instruction queue and the memory and control units. The memory system consists of eight parallel memory banks to allow simultaneous memory access by several processing elements. The instruction queue has the
Figure 0.6.

LAU system architecture.

capacity to hold 128 instructions.

0.5.3. Manchester Data-Flow Architecture

The Manchester machine is an example of a dynamic data-flow architecture that uses tagged tokens [Watson 1979], [Watson 1982], and [Gurd 1985]. The basic architecture of the machine is a circular pipeline. A block diagram of the architecture is shown in Figure 0.7.

The Manchester architecture consists of a circular pipeline of five elements — processing unit, switch, token queue, matching store, and node store. The processing unit consists of a number of processing elements that execute in parallel. Each processing unit produces (tagged) tokens which are sent, via the switch, to a token queue. The switch also allows the host processor to inject tokens into the token queue and to receive result tokens from the processing units.

Tokens are removed from the token queue by the matching store. The purpose of the matching store is to group together tokens that are destined for the same actor and have matching tags. Actors can have either one or two input arcs. Each token is marked with a bit that indicates whether it has a partner. Tokens destined for single-input actors bypass the matching store and proceed to the node store. Tokens destined for two-input actors must be matched. When such a token arrives, an associative search of the token store is done. If a partner is found, the pair of tokens is forwarded to the node store. Otherwise, the unmatched token is stored in the matching store.

The node store contains a data-flow program graph that was downloaded from the host. When tokens arrive, the corresponding activity template is retrieved from the node store and an
Manchester data-flow architecture.

An executable package consisting of the tokens and the activity template is forwarded to the processing unit. There the specified function is computed and the result tokens are formed.

A prototype system has been built and tested. The prototype contains 14 processing elements implemented using bit-slice technology. The matching store is implemented using a hardware tag hashing scheme to address 16 parallel read/write memory banks. The prototype has achieved maximum computation rates of between 1 and 2 million instructions per second [Gurd 1985].
0.5.4. MIT Dynamic Data-Flow Architecture

The MIT dynamic data-flow machine is a multiprocessor architecture based on a packet communication network [Arvind 1978], [Arvind 1980a], [Arvind 1980b], [Arvind 1980c], [Arvind 1981], and [Arvind 1982]. (It is also called a U-interpreter [Arvind 1982] and the Irvine data-flow machine [Treleaven 1982].) This architecture uses tagged tokens to support loop unravelling and program reentrancy. A block diagram of the system architecture is shown in Figure 0.8. The architecture simply consists of \( n \) processing elements together with an \( n \times n \) packet switching network.

The architecture of the processing elements is a circular pipeline consisting of a waiting-matching section, an instruction-fetch section, and a service section, as shown in Figure 0.9. Note that the architecture of the processing element is very similar to that of the Manchester machine. The waiting-matching section performs the functions of the matching store and token queue. The instruction-fetch section accepts tokens from the waiting-matching section and retrieves the corresponding template from the instruction memory. Finally, the service section computes the required function and transmits the result tokens either back to the local waiting-matching section or to a remote processing element using the network.

In this architecture, instruction memory is distributed among all the processing elements. this means that the activity templates comprising the data-flow graph must be distributed among
the processing elements. A scheduler, which runs on the host, attempts to determine the optimum distribution of activity templates at compilation time.

This proposed architecture has been extensively simulated and emulated. In addition, work on a custom VLSI implementation was reported in [Arvind 1981]. However, a recent status report on the project does not mention a hardware prototype [Srini 1986].

0.6. Criticisms of Data Flow

The data-flow execution model has been suggested as a means to automatically detect and exploit parallelism to effect high-speed or high-throughput computation. However, practical data-flow architectures have achieved only limited success in accomplishing this goal. A number of problems with the data-flow execution model have been identified. (For example, [Gajski 1982] is an excellent critique of data-flow architectures). These problems manifest themselves in practical machines as an inability to achieve the maximum possible increase in speed or throughput. In this section, some of the major problems with the data-flow execution model will be discussed.

First, data-flow machines cannot execute sequential code as efficiently as conventional, Von Neumann machines. This is due to the inability of data-flow machines to utilize instruction look-
ahead and pipelining when executing sequential code. In the data-flow execution model, an instruction can begin to execute as soon as its operands are available. In sequential code, however, the operands for one instruction are not available until the preceding instruction has completed executing. Since a data-flow machine cannot determine which instruction will execute next until the operands for the instruction have been computed, the next instruction and its operands cannot be prefetched. However, Von Neumann architectures use a program counter to specify the next instruction to be executed. As a result, pipelined execution is possible. For this reason, data-flow machines generally execute sequential code more slowly.

The preceding problem is compounded by the fact that many data-flow architectures are based on a long pipeline of functional units consisting of an instruction memory, a token matching unit, an instruction or token queue, and execution units. This long pipeline has the effect of slowing sequential execution even more. More importantly, it creates a need for a massive amount of parallelism in order to keep all the functional units of the machine busy. (For example, a 64-processor version of the MIT dynamic data-flow machine requires parallelism of degree 640 to keep all functional units busy [Gajski 1982].) The kinds of programs that exhibit the requisite parallelism invariably involve vector and matrix algorithms. Architectures that exploit such regular parallelism already exist (e.g., SIMD machines).

Another source of problems in data-flow architectures is the manner in which data are handled. Data-flow system models have no explicit storage mechanism. Instead, all values in use in a computation are carried by tokens that circulate in the system. This problem is exacerbated in dynamic data-flow architectures where there is no binding between a data token and some memory location. Such architectures require large associative matching stores and long token queues.

Since data tokens are always circulating in data-flow machines, data-flow processing elements are essentially memory-to-memory machines. That is, all the operands of an instruction executed by a given data-flow processing element come from outside that element; all the results computed by a given processing element leave that element. As a result, processing elements cannot take advantage of local storage registers to eliminate data transfers between functional units.

The inability of data-flow systems to handle vectors, arrays, and data structures efficiently is another criticism of the data-flow execution model. This inefficiency is a direct result of the functionality principle of the data-flow execution model — actors do not have side effects. For this reason, actors that manipulate data structures can not alter existing structures. Instead, they must create new instances of the complete data structures. This precludes element-by-element updating of a single vector or array stored in a fixed area of some memory.

Finally, there is the practical problem of debugging a data-flow system. This is a problem for both the diagnosis and maintenance of the data-flow system hardware as well as the development of software. Data-flow machines have decentralized control mechanisms as opposed to the centralized control of conventional, Von Neumann machines. Debugging is a potentially difficult task in a machine that has no program counter and whose execution is essentially non-deterministic [Gajski 1982].

0.7. Thesis

In this section, some new ideas for a data-flow-based execution model and processing element architecture will be presented. These ideas represent an attempt to redress some of the criticisms of data-flow systems described in the preceding section. The basic approach in the development of these ideas has been to use elements of conventional, Von Neumann architectures to supplant some local aspects of the data-flow execution model.
This work has evolved from a new data-flow execution model and program representation scheme, originally described in [Preiss 1984], which can best be described as pseudo-static. In pseudo-static data-flow architectures, the data-flow program graph is represented as a multiply-linked collection of activity templates. As in static architectures, the activity templates consist of i) an operation code part, ii) operand slots, and iii) destination pointers. Pseudo-static architectures differ from static ones in that the activity template is not stored in contiguous memory locations. Instead, the operation code and destination pointers are stored in an instruction space and the operand slots are stored in a data space (see Figure 0.10). There is a one-to-one correspondence (namely identity) between activity template addresses in instruction and data spaces. By associating several data spaces with one instruction space, a graph can be made reentrant. The advantage of this scheme is that reentrancy is accomplished without the overhead of code copying or tagged tokens.

![Figure 0.10.](image)

Pseudo-static data-flow activity template layout.

In this thesis, it is proposed that the identity relationship between addresses in instruction and data spaces is not needed (as reported in [Preiss 1985]). Instead, the data space can be considered as an indexed queue of operands. It is claimed that an execution model that uses an indexed queue as the basic mechanism for handling operands is a natural way to interpret data-flow program graphs.

One of the problems with data-flow architectures is the need to dynamically evaluate firing rules (at execution time) to determine when an actor may fire. This means that instruction look-ahead and overlapped execution is not possible. It is proposed that all actors obey the same firing rules: An actor is enabled when all its input arcs contain a token; it consumes all the input tokens; and then it produces a result on all its output arcs. Furthermore, it is proposed that programs use strictly acyclic data-flow graphs. As a result, the need for dynamically evaluating firing rules has been eliminated. This is because all the arcs of a data-flow program graph will receive exactly one token during the course of a computation (in a given context). This means that the actor execution sequence can be determined statically at compilation time. With a statically determined actor (instruction) sequence, it becomes possible to take advantage of the techniques of instruction look-ahead and overlapped execution of conventional, Von Neumann architectures.
The restriction that only acyclic graphs are used and that all arcs of the graph receive exactly one token during the course of the computation means that special techniques are needed to implement iteration and function calls, as well as conditional execution. In the pseudo-static architecture, program reentrancy is accomplished by associating more than one data space with a given instruction space. It is proposed that the same technique be used to associate more than one operand queue with an acyclic data-flow graph. The tokens found in a given operand queue correspond to a given context of the computation; i.e., they belong to a particular iteration of a loop, a particular function call, or a particular outcome of a conditional. This technique is similar to that used in the LAU system architecture [Plas 1976], except the pseudo-static execution mechanism proposed here allows reentrancy. Note that this method also has an advantage over dynamic data-flow architectures in that the context is not carried by tagged tokens.

Another problem with data-flow systems is their inability to deal efficiently with data structures. This inability is the result of imposing the strict functional semantics to all data-flow actors. In effect, the programs executed by data-flow machines can only exhibit (data) flow dependences. Since limited success has been achieved in this approach, it is proposed that the restriction to only flow dependences be eliminated. Instead, it is proposed that, in addition to flow dependences, data-flow graphs should also contain arcs representing data output dependences and data antidependences (for data structure manipulations only). These arcs can then carry special tokens called control tokens whose sole purpose is to sequence the data structure manipulations. In effect, data-flow graphs can now have controlled side-effects. The same technique can be used to control other actions having side effects such as input and output of data. As a result of the use of control tokens to control side effects, the proposed system can use conventional random-access, read/write memory for data structure storage.

Finally, a data-flow processing element that uses an operand queue as the basic mechanism for manipulating data (tokens) is proposed. Such a design is motivated by two factors. First, it is a natural consequence of the pseudo-static, indexed queue machine model for interpreting data-flow graphs. Second, it reflects the fact that operand (or instruction) queues are present in virtually all other data-flow architectures. Thus, an operand queue appears to be an integral part of data-flow machines. Furthermore, by using an operand queue as a component of the processing element itself, the benefits of having local storage (i.e., registers) can be exploited. In particular, it is possible to execute data-flow graphs in a register-to-register rather than memory-to-memory mode.

Although the execution mechanisms proposed in this thesis violate some of the assumptions of the pure data-flow model, the basic goal of dynamically exploiting the available parallelism in a computational task is retained.

0.8. Outline of Research

In the preceding section, a new approach to the design of a data-flow architecture that attempts to solve some of the problems with other data-flow architectures was introduced. In this section, the work done to develop and support these ideas will be briefly outlined. This work consists of four main phases:

1. Some theoretical work was done to establish a mathematically sound foundation for the queue-based execution model.
2. Various issues associated with generating software for queue machines were studied.
3. A paper design of a processing element architecture that implements the queue machine execution model was made.
4. A software simulation of a queue-machine-based multiprocessor system was done to confirm the concepts introduced in this thesis and to obtain performance estimates.
The first phase of the research involved the development of a mathematically sound formal specification of the queue-based execution model. This work is essential because it establishes a solid foundation for the subsequent work in that it proves that the basic underlying execution mechanisms are correct. This work involved the specification of two queue-based execution models. The first of these models is the simple queue machine execution model. This model is used to prove that queue-based execution can be used to evaluate arbitrary expressions and is, therefore, as powerful as other execution mechanisms such as stack-based ones. The second of these models is the indexed queue machine execution model. This model is shown to be a natural mechanism for executing data-flow graphs. In particular, it is proved that acyclic data-flow graphs can be considered as generators of instruction sequences for an indexed queue machine.

The second phase of the research consists of a study of the issues associated with constructing programs for queue machines. A mechanism necessary for implementing function calls, iteration, and conditional execution was developed. This mechanism, called dynamic data-flow graph splicing, involves the dynamic (i.e., at execution time) splicing together of acyclic data-flow graphs in order to accomplish the required computation. In order to support this mechanism, a program must be partitioned into a collection of acyclic data-flow graphs that have been augmented with necessary synchronization and data-transfer mechanisms. The issues associated with the automatic partitioning of programs and the generation of the appropriate data-flow graphs were addressed. A prototype compiler for OCCAM source language programs was built to test and verify the graph generation algorithms developed.

The third phase of this project involved the design of a practical processing element that implements the indexed queue machine execution model. An important goal in the development of this design was to provide an architecture that supports the conventional, Von Neumann execution model as well as the new, queue-based model. This was done in recognition of the fact that there are some forms of computation that execute more efficiently on Von Neumann architectures than they do on data-flow architectures. This processing element is intended to be a component of a multiprocessor system. To facilitate the communication of values (tokens) between processing elements, a message-passing scheme is proposed. This scheme is based on the use of dedicated message-handling hardware. The protocols to accomplish the message passing were developed and studied.

The final phase of the research involved the confirmation of the new concepts introduced in this thesis. First, a specification of the data-path architecture was constructed. This architecture was then emulated using a computer program to verify that the processing element behaved as expected. Second, the message-passing protocols were studied (empirically) by computer simulation to determine that they function correctly. Finally, a computer simulation of the multiprocessor system architecture was constructed. This simulation served a number of purposes. First, it gave strong evidence that the code generated by the compiler was correct. Second, it demonstrated that the dynamic data-flow splicing mechanisms work. Finally, it allowed the evaluation of the performance of the system for a number of benchmark computer programs.

0.9. Summary

In this chapter, the background and motivation for this research project were presented. The main goal is the development of a multiprocessor computer architecture based on the data-flow execution model. Data-flow systems have been suggested as a means to achieve high-speed or high-throughput computation by automatically exploiting the inherent parallelism of programs. A number of problems with the data-flow execution model have prevented the achievement of the expected performance. This thesis addresses some of these problems.

The basic elements of this thesis are i) the development of a pseudo-static data-flow architecture based on an execution mechanism that uses a queue to handle operands; ii) the use of a
dynamic data-flow program graph splicing mechanism to coordinate the parallel evaluation of acyclic data-flow graphs at execution time; iii) the design and implementation of various algorithms for the compilation of OCCAM source language programs into collections of acyclic graphs; iv) the design of a processing element and multiprocessor architecture that supports both the queue-based execution model as well as conventional, Von Neumann execution; and v) the confirmation and performance evaluation of the new concepts introduced in this thesis by software simulation and emulation. The queue-based execution models are described in Chapter 3. The dynamic data-flow graph splicing mechanism and program compilation methods are presented in Chapter 4. The processing element and multiprocessor system architecture are described in Chapter 5. Finally, the simulation and performance results are presented in Chapter 6.

Chapter 3
Queue Machines

3.10. Introduction

In this chapter, the simple queue machine and indexed queue machine execution models will be introduced. These execution models use a queue as the underlying mechanism for the manipulation of operands and results.

The simple queue machine execution model uses a first-in, first-out queue. It will be shown that instruction sequences for such machines are easily generated from expression parse trees. It will also be shown that queue machines have the potential for efficiently exploiting pipelined ALUs.

The indexed queue machine is an extension of the simple queue machine execution model. It will be shown that instruction sequences on indexed queue machines are the natural execution model for computation tasks specified by a certain class of data-flow graphs. This is the main motivation for the development of such machines.

3.11. Simple Queue Machine Execution Model

The simple queue machine execution model uses a first-in, first-out (FIFO) queue data structure as the underlying control mechanism for the manipulation of operands and results. A queue machine is analogous to a stack machine in that it has operations in its instruction set which implicitly reference an operand queue, just as a stack machine has operations which implicitly reference an operand stack. In a stack machine, implicitly referenced operands are retrieved (popped) from the top of an operand stack and results are returned (pushed) back onto the top of the operand stack. In a queue machine, operands are retrieved from the front of the queue of operands and results are returned to the rear of the queue of operands.

For example, consider an add instruction. On a stack machine, the add instruction pops two operands from the top of the operand stack, computes their sum, and pushes the sum back onto the stack. On a queue machine, the add instruction removes two operands from the front of the operand queue, computes their sum, and places the result at the rear of the queue. In the former case, the result of the operation is immediately available at the top of the stack. In the latter case, the result will be behind any other operands in the queue.

As an example to compare the queue-based and stack-based execution models, the instruction sequences required to compute the statement \( f \leftarrow ab + \frac{(c - d)}{e} \) on a stack machine and a queue machine are shown in Table 3.1. Note that the two instruction sequences have the same length and that the same set of instructions are used. Furthermore, the queue machine instruction sequence is simply a permutation of the stack machine instruction sequence. This example also
illustrates how memory operations work on a queue machine. On a queue machine, the fetch operation retrieves an operand from memory and places it at the rear of the operand queue. A store operation takes the first operand from the front of the queue and stores it in memory.

$$f \leftarrow ab + \frac{(c-d)}{e}$$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction sequence</td>
<td>instruction sequence</td>
</tr>
<tr>
<td>stack contents</td>
<td>queue contents</td>
</tr>
<tr>
<td>fetch $a$</td>
<td>fetch $c$</td>
</tr>
<tr>
<td>$a$</td>
<td>$c$</td>
</tr>
<tr>
<td>fetch $b$</td>
<td>fetch $d$</td>
</tr>
<tr>
<td>$b, a$</td>
<td>$c, d$</td>
</tr>
<tr>
<td>mul</td>
<td>fetch $a$</td>
</tr>
<tr>
<td>$ab$</td>
<td>$c, d, a$</td>
</tr>
<tr>
<td>fetch $c$</td>
<td>fetch $b$</td>
</tr>
<tr>
<td>$c, ab$</td>
<td>$c, d, a, b$</td>
</tr>
<tr>
<td>fetch $d$</td>
<td>sub</td>
</tr>
<tr>
<td>$d, c, ab$</td>
<td>$a, b, c-d$</td>
</tr>
<tr>
<td>sub</td>
<td>fetch $e$</td>
</tr>
<tr>
<td>$c-d, ab$</td>
<td>$a, b, c-d, e$</td>
</tr>
<tr>
<td>fetch $e$</td>
<td>mul</td>
</tr>
<tr>
<td>$e, c-d, ab$</td>
<td>$c-d, e, ab$</td>
</tr>
<tr>
<td>div</td>
<td>div</td>
</tr>
<tr>
<td>$\frac{(c-d)}{e}, ab$</td>
<td>$ab, \frac{(c-d)}{e}$</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>$ab + \frac{(c-d)}{e}$</td>
<td>$ab + \frac{(c-d)}{e}$</td>
</tr>
<tr>
<td>store $f$</td>
<td>store $f$</td>
</tr>
</tbody>
</table>

Table 3.1.

Queue machine and Stack machine instruction sequences for evaluating

$$f \leftarrow ab + \frac{(c-d)}{e}.$$ 

In the preceding example, it was shown that both a queue machine and a stack machine can be used to evaluate a particular expression. In the next section, it will be shown that the queue machine can be used to evaluate an arbitrary expression, and an algorithm will be given for generating the instruction sequence required to evaluate a given expression on a queue machine.

3.12. Evaluating Expressions on a Queue Machine

In this section, it will be shown that:

- A queue machine can be used to evaluate an arbitrary expression.
- A queue machine instruction sequence for an arbitrary expression can be efficiently derived from that expression.

To prove the first claim above, an algorithm for generating a queue machine instruction sequence, given the parse tree of an expression, will be described. To prove the second claim above, it will be shown that this algorithm has a time complexity of $O(n)$ and a space requirement of $O(n)$ where $n$ is the number of nodes in the parse tree of the expression. It is assumed that the parse tree for a given expression is readily available. The actual process of constructing parse trees for expressions is well understood and is not the issue here (e.g., see Aho 1977).

It is well known that the stack machine instruction sequence corresponding to a given expression parse tree can be obtained from the parse tree by doing a post-order traversal of it. It
will be shown that the queue machine instruction sequence can be obtained by traversing the
parse tree in a new way. This traversal method will be called a *level-order traversal*. The follow-
ing example will serve to illustrate how a level-order traversal proceeds.

The parse tree corresponding to the statement of Table 3.1 is shown in Figure 3.1.(a). The
leaf nodes of the parse tree correspond to fetch operations and the internal nodes of the parse tree
 correspond to arithmetic operators.

Figure 3.1.
The (binary) expression parse tree (a), level order of nodes (b), and level-order conju-
gate tree (c) for the statement \( f \leftarrow ab + \frac{(c - d)}{e} \).

A level-order traversal of the parse tree of Figure 3.1.(a) is performed by visiting the nodes of
the tree in the order shown in Figure 3.1.(b). Informally, a level-order traversal is done by vis-
iting the nodes of the parse tree from the deepest to the shallowest levels and from left to right
within each level. It will be shown below that a level-order traversal produces a queue machine
instruction sequence.

Unfortunately, a level-order traversal done directly on the parse tree is inefficient and com-
plicated because it involves back-tracking. For example, in Figure 3.1.(b), node 6 has to be
inspected to see if it has any descendents at a deeper level than nodes 1 and 2 before the instruc-
tions corresponding to nodes 1 and 2 can be emitted. However, a level-order traversal can be
accomplished efficiently in two steps. First, a new tree called the *level-order conjugate tree*
is constructed. Then, an *in-order traversal* of the conjugate tree is performed.
The level-order conjugate tree of the parse tree of Figure 3.1.(a) is shown in Figure 3.1.(c). The details of constructing the level-order conjugate tree will be described below. Note that an in-order traversal of the tree of Figure 3.1.(c) produces the queue machine instruction sequence of Table 3.1.

The concepts introduced in the preceding example will now be presented formally. First, the concepts of level-order and level-order traversal will be introduced. Then, it will be shown that a level-order traversal of a (binary) expression parse tree yields a valid queue machine instruction sequence. Finally, the level-order tree traversal algorithm will be presented and its performance will be quantified.

**Definition:** A binary tree $T$ is either the empty set $T = \emptyset$, or an ordered set $T = \{R, T_l, T_r\}$ in which $R$ is a node called the root of $T$, and $T_l$ (resp. $T_r$) is a binary tree called the left (resp. right) subtree of $T$.

**Definition:** $N(T)$ is the set of nodes of a binary tree $T$, i.e.
\[
N(T) = \emptyset \iff T = \emptyset,
N(T) = \{R\} \cup N(T_l) \cup N(T_r) \iff T \neq \emptyset.
\]

**Definition:** $S(T)$ is the set of all subtrees of a binary tree $T$, i.e.
\[
S(T) = \emptyset \iff T = \emptyset,
S(T) = \{T\} \cup S(T_l) \cup S(T_r) \iff T \neq \emptyset.
\]

**Definition:** $\Gamma_T(n)$ is the level of node $n \in N(T)$ in tree $T = \{R, T_l, T_r\}$, i.e.
\[
\Gamma_T(R) = 0,
\forall m \in N(T_l): \Gamma_T(m) = \Gamma_T(m) + 1,
\forall m \in N(T_r): \Gamma_T(m) = \Gamma_T(m) + 1.
\]

**Definition:** $L_T(n)$ (resp. $R_T(n)$) is the set of nodes in the left (resp. right) subtree of tree $T'$, the subtree of $T$ having as its root the node $n$, i.e.
\[
L_T(n) = N(T'): T' = \{n, T'_l, T'_r\} \in S(T),
R_T(n) = N(T'): T' = \{n, T'_l, T'_r\} \in S(T).
\]

**Definition:** $D_T(n)$ is the set of descendents of node $n$ in tree $T$, i.e.
\[
D_T(n) = L_T(n) \cup R_T(n).
\]

**Definition:** (level order precedence) $\pi_T$ is a relation defined on $N(T)$ such that
\[
\forall a, b \in N(T): a \pi_T b \iff (a = b) \lor (a \neq b \land \Gamma_T(a) > \Gamma_T(b)) \lor (a \neq b \land L_T(a) = L_T(b) \land (\exists c \in N(T): a \in L_T(c) \land b \in R_T(c)).
\]

The relation $\pi_T$ will be called the level order relation on the nodes of a binary tree $T$. Figure 3.2 shows a trivial binary tree. For this tree, $a \pi_T a$ since $a = a$; $a \pi_T c$ since $a \neq c$ and $\Gamma_T(a) > \Gamma_T(c)$; and $a \pi_T b$ since $a \neq b$ and $\Gamma_T(a) = \Gamma_T(b)$ and $a \in L_T(c)$ and $b \in R_T(c)$.

**Lemma:** $\pi_T$ is a partial order relation on the set of nodes $N(T)$ of a binary tree $T$.

**Proof:** It must be shown that $\pi_T$ is 1) reflexive $(x \pi_T y \land y \pi_T z \implies x = y)$, and 3) transitive $(x \pi_T y \land y \pi_T z \implies x \pi_T z)$.

1) (reflexive) $\forall x \in N(T): x \pi_T x$ by definition.

2) (antisymmetric) Proof by contradiction: Given $x \pi_T y \land y \pi_T x$. Assume $x \neq y$. By definition, $\Gamma_T(x) \neq \Gamma_T(y) \implies \Gamma_T(x) > \Gamma_T(y) \lor \Gamma_T(x) < \Gamma_T(y)$ (contradiction). Therefore,
3. Given $k_i$ and $k_j$.

Assume $x \neq y$. Clearly $x = y \lor y = z \implies x \pi_T z$.

Now assume $x \neq y \neq z \neq x$. There are three possibilities:

i) $\Gamma_T(x) > \Gamma_T(y) \land \Gamma_T(y) \geq \Gamma_T(z)$.
   \[ \therefore \quad x \pi_T z \]

ii) $\Gamma_T(x) \geq \Gamma_T(j) \land \Gamma_T(y) > \Gamma_T(z)$.
   \[ \therefore \quad x \pi_T z \]

iii) $\Gamma_T(x) = \Gamma_T(y) \land \Gamma_T(y) = \Gamma_T(z)$.

Then $\exists k_1: x \in L_T(k) \land y \in R_T(k)$ and $\exists k_2: y \in L_T(k') \land z \in R_T(k')$. Again, there are six possibilities for the relative positions of $k$ and $k'$ in $T$.

a) $k = k'$.
   \[ \therefore \quad x \in L_T(k) \land z \in R_T(k). \therefore \quad x \pi_T z \]

b) $k \neq k' \land k \in L_T(k')$.
   \[ \therefore \quad x \in L_T(k') \land z \in R_T(k'). \therefore \quad x \pi_T z \]

c) $k \neq k' \land k \in R_T(k')$.
   This is not possible because it implies $y \in L_T(k') \land y \in R_T(k')$.

---

Figure 3.2.

Trivial binary tree illustrating \( a \pi_T a, a \pi_T c, a \pi_T b \).
Since nodes traversal of binary trees. The next step is to consider binary trees as parse trees for expressions.

The relation \( \pi_T \) is a binary tree, \( (\vdash n_1, \ldots, n_k) \) is a chain. Therefore, there exists a unique mapping from the nodes \( n \in N(T) \) to the integers \( 1 \cdots |N(T)| \) due to the isomorphism of chains.

**Definition:** \( \Pi(T) \) is the level-order traversal of a binary tree \( T \). It is the following sequence of the nodes of \( T \):

\[
\Pi(T) = \{n_1, \ldots, n_{|N(T)|}; \forall i, j, 1 \leq i < j \leq |N(T)|; n_i \pi_T n_j \}. 
\]

The preceding analysis has formally presented the notions of level-order, and level-order traversal of binary trees. The next step is to consider binary trees as parse trees for expressions.

**Definition:** \( O_0 \) (resp. \( O_1, O_2 \)) is the set of nullary (resp. unary, binary) operators.

For any operator \( n \), \( A(n) \) is the **arity** of the operator, i.e.

\( A(n) = k \iff n \in O_k \).

**Definition:** A (binary) expression **parse tree** \( P \) is either the empty set \( P = \emptyset \) or the ordered set \( P = \{n, P_1, P_2\} \) where \( n \) is an operator and \( P_1 \) and \( P_2 \) satisfy:

\[
\begin{align*}
(n \in O_0 \land P_1 = \emptyset \land P_2 = \emptyset) \lor \\
(n \in O_1 \land P_1 \neq \emptyset \land P_2 = \emptyset) \lor \\
(n \in O_2 \land P_1 \neq \emptyset \land P_2 \neq \emptyset).
\end{align*}
\]

**Definition:** The evaluation \( E(I) \) of a sequence of operators \( I = \{n_1, \ldots, n_k\} \) on a queue machine is the sequence of states \( \{S_1, \ldots, S_f\} \) in which each state \( S_i = (I_i, Q_i) \) is an ordered pair consisting of \( I_i = \{n_1, \ldots, n_k\} \) which is the remainder of the input sequence yet to be evaluated and \( Q_i = \{m_1, \ldots, m_i\} \) which is the contents of the queue. The initial state is \( S_1 = (I, \emptyset) \). The method of determining state \( S_{i+1} \) given state \( S_i \) is:

\[
\begin{align*}
i) \quad & S_i = (\{n_1, \ldots, n_k\}, \{m_1, \ldots, m_i\}) \land n_i \in O_0 \\
& \implies S_{i+1} = (\{n_{i+1}, \ldots, n_k\}, \{m_1, \ldots, m_i, n_i\}) \\
\end{align*}
\]

\[
\begin{align*}
ii) \quad & S_i = (\{n_1, \ldots, n_k\}, \{m_1, m_2, \ldots, m_i\}) \land n_i \in O_1 \\
& \implies S_{i+1} = (\{n_{i+1}, \ldots, n_k\}, \{m_2, \ldots, m_i, n_i\}) \\
\end{align*}
\]
iii) \( S_i = (\{n_1, \cdots, n_k\}, \{m_1, m_2, m_3, \cdots, m_l\}) \land n_i \in O_2 \)
\[ \implies S_{i+1} = (\{n_{i+1}, \cdots, n_k\}, \{m_1, \cdots, m_l, n_i\}) \]

**Definition:** \( \lambda_T(n) \) (resp. \( \rho_T(n) \)) is the left (resp. right) immediate descendant of node \( n \) in tree \( T \).

**Lemma:** Consider a state \( S_i \) in the evaluation of the level-order traversal of a (binary) expression parse tree \( P \), i.e. \( S_i = (I_i, Q_i) \in E(\Pi(P)) \). If \( I_i = \{n_1, \cdots, n_{|N(P)|}\} \) satisfies the condition \( \forall n' \in N(P) - I_i \colon \Gamma_p(n_i) > \Gamma_p(n') \) then \( Q_i = \{m_1, \cdots, m_l\} \) satisfies the condition \( \forall m \in Q_i \colon \Gamma_p(m) = \Gamma_p(n_i) + 1 \) and \( \forall m' \in N(P) - Q_i \colon \Gamma_p(m_1) \neq \Gamma_p(m') \).

This lemma states that for every state in the evaluation of parse tree \( P \) in which the next operation is the first in its level of \( P \), the queue contains only the results of the operations on the preceding level of the tree. In other words, at the start of evaluation of a given level, all the operands needed for that level are available in the queue.

**Proof:** (by induction)
1) (base case) Clearly, \( S_1 = (\{n_1, \cdots, n_{|N(P)|}\}, \emptyset) \) satisfies the conditions of the lemma.
2) (induction) Assume the conditions of the lemma are satisfied by some state \( S_i = (\{n_1, \cdots, n_{|N(P)|}\}, \{m_1, \cdots, m_l\}) \). Now, choose \( k \) such that \( \Gamma_p(n_i) = \Gamma_p(n_{i+j}) \), \( j = 0, \cdots, k-1 \). Clearly, \( \Gamma_p(n_{i+k}) = \Gamma_p(n_i) + 1 \). Thus, it must be shown that the state \( S_{i+k} \) satisfies the conditions of the lemma. In other words, the operations \( n_i, \cdots, n_{i+k-1} \) must consume the operands \( m_1, \cdots, m_l \). Let \( s = \sum_{j=0}^{k-1} A(n_{i+j}) \).

Assume \( s > l \). Then \( \exists n' \in \{n_i, \cdots, n_{i+k-1}\} \) such that \( \lambda(n') \notin Q_i \) (contradiction).
Assume \( s < l \). Then \( \exists n' \in N(P) - \{n_i, \cdots, n_{i+k-1}\} \) such that \( \lambda(p(n')) \in Q_i \) (contradiction).
Therefore \( s = l \). Thus \( S_{i+k} = (\{n_{i+k}, \cdots, n_{|N(P)|}\}, \{n_i, \cdots, n_{i+k-1}\}) \), which satisfies the conditions of the lemma. □

**Corollary 1:** Given a (binary) expression parse tree \( P \),
\[ \forall S_i = (\{n_1, \cdots, n_{|N(P)|}\}, \{m_1, \cdots, m_l\}) \in E(\Pi(P)) \]
\[ (n_i \in O_1 \implies m_1 = \lambda_p(n_i)) \land \]
\[ (n_i \in O_2 \implies m_1 = \lambda_p(n_i) \land m_2 = \rho_p(n_i)) \].

This corollary states that not only are there enough operands at every state of the evaluation of the level-order traversal of a binary tree, but that in fact the operands are the correct ones.

**Proof:** Consider a state \( S_i = (I_i, Q_i) \) where \( I_i = \{n_1, \cdots, n_{|N(P)|}\} \) such that \( \forall n' \in N(P) - I_i \colon \Gamma_p(n') > \Gamma_p(n_i) \). Choose \( k \) such that \( \forall j \in \{0, \cdots, k-1\} : \Gamma_p(n_{i+j}) = \Gamma_p(n_{i+k}) \) From the lemma, \( Q_i \) contains all the operands needed for the next \( k \) operations. The rules for evaluation of queue machine instruction sequences guarantee that \( n_i \pi p_n i_{i+1} \pi p \cdots \pi p n_{i+k-1} \) and \( m_1 \pi p m_2 \pi p \cdots \pi p m_l \). Thus, the operators and operands are matched. □

**Corollary 2:** The final state \( S_f \) of the evaluation of (binary) expression parse tree \( P = \{r, P_1, P_2, P_3\} \) is \( S_f = (\emptyset, \{r\}) \).

**Proof:** Consider state \( S_i = (\{n_{|N(P)|}\}, Q_i) \). Clearly, the lemma applies to this state since \( n_{|N(P)|} \) is the first and only node at level 0 of \( P \) since it is the root of \( P \). Thus, \( A(n_{|N(P)|}) = |Q_i| \), and the next state is \( S_f = (\emptyset, \{n_{|N(P)|}\}) \). □

The preceding lemma and its two corollaries prove the first major result of this section, namely that the queue machine evaluation of a level-order traversal of a (binary) expression parse tree correctly computes the expression. At each step, the correct operands are available and the
execution terminates with the result as the single element of the queue. It remains now to show an efficient way to do the level-order traversal.

**Definition:** A right-only binary tree $T$ is either the empty set $T = \emptyset$ or an ordered set $T = (\{R, \emptyset, T_r\},$ where $R$ is a node called the root of $T$ and $T_r$ is a right-only binary tree called the right subtree of $T$. Clearly, right-only binary trees are binary trees.

**Definition:** A tree of right-only binary trees $T$ is either the empty set $T = \emptyset$ or an ordered set $T = (\{R, T_l, T_r\},$ where $R$ is a node called the root of $T$, $T_l$ is a tree of right-only binary trees, and $T_r$ is a right-only binary tree. Clearly, a tree of right-only binary trees is a binary tree.

**Definition:** The level-order conjugate $\delta(T)$ of a binary tree $T$ is a tree of right-only binary trees with the following properties:

i) $N(\delta(T)) = N(T)$,

ii) $\forall n \in N(\delta(T)), \forall k \in R_{\delta(T)}(n): \Gamma_{\delta(T)}(n) = \Gamma_T(k) \land n \pi_T k$,

iii) $\forall n \in N(\delta(T)), \forall k \in L_{\delta(T)}(n): \Gamma_{\delta(T)}(n) < \Gamma_T(k)$.

**Definition:** The in-order traversal $\iota(T)$ of a binary tree $T = \emptyset$ is the empty sequence $\iota(T) = \emptyset$, and the in-order traversal $\iota(T)$ of a binary tree $T = \{n, T_l, T_r\}$ is the sequence of its nodes $\iota(T) = \{n_1, \ldots, n_{|N(T)|}; n_i \in N(T)\}$ constructed by appending in order:

i) $\iota(T_l)$,

ii) $\{n\}$,

iii) $\iota(T_r)$.

**Lemma:** For all binary trees $T$, $\iota(\delta(T)) = \Pi(T)$.

**Proof:** If $T = \emptyset$, then clearly the lemma is true. Consider $\delta(T) = \{n, \delta_R, \delta_l\}$. An in-order traversal of $\delta(T)$ consists of three sequences $s_1 = \iota(\delta_l), s_2 = \{n\}, s_3 = \iota(\delta_R)$. By the definition of $\delta(T)$, $\forall n' \in \delta_l: \Gamma_T(n') > \Gamma_T(n)$. Therefore, $n' \pi_T n$.

By the definition of $\delta(T)$, $\delta_l$ is a right-only tree. In-order traversal of a right-only tree constructs a sequence which is a concatenation of the root of the tree and an in-order traversal of its right subtree. Therefore, by the definition of $\delta(T)$, $\delta_l = \{m_1, \ldots, m_l\}$ such that $\forall m \in \iota(\delta_l): \Gamma_T(m) = \Gamma_T(n)$ and $n \pi_T m_1 \pi_T m_2 \pi_T \ldots \pi_T m_l$.

Hence, $\iota(\delta(T)) = \{n_1, \ldots, n_{|N(T)|}; \forall i, j, 1 \leq i < j \leq |N(T)|: n_i \pi_T n_j\}$. Therefore, $\iota(\delta(T)) = \Pi(T)$.

The preceding lemma shows that a level-order traversal of a binary tree can be constructed by doing an in-order traversal of the level-order conjugate of that tree. Since an (recursive) in-order traversal of a tree $T$ has a time complexity of $O(|N(T)|)$ and a (worst-case) space requirement of $O(|N(T)|)$, a level-order traversal is efficiently generated from the level-order conjugate tree.

An algorithm for constructing the level-order conjugate tree of a given binary tree is shown in Figure 3.3. This algorithm is presented in the Turing programming language [Holt 1984]. The algorithm does a reverse post-order traversal of the binary tree (i.e. visit the root, traverse the right subtree, traverse the left subtree), constructing the level-order conjugate tree as it goes.
var node : collection of forward nodeType

type nodeType :
  record
    contents : TypeOfParseTreeNodeContents
    left, right : pointer to node
  end record

procedure BuildConjugate (conjugateTree, parseTree : pointer to node)
  % This procedure builds a level-order conjugate of a binary parse tree.

  import (var node)
  var tmp : pointer to node

  if parseTree not = nil (node) then
    if node (conjugateTree) .left = nil (node) then
      new node, tmp
      node (tmp) .left := nil (node)
      node (tmp) .right := nil (node)
      node (tmp) .contents := node (parseTree) .contents
      node (conjugateTree) .left := tmp
    else
      new node, tmp
      node (tmp) .left := nil (node)
      node (tmp) .right := node (node (conjugateTree) .left) .right
      node (tmp) .contents := node (node (conjugateTree) .left) .contents
      node (node (conjugateTree) .left) .right := tmp
      node (node (conjugateTree) .left) .contents :=
        node (parseTree) .contents
    end if
    BuildConjugate (node (parseTree) .right, node (conjugateTree) .left)
    BuildConjugate (node (parseTree) .left, node (conjugateTree) .left)
  end if
end BuildConjugate

Figure 3.3.(a)

Turing algorithm for computing the level-order conjugate tree of a binary tree (continued on next page).
procedure InOrderTraverse (ptr : pointer to node)
% This procedure does an in-order traversal of a binary parse
% tree, emitting instructions as it goes.

import (node)

if ptr not = nil (node) then
    InOrderTraverse (node (ptr) .left)
% Emit the instruction corresponding to node (ptr) .contents
    InOrderTraverse (node (ptr) .right)
end if
end InOrderTraverse

procedure ConstructSequence (parseTree : pointer to node)
% This procedure constructs a Queue Machine instruction sequence
% for a given parseTree by first building the level-order
% conjugate tree by calling BuildConjugate and then
% emitting the instruction sequence by calling InOrderTraverse.

import (var node, BuildConjugate, InOrderTraverse)
var conjugateTree : pointer to node

new node, conjugateTree
node (conjugateTree) .left := nil (node)
node (conjugateTree) .right := nil (node)

BuildConjugate (conjugateTree, parseTree)
InOrderTraverse (node (conjugateTree) .left)
end ConstructSequence

Figure 3.3.(b)
Turing algorithm for computing the level-order conjugate tree of a binary tree.

Lemma: The algorithm of Figure 3.3 correctly constructs the level-order conjugate
of a binary tree.

Proof: The procedure BuildConjugate constructs an augmented level-order conjugate tree
\( \delta' = \{s, \delta(T), \emptyset\} \) where \( s \notin N(T) \) is a sentinel node. For convenience, \( s \) is defined such that
\( \forall n \in N(T): n \pi_T s \) and \( \Gamma_T(s) = -1 \).
The BuildConjugate procedure has two arguments, a pointer to a node \( q \in N(\delta(T)) \) in the conjugate
tree and a pointer to a node \( p \in N(T) \) in the original (parse) tree. It will be shown that the
BuildConjugate procedure satisfies the following invariants:

i) \( \delta'(T) \) is a tree of right-only binary trees,

ii) \( \forall n \in \delta'(T), \forall k \in L_{\delta'(T)}(n): \Gamma_T(k) > \Gamma_T(n), \)

iii) \( \forall n \in \delta'(T), \forall k \in R_{\delta'(T)}(n): \Gamma_T(k) = \Gamma_T(n) \land n \pi_T k. \)
iv) \( \Gamma_T(p) = \Gamma_T(q) + 1 \).

At each step, the \textit{BuildConjugate} procedure adds a new node to the partially constructed level-order conjugate tree. There are two ways that it can do this:

1) If \( \lambda_{\delta'(i)}(q) = \emptyset \), then create a new node \( \lambda_{\delta'(i)}(q) = \{ p, \emptyset, \emptyset \} \).
   - Condition i) is satisfied by the new tree since the resulting tree is obviously a tree of right-only binary trees.
   - Condition ii) is satisfied by the new tree since \( \Gamma_T(p) = \Gamma_T(q) + 1 \).
   - Condition iii) is satisfied trivially, since no right subtrees of \( \delta'(T) \) were affected.

2) \( \lambda_{\delta'(T)}(q) \neq \emptyset \). Let \( \lambda_{\delta'(T)}(q) = \{ n, T_l, T_r \} \). Then create a new node \( \lambda_{\delta'(T)}(q) = \{ p, T_l, \{ n, \emptyset, T_r \} \} \).
   - Condition i) is satisfied by the new tree since the resulting tree is obviously a tree of right-only binary trees.
   - Condition ii) is satisfied by the new tree since \( \Gamma_T(p) = \Gamma_T(q) + 1 \).
   - Since the \textit{BuildConjugate} procedure traverses the tree \( T \) in reverse post-order, \( \rho_T \). Thus, condition iii) is satisfied by the new tree.

After completing one of the above steps, the \textit{BuildConjugate} procedure calls itself recursively, first setting \( p \leftarrow \rho_T(p) \) and \( q \leftarrow \lambda_{\delta'(T)}(q) \); and then setting \( p \leftarrow \lambda_T(p) \) and \( q \leftarrow \lambda_{\delta'(T)}(q) \). Clearly, both cases satisfy the condition iv).

The algorithm starts with \( \delta'(T) = \{ s, \emptyset, \emptyset \} \), \( p \) is initially the root of \( T \), and \( q \) is initially \( s \). Thus, the initial condition satisfies the invariants.

The algorithm terminates when all the nodes of \( T \) have been visited. Hence, the algorithm correctly computes the level-order conjugate of a binary tree. \( \square \)

The preceding lemma shows that the algorithm of Figure 3.3 correctly computes the level-order conjugate of a binary tree. Since the \textit{BuildConjugate} procedure does a (recursive) reverse post-order traversal of a binary tree \( T \), its time complexity is \( O(|N(T)|) \) and since it constructs a conjugate tree as it goes, it has a space requirement of \( O(|N(T)|) \). Thus, it is possible to construct efficiently a queue machine instruction sequence from the parse tree of an expression. This procedure was coded and executed to obtain the quantitative results presented in the following section.

### 3.13. On the Pipelined Execution of Queue-Based Instruction Sequences

In this section, it will be shown that the queue-based execution mechanism implemented in a pipelined arithmetic/logic unit (ALU) is more efficient than the stack-based execution model. Figure 3.4 shows the flow of data between the ALU, the memory, and the queue or stack. Observe that the queue-based execution model has the potential to take advantage of a pipelined ALU when the queue of operands is not empty. On the other hand, the stack-based execution model cannot exploit a pipelined ALU, since the results of one operation must be returned to the top of the stack before they can become the operands of the next operation.

To quantify the benefit to performance of the queue-based execution model over the stack-based execution model on a pipelined ALU, a computation was done to calculate the average speed-up possible when evaluating a (binary) expression parse tree with a given number of nodes. The speed-up was defined as the ratio of the number of cycles taken to evaluate an expression on a stack machine with an n-stage pipelined ALU, to the number of cycles taken to evaluate the same expression on a queue machine with an n-stage pipelined ALU. Two cases were considered:

**Case 1:** Both processors emit instructions at a rate of at most one instruction per cycle. An instruction cannot be emitted until its operands are available at the top of the stack or the front of the queue. A fetch operation cannot be issued until the ALU
Figure 3.4.

Flow of operands and results in queue-based execution model (a) and stack-based execution model (b).

is idle. (non-overlapped fetch/execute).

**Case 2:** Both processors emit instructions at a rate of at most one instruction per cycle. An instruction cannot be emitted until its operands are available at the top of the stack or the front of the queue. A fetch operation can be issued immediately and takes one cycle. (overlapped fetch/execute).

Rather than comparing the execution times of an arbitrary benchmark program, the average execution time required to evaluate all possible (binary) expression parse trees (as defined in the previous section) was calculated as a function of the number of nodes in the parse tree. Thus, the number of nodes in the parse tree was used as a measure of the complexity of the expression being evaluated.

A computer program was written which enumerates all the (binary) expression parse trees with a given number of nodes and determines the number of cycles required to evaluate the expression on both queue- and stack-based machines with n-stage pipelined ALUs. The enumeration procedure was adapted from [Solomon 1980]. For example, Figure 3.5 shows the four (binary) expression parse trees with exactly four nodes. The caption of Figure 3.5 lists four expressions which would generate the parse trees shown.

The results obtained for a two-stage pipelined ALU are shown in Table 3.2. Note that the queue-based execution model always meets or exceeds the performance of the stack-based machine. This is true for all instruction sequences (not just the average). For example, with 11 nodes in a parse tree, the execution time is almost 10% less than that of the stack machine. Note also that case 2 performs better than case 1. This is as expected, since the queue machine can take better advantage of overlapped fetch and execution.

In Table 3.3, the effect of varying the number of ALU pipeline stages is shown for a fixed number of nodes in the (binary) expression parse tree. This table shows the interesting result that under case 1, the benefit of the queue machine over the stack machine increases with longer ALU pipelines. On the other hand under case 2, the benefit of the queue machine is maximum for a
Figure 3.5.
The four binary expression parse trees with exactly four nodes: $-(-x)$, (a); $-x \times y$, (b); $-x \times y$, (c); $x \times (-y)$, (d).

<table>
<thead>
<tr>
<th>nodes in parse tree</th>
<th>number of trees</th>
<th>case 1</th>
<th>case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>1.02</td>
<td>1.02</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>7</td>
<td>45</td>
<td>1.04</td>
<td>1.05</td>
</tr>
<tr>
<td>8</td>
<td>101</td>
<td>1.05</td>
<td>1.07</td>
</tr>
<tr>
<td>9</td>
<td>227</td>
<td>1.05</td>
<td>1.08</td>
</tr>
<tr>
<td>10</td>
<td>510</td>
<td>1.05</td>
<td>1.09</td>
</tr>
<tr>
<td>11</td>
<td>1146</td>
<td>1.06</td>
<td>1.10</td>
</tr>
</tbody>
</table>

| Speed-up$^\dagger$ = $\frac{StackMachineCycles}{QueueMachineCycles}$ |

Table 3.2.

Speed-up figures as a function of the number of nodes in the parse tree for a two-stage pipelined ALU.

two-stage pipelined ALU. This decreasing benefit is the result of the unrealistic nature of case 2 with respect to stack machines. By allowing the immediate issuing of fetch operations, the stack machine is allowed to do its pushes and pops out of order, which violates the basic stack execution mechanism.

The focus of this section has been a comparison of stack- vs. queue-based execution mechanisms. Of course, an optimally assigned register machine can take advantage of a pipelined ALU at least to the same extent that a queue machine can (up to the limits imposed by the number of CPU registers).

The indexed queue machine execution model is an extension of the simple queue machine execution model. It uses an indexed queue as the underlying control mechanism for the manipulation of operands and results [Preiss 1985]. The result of each operation is assigned an index. The index specifies an offset from the front of the operand queue. The result is placed in the operand queue at the position specified by its index. In effect, the result of an operation can be placed at any position in the operand queue. However, operands are still only retrieved from the front of the operand queue.

When constructing instruction sequences for an indexed queue machine, special care must be taken to ensure that the result of one operation does not overwrite the result of some other operation. Furthermore, the front of the queue must always contain valid operands — there must not be a hole in the queue.

As an example to illustrate the indexed queue machine execution model, the instruction sequence required to compute the statement $d ← \frac{a}{a+b} + (a+b)c$ on an indexed queue machine is shown in Table 3.4. Note that the operations may have more than one result index. This simply means that the result is duplicated once for each index value. Note also that the result index is specified as an offset from the front of the operand queue after the operands for the current instruction have been removed.

Figure 3.6.(a) shows the parse tree for the expression of Table 3.4. The instruction sequence needed to evaluate this expression on a simple queue machine requires 11 operations — one for each node of the parse tree. By combining common subtrees, the programmer can transform the parse tree of Figure 3.6.(a) into the data-flow graph of Figure 3.6.(b). The resulting graph cannot be directly evaluated on a simple queue machine. However, it can be evaluated on an indexed queue machine. The data-flow graph contains 7 nodes, each one corresponding to an operation in the indexed queue machine instruction sequence. In the next section, it will be shown that an indexed queue machine is the natural model for evaluating computation tasks expressed as data-flow graphs. In effect, data-flow graphs can be considered as generators of valid (indexed) queue machine instruction sequences.
Table 3.4.
Indexed queue machine instruction sequence for evaluating $d \leftarrow \frac{a}{a+b} + (a+b)c$.

3.15. Executing Data-Flow Graphs on a Queue Machine

In this section, it will be shown that (acyclic) data-flow graphs can be used as generators of valid instruction sequences for (indexed) queue machines. An acyclic data-flow graph is a directed acyclic graph (DAG) in which the nodes represent operators and the arcs represent the flow of operands. It will be shown that such graphs induce a partial-order relation on the set of operators and that all instruction sequences satisfying the partial-order will be correctly evaluated by a (indexed) queue machine.

Definition: A directed graph $G = (V, E)$ consists of a set of vertices $V = \{n_1, \cdots, n_k\}$ and a set $E$ of ordered pairs of vertices called edges, $E = \{(v, w) : v, w \in V\}$.

Definition: A path $p$ in a graph $G = (V, E)$ is a sequence of vertices $\{v_1, \cdots, v_l\} : v_i \in V$ such that $\{(v_1, v_2), (v_2, v_3), \cdots, (v_{l-1}, v_l)\} \subseteq E$.

Let $P(G)$ be the set of all paths in graph $G$.

Definition: A cycle is a path $p = \{v_1, \cdots, v_l\}$ such that $l \geq 2$ and $v_1 = v_l$.

Definition: A directed acyclic graph (DAG) $G$ is a directed graph with no cycles.

Definition: An (acyclic, binary) data-flow graph $G = (V, E)$ is a DAG with the following properties:

i) its vertices, called operators, are partitioned into three sets, $O_0$, $O_1$, and $O_2$, called nullary, unary, and binary operators, respectively; i.e., $\forall n \in V: n \in O_0 \lor n \in O_1 \lor n \in O_2$;
Figure 3.6.

Binary expression parse tree (a) and binary acyclic data-flow graph (b) for the statement $d \leftarrow \frac{a}{a+b} + (a+b)c$.

ii) its edges, $E$, are expanded to become ordered triplets $(v, w, l)$ such that
   $v, w \in V$, and
   $w \in O_1 \Rightarrow l = 0$,
   $w \in O_2 \Rightarrow l \in \{0, 1\}$;

iii) $\forall n \in V$:
    $n \in O_0 \Rightarrow \forall (v, w, l) \in E: v \neq n$,
    $n \in O_1 \Rightarrow \exists$ unique $(v, w, l) \in E: w = n \land l = 0$,
    $n \in O_2 \Rightarrow \exists$ unique $(v, w, l) \in E: w = n \land l = 0$, and
    $\exists$ unique $(v, w, l) \in E: w = n \land l = 1$.

The preceding definition introduces the notion of acyclic binary data-flow graph. A data-flow graph is a DAG with the restriction that its nodes which are nullary operators have no input arcs, its nodes which are unary operators have exactly one input arc, and its nodes which are binary operators have exactly two input arcs labelled 0 and 1. The fact that this analysis considers only operators with two or fewer operands is not a restriction, as the results can be generalized for operators with any fixed maximum number of operands.

**Definition:** $\pi_G$ is a relation defined on the set of the vertices $V$ of a (binary, acyclic) data-flow graph $G = (V, E)$ given by:

$v \pi_G w \iff (v = w) \lor (v \neq w \land \exists p \in P(G): p = \{v, \ldots, w\}$.

**Lemma:** $\pi_G$ is a partial order relation on the set of vertices $V$ of an (acyclic, binary) data-flow graph $G$. 
Proof: It will be shown that $\pi_G$ is reflexive, antisymmetric, and transitive.

1) (reflexive) By definition, $\forall v \in V: v \pi_G v$.

2) (antisymmetric) Consider $v, w \in V: v \pi_G w \land w \pi_G v$. Assume $v \neq w$. Then $\exists p, q \in P(G): p = \{v, \ldots, w\} \land q = \{w, \ldots, v\}$. Thus, $\exists p' \in P(G): p' = \{v, \ldots, w, \ldots, v\}$ a circular path (contradiction).

Therefore, $v \pi_G w \land w \pi_G v \Rightarrow v = w$.

3) (transitive) Consider $v, w, x \in V: v \pi_G w \land w \pi_G x$. Then, $\exists p, q \in P(G): p = \{v, \ldots, w\} \land q = \{w, \ldots, v\}$. Thus, $\exists p' \in P(G): p' = \{v, \ldots, w, \ldots, x\}$. Therefore, $v \pi_G x$. \(\square\)

The preceding lemma proves that (acyclic, binary) data-flow graphs induce a partial order on the operations in the graph. However, the relation $\pi_G$ is not a total order, since it is possible to construct a graph in which there is no path between two of its nodes. It will now be shown that the relation $\pi_G$ can be used to generate a valid indexed queue machine instruction sequence.

**Definition:** An instruction sequence for an indexed queue machine is a sequence $I = \{n_1, \ldots, n_k\}$ of pairs $n_i = (v_i, P_i)$, where $v_i$ is an operator and $P_i$ is a (possibly empty) set of non-negative integers representing the indices of the result.

**Definition:** The evaluation of an indexed queue machine instruction sequence $I = \{n_1, \ldots, n_k\}$ is the sequence of states $E = \{S_1, \ldots, S_f\}$ where $S_i = (I_i, Q_i, r_i)$. $I_i = \{n_1, \ldots, n_k\}$ represents the portion of the instruction sequence yet to be evaluated, $Q_i = \{q_0, \ldots, q_l\}$ represents the contents of the operand queue, and $r_i$ is the index of the front of the operand queue. The initial state is $S_1 = (I, \{\varepsilon, \varepsilon, \ldots, \varepsilon\}, 0)$.

Given state $S_i = (I_i, Q_i, r_i)$ where $I_i = \{n_1, \ldots, n_k\}$, $Q_i = \{q_0, \ldots, q_l\}$, and $n_i = (v_i, \{p_1, \ldots, p_m\})$. The next state $S_{i+1}$ is computed as follows:

i) $I_{i+1} = \{n_1, \ldots, n_k\}$,

ii) $r_{i+1} = r_i + A(v_i)$,

iii) $Q_{i+1}$ is obtained from $Q_i$ by replacing all of $q_{p_1}, q_{p_2}, \ldots, q_{p_m}$ with $v_i$.

**Definition:** Given an (acyclic, binary) data-flow graph $G$, a valid indexed queue machine instruction sequence is a sequence constructed as follows:

1) Choose a sequence $\{v_1, \ldots, v_{|V|}\}$ of the nodes $v_j \in V$ such that $\forall v_j, v_j \in V, i < j: \neg(v_j \pi_G v_i)$. Such a sequence can always be constructed since $\pi_G$ is a partial order.

2) Compute $\{o_1, \ldots, o_{|V|}\}$ such that $o_j = \sum_{j=1}^{i-1} A(v_j)$.

3) Form the result index sets $\{P_1, \ldots, P_{|V|}\}$ such that $\forall (v_j, v_j, 1) \in E: o_j + 1 \in P_j$.

4) Finally, form the instruction sequence $\{n_1, \ldots, n_{|V|}\}$ such that $n_1 = (v_1, P_1)$.

**Lemma:** For every state $S_i$ in the evaluation of a valid indexed queue machine instruction sequence $I$ for a given (acyclic, binary) data-flow graph $G = (V, E)$ constructed as above, where $S_1 = \{I, Q_1, r_i\}$, $I_1 = \{n_1, \ldots, n_{|V|}\}$, $n_i = (v_i, \{p_1, \ldots, p_m\})$, and $Q_i = \{q_0, \ldots, q_l\}$, the following conditions are true:

i) $v_j \in O_1 \Rightarrow q_j = w$ such that $(w, v_j, 0) \in E$.

ii) $v_j \in O_2 \Rightarrow q_{n+1} = q_j$, $q_{n+1} = x$ such that $(w, v_j, 0) \in E$ and $(x, v_i, 1) \in E$. 

- xlix -
Proof: (by course-of-values induction)

1) (base case) Consider \( S_1 = (\{n_1, \ldots, n_{|V|}\}, \{\epsilon, \epsilon, \ldots, \epsilon\}, 0) \), where \( n_1 = (v_1, P_1) \). If \( v_1 \not\in O_0 \) then by the definition of (acyclic, binary) data-flow graphs, \( \exists v_j \in V: (v_j, v_1, 0) \in E \), i.e. \( v_j \pi_G v_1 \). However, the instruction sequence was constructed such that \( \forall v_j \in V, j \neq 1: \neg(v_j \pi_G v_1) \) (contradiction). Thus, \( v_1 \in O_0 \). Thus, \( S_1 \) satisfies the conditions of the lemma.

2) (induction) Assume states \( S_1, \ldots, S_{i-1} \) satisfy the lemma. Consider \( S_i = (I_i, Q_i, r_i) \), where \( I_i = \{n_i, \ldots, n_{|V|}\}, n_i = (v_i, P_i) \), and \( Q_i = \{q_0, \ldots, q_l\} \). There are three possibilities:
   
   i) \( v_i \in O_0 \).
   Then, the state \( S_i \) satisfies the conditions of the lemma (trivial).

   ii) \( v_i \in O_1 \).
   Then by the definition of (acyclic, binary) data-flow graphs, \( \exists \) unique \( v_j \in V: (v_j, v_i, 0) \in E \). Hence, \( v_j \pi_G v_i \).
   The instruction sequence was constructed such that \( v_j \in\{v_1, \ldots, v_{i-1}\} \). Thus, in some preceding state, \( v_j \) was stored in \( q_{o_i} \). However, \( a_i = \sum_{j=1}^{i-1} A(v_j) = r_i \). Therefore, \( q_{r_i} = v_j \), satisfying the conditions of the lemma.

   iii) \( v_i \in O_2 \).
   Then by the definition of (acyclic, binary) data-flow graphs, \( \exists \) unique \( v_j, v_k \in V: (v_j, v_i, 0), (v_k, v_i, 1) \in E \). Hence, \( v_j \pi_G v_i \) and \( v_k \pi_G v_i \).
   The instruction sequence was constructed such that \( v_j, v_k \in\{v_1, \ldots, v_{i-1}\} \). Thus, in some preceding state, \( v_j \) was stored in \( q_{o_i} \), and in another preceding state, \( v_k \) was stored in \( q_{o_{i+1}} \). Since \( a_i = r_i \), the conditions of the lemma are satisfied. \( \square \)

The preceding lemma is the second major result of this chapter. Namely, it has been shown that (acyclic, binary) data-flow graphs can be used to generate valid instruction sequences for indexed queue machines. Furthermore, the evaluation on an indexed queue machine of such a sequence produces the correct results. Therefore, the indexed queue machine is the natural mechanism for evaluating acyclic data-flow graphs.

3.16. Conclusions

In this chapter, two queue-based execution models were introduced. It was shown that instruction sequences for the simple queue machine execution model can be efficiently generated from (binary) expression parse trees. Furthermore, queue machines have the potential to exploit pipelined ALUs more efficiently than stack machines.

The second execution model, the indexed queue machine, was shown to be useful for evaluating (acyclic, binary) data-flow graphs. In particular, such graphs were shown to be generators of valid instruction sequences for indexed queue machines.

Chapter 4

Data Flow on a Queue Machine

4.17. Introduction

In the preceding chapter, the notion of an indexed queue machine was introduced. It was shown that the indexed queue machine is the natural execution mechanism for the evaluation of acyclic data-flow graphs. In this chapter, the issues associated with programming indexed queue machines will be addressed. In particular, the processes involved in translating OCCAM [INMOS 1984] programs into queue machine instruction sequences will be described.
First, the notion of dynamic data-flow graph splicing will be introduced. This is a method by which subroutine calls, iteration, and conditional execution are done using acyclic data-flow graphs. Then, it will be shown how programs written in OCCAM can be partitioned into program fragments corresponding to acyclic data-flow graphs.

Next, the issues associated with actually constructing data-flow graphs from OCCAM program fragments will be discussed. The basic steps involved in constructing data-flow graphs from a high-level language are data-flow analysis and live-value analysis. Procedures for data-flow analysis and live-value analysis will be presented. In addition, it will be shown that sequencing the inputs to a graph has an important effect on performance and a procedure for doing this will be described. A major problem with conventional data-flow systems is their inability to deal with side-effects. A mechanism to deal with side-effects will be described. Finally, a heuristic algorithm for generating a queue machine instruction sequence from an acyclic data-flow graph will be presented.

All the preceding concepts and algorithms have been embodied in a prototype compiler that translates OCCAM programs into queue machine instruction sequences. The major components of the compiler will be described.

4.18. Dynamic Data-Flow Graph Splicing

In the previous chapter, it was shown that acyclic data-flow graphs can be used as generators of queue machine instruction sequences. However, only limited forms of computation are possible on such graphs. In terms of high-level languages, such graphs correspond to sequences of assignment statements in a single assignment language. In this section, a technique for implementing subroutines, conditional execution, and iteration will be described. The basis of this method is the partitioning of a program into a collection of acyclic data-flow graphs and the connecting together of these graphs at execution time using a mechanism called dynamic data-flow graph splicing.

The dynamic data-flow graph splicing mechanism is based on the notions of channels and contexts. A channel is an abstract entity whose purpose is to provide a unidirectional communications path between two contexts. All channels have unique channel identifiers. A context represents the state of a process that is evaluating an acyclic data-flow graph. A context consists of 1) a queue machine instruction sequence (together with a program counter) corresponding to an acyclic data-flow graph, 2) an operand queue used to evaluate the acyclic data-flow graph according to the queue machine execution model, and 3) a pair of channels, called the in and out channels of the context, used for intercontext communications. Conditional execution, iteration, and subroutine calls are implemented using special queue machine instructions for context generation and intercontext communications. Since these instructions connect the data-flow graphs in different contexts at execution time, the technique is called dynamic data-flow graph splicing.

There are two queue machine instructions used for intercontext communications. They are send and receive (represented by the symbols ! and ?, respectively). The data-flow graphical representation of these instructions is shown in Figure 4.1. The send instruction is a two operand instruction. The first is a channel identifier, $c$, and the second is a value, $x$, to be transmitted on the channel. The receive instruction has as its single operand a channel identifier, $c$. The result of the receive instruction is the value, $x$, received on the channel. A context executing the send instruction on a given channel blocks until another context executes a receive on the same channel. Similarly, a context executing a receive instruction on a given channel blocks until another context executes a send instruction on the same channel. Thus, the send and receive actors are blocking primitives that provide an unbuffered, simplex communications path. Such a strategy is sometimes called a rendezvous [Tanenbaum 1985].
The send and receive instructions are non-standard data-flow actors in two senses. First, they are not free from side-effects. Both send and receive affect the execution of other contexts. Second, the receive instruction is not a function in the mathematical sense. That is, its result is not strictly a function of its operand.

Since the send and receive instructions are non-standard data-flow actors, the order in which they are executed affects the outcome of the computation. To ensure deterministic results, the instructions must be sequenced. To provide a means for sequencing these instructions, they are augmented as shown in Figure 4.2. The inputs and outputs labelled $K_1$ and $K_2$ carry operands called control tokens whose sole purpose is to sequence the instructions. These arcs are an artifact of the data-flow graphical representation of the computation — they do not appear in the queue machine instruction sequence derived from the data-flow graph.

Figure 4.3 illustrates the use of control token arcs to sequence the transmission of a pair of values between two contexts using a single channel. In this example, the two values, $x$ and $y$, are transferred in order from the sending context to the receiving context over a single channel, $c$. The control token $K_1$ ensures that the sender sends the values in order. The control token $K_2$ ensures that the receiver receives the values in the same order.

There are two queue machine instructions used for the generation of new contexts. They are rfork (recursive fork) and ifork (iterative fork). The data-flow graphical representation of these instructions is shown in Figure 4.4. Both fork instructions are single-operand instructions. In both cases, the operand is to be interpreted as a pointer to (i.e. the address of) a queue machine instruction sequence. The effect of the fork instruction is to create a new context. The instruction sequence associated with the new context is that specified by the operand. The operand queue of the new context is initially empty. The in and out channels of the new context depend on the kind of fork executed.

When an rfork is executed, a new context with two new channels is created. The results of the rfork instruction are the channel identifiers of two distinct channels called the in and out channels of the newly created context. The parent context directly receives the channel identifiers since it may have many children and, therefore, needs a way to distinguish among them. The same identifiers are incorporated into the state of the child context. The meta-identifiers in and out implicitly refer to these channel identifiers. Using these channel identifiers, the parent context can send operands to the child context (on $c_{in}$) and the child can send its results back to the parent.
Augmented data-flow actors for intercontext communication: Send (a) and Receive (b).

When an ifork is executed, a new context with one new channel is created. The new channel is the in channel of the child context. The child context inherits the out channel of the parent context. The result of the ifork is the channel identifier of the in channel of the child context. Using this identifier, the parent context can send operands to the child context (on c_{in}). Note that the parent cannot receive results from its child. However, using the inherited out channel, the child context can send results to a predecessor context of the parent. The ifork instruction is used to implement iteration and it facilitates loop unravelling.

Dynamic data-flow program splicing can also be used to implement conditional execution and iteration. The program of Figure 4.6 illustrates both conditional execution and iteration. This example also introduces comparison instructions and the select (sel) instruction. Comparison instructions produce a Boolean result. The result of a select instruction is its second or third operand depending on whether the first operand is true or false (respectively). In Figure 4.6, the select instruction chooses either the loop body or the loop terminator as the argument to the fork. Thus, a new context is generated for each iteration of the loop. Similarly, a new context would be generated for each branch of conditional execution. Note that the main context uses an rfork to create the first iteration of the loop, whereas the loop body uses ifork to create subsequent iterations. This allows the final iteration of the loop (i.e. the loop terminator) to send the results of the loop directly to the main context via the out channel. The advantage of the iterative (ifork) approach rather than a recursive (rfork) approach is that resources allocated to an iteration...
context can be released as soon as that iteration is completed.
The dynamic data-flow graph splicing mechanism was first described in [Preiss 1984] and later modified in [Preiss 1985]. The semantics of the send and receive actors as presented here represent a further modification of the earlier work. In particular, the semantics of send and receive were chosen to correspond to the semantics of the OCCAM programming language output and input statements. The underlying notion of dynamic graph splicing remains the same.

- lxiv -
\textbf{var} sum, result:
\textbf{seq}
\hspace{1em} sum := 0
\hspace{1em} \textbf{seq} k = [1 \textbf{for} 10]
\hspace{1em} sum := sum + k
\hspace{1em} result := sum

\textbf{Figure 4.6.}

Iteration on a Queue Machine.
4.19. Partitioning OCCAM Programs into Acyclic Graphs

In the previous section, it was shown that dynamically created contexts that execute acyclic data-flow graphs can be combined using the dynamic data-flow graph splicing mechanism to build arbitrary programs. In this section, the issue is the selection of a suitable source programming language and the method by which programs written in that language can be partitioned into a collection of acyclic data-flow graphs. The target architecture is a multiprocessor queue machine. Thus, the goal in partitioning the program is the exploitation of the parallelism inherent in the program. The granularity of parallelism in this approach is controlled by the amount of computation within a context. The computation within a context is performed using the indexed queue machine execution model. Parallelism is achieved by using multiple queue machines, each responsible for executing its own set of contexts.

Choosing the size of a context, i.e. the number of nodes in the acyclic data-flow graph, involves a trade-off between the overhead of context generation and intercontext communication against the performance benefits of parallelism. If the size of contexts is decreased, then more contexts must be created and more data must be communicated between contexts. If contexts are made larger, then performance suffers since queue machines cannot exploit all the intracontext parallelism.

The choice of the context as the basic granule of computation represents a compromise between the conventional data-flow execution model on the one hand, and task- or process-level parallelism of languages such as Concurrent Euclid and Ada on the other. The conventional data-flow execution model attempts to exploit the fine-grained parallelism at the level of individual operators and instructions. Limited success has been made in this approach because of the overhead associated with detecting operator-level parallelism at execution time. On the other hand, the task- or process-based approach relegates the responsibility for the detection of parallelism to the programmer. The programmer must explicitly partition a program into granules of computation. The goal in the context-based approach is to attempt to partition a program into granules of computation which are more complex than single instructions, yet less complex than processes or tasks, and to automatically exploit the parallelism between contexts. The basic criterion for the size of a context is that the computation it contains corresponds to an acyclic data-flow graph.

The programming language chosen should lend itself as naturally as possible to the context-based paradigm for parallelism. Ideally, the separation into contexts could be based on implicit as well as explicit cues in the program itself. Languages developed for data-flow architectures such as VAL [Dennis 1984b], ID [Arvind 1978], SALAD [Christoper 1981], and LAU [Plas 1976], reflect the underlying execution mechanism. These languages are designed to permit the automatic detection of operator-level parallelism at execution time. Such languages are usually single-assignment languages and they ensure that all actors are free from side-effects and purely functional. It is felt that these restrictions unduly constrain the programmer. In addition, these restrictions are not needed in the queue machine execution model where side-effects and non-functional actors are allowed. On the other hand, languages such as Concurrent Euclid [Holt 1983] and Ada [DoD 1983] require the programmer to explicitly specify all the parallelism and intercontext communication. Such an approach would be extremely tedious for the context-level of process granularity. In addition, such approaches usually enforce a fixed (at compile time) number of parallel processes, whereas the concern here is with dynamically created contexts.

The OCCAM programming language, originally developed for the Transputer computer architecture, has several features which make it suitable for the context-based approach to parallelism [Whitby-Strevens 1985]. OCCAM is based on the communicating sequential process (CSP) paradigm for parallel programming [Hoare 1978]. This paradigm naturally maps to contexts and intercontext communication. In addition, groups of statements in OCCAM must be
combined into a hierarchy of sequential and parallel parts using the operators \texttt{seq} and \texttt{par}. This static hierarchy can be used to partition the program into a collection of acyclic data-flow graphs. Finally, an implementation of OCCAM is allowed to support dynamic process creation. The method for doing this is shown in the OCCAM program fragment of Figure 4.7. The replicated \texttt{par} statement causes \(n\) occurrences of \texttt{Process} to execute in parallel, where \(n\) can be determined at run time. This construct is easily implemented using the dynamic data-flow graph splicing mechanisms discussed in the preceding section.

\begin{verbatim}
seq
  n := a value determined at run time
par i = [1 for n]
  Process (i)
\end{verbatim}

Figure 4.7.

OCCAM program fragment illustrating dynamic process creation.

A complete discussion of the OCCAM programming language is not necessary for the purpose of this section. The way in which OCCAM programs are partitioned into acyclic data-flow graphs can be described without using too many language details. There are five primitive statements (processes in OCCAM parlance), namely assignment, input, output, wait (real-time synchronization), and skip (no operation). (The wait actor will be discussed in a later section.) Each of these statements can be translated directly into an acyclic data-flow graph. Figure 4.8 illustrates how this can be done.

Groups of statements can be combined using one of two constructs — \texttt{seq} and \texttt{par}. The resulting data-flow graph is constructed from the data-flow graphs of the components as shown in Figure 4.9.(a) and Figure 4.9.(b). The inputs and outputs from the component data-flow graphs may be both control and data tokens. (There is no significance to the fact that only three inputs and outputs are shown — each component may have an arbitrary number of inputs and outputs.) In the implementation of the \texttt{par} construct, the outputs are synchronized using the \texttt{and} (\(\wedge\)) actor. (This actor computes the bitwise logical and of its two arguments.) The semantics of OCCAM ensure that at most one component process of a \texttt{par} construct will modify a variable. Thus, the only tokens that need to be synchronized are the control tokens.

All the preceding constructs result in a single data-flow graph. The dynamic splicing mechanism requires separate contexts for subroutines, iteration, and conditional execution. Thus, a new graph is created for each OCCAM subroutine (i.e., \texttt{proc} construct). The two iteration paradigms in OCCAM, the \texttt{while} loop and the replicated \texttt{seq}, are both implemented according to the method of iteration described in the previous section. That is, a new graph is created for the body of the loop and a loop terminator returns the results to the main program graph. Conditional execution in OCCAM is done with the \texttt{if} statement. A separate graph is created for the body of each branch of the \texttt{if} as described in the preceding section.

Finally, there is the replicated \texttt{par} construct of OCCAM which is used to dynamically create parallel processes. The implementation of this construct using the dynamic data-flow graph splicing paradigm is shown in Figure 4.10. The approach is similar to that of iteration — a new context is created for each instance of the \texttt{par} body. The difference is that \texttt{rfork} is used to create the new contexts instead of \texttt{ifork}. This is done so that execution can be resynchronized after all of the parallel contexts in the \texttt{par} body have completed execution.
<table>
<thead>
<tr>
<th>primitive</th>
<th>example</th>
<th>actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>$z := x + y$</td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>input</td>
<td>$c ? x$</td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>output</td>
<td>$c ! x$</td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>wait</td>
<td><strong>wait now after $t$</strong></td>
<td><img src="" alt="Diagram" /></td>
</tr>
<tr>
<td>skip</td>
<td><strong>skip</strong></td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
4.20. Data-Flow Analysis

In the previous section, it was shown how a program written in the OCCAM programming language can be partitioned into a collection of data-flow graphs. A technique for constructing the data-flow graph given an OCCAM program will be described in this section. This method is based on the flow analysis procedure for the translation of high-level languages to a data-flow graph presented in [Allan 1979] and [Allan 1980]. There are two goals in data-flow analysis. First, data-flow analysis involves the collection and organization of the information needed to construct a data-flow graph. Second, the live-value analysis phase of data-flow analysis provides optimization information which can be used to reduce intercontext communications.
The first step in the procedure is the translation of the OCCAM program (by a compiler) into an intermediate form table (IFT). Each entry of the IFT has five fields:

i) the type of the entry,

ii) $I$, the set of input values for the entry,

iii) $O$, the set of output values for the entry,

iv) $T$, the syntax tree associated with the entry, and

v) $E = \{E_1, E_2, \ldots, E_n\}$, an ordered set of ordered sets $E_i$ of the IFT indices corresponding to the components of the body of the entry.

The IFT entry format described in [Allan 1979] and [Allan 1980] contains only the first four fields. There are two benefits to adding the $E$ set. First, it is required to support the par construct (not present in [Allan 1979] and [Allan 1980]). Second, it greatly simplifies the data-flow analysis algorithms presented in [Allan 1979] and [Allan 1980].
There are two kinds of IFT entry. *Non-interface* IFT entries correspond to simple high-level entities, i.e. OCCAM primitives, conditions, and replicators (see Table 4.1). These entries have an empty $E$ set and may have non-empty syntax trees.

The *interface* IFT entries correspond to compound high-level statements and may have non-empty $E$ sets, but always have empty syntax trees. An interface entry represents a staging area for values used within and produced by a block of instructions. The rules for constructing the interface IFT entries for OCCAM programming language constructs are shown in Table 4.2.

There are three steps in the data-flow analysis procedure. First, the IFT is constructed during the parsing of the OCCAM program and the $I$, $O$, and $E$ sets are constructed as described in the tables. Second, The $I$ and $O$ sets are modified by augmenting each value with two sets, $D$ and $U$, which establish a relationship between the definition and subsequent use of a value.

![Diagram of process creation on a queue machine](image_url)

```
par i = [0 for 10]
var square:
square := i * i
```

**Figure 4.10.**

Dynamic process creation on a queue machine.
Table 4.1.

Non-interface Intermediate Form Table (IFT) entries.

<table>
<thead>
<tr>
<th>type</th>
<th>example</th>
<th>I</th>
<th>O</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>$z := x + y$</td>
<td>${x, y}$</td>
<td>${z}$</td>
<td>$\emptyset$ ($:= , z , (+, x, y)$)</td>
</tr>
<tr>
<td>input</td>
<td>$c ? x$</td>
<td>${K, c}$</td>
<td>${K, x}$</td>
<td>$\emptyset$ ($?, c, x$)</td>
</tr>
<tr>
<td>output</td>
<td>$c ! x + y$</td>
<td>${K, c, x, y}$</td>
<td>${K}$</td>
<td>$\emptyset$ ($!, c, (+, x, y)$)</td>
</tr>
<tr>
<td>wait</td>
<td>wait now after $x$</td>
<td>${K, x}$</td>
<td>${K}$</td>
<td>$\emptyset$ ($wait, x$)</td>
</tr>
<tr>
<td>skip</td>
<td>skip</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$ ($))</td>
</tr>
<tr>
<td>condition</td>
<td>$x &lt; y$</td>
<td>${x, y}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$ ($&lt;, x, y$)</td>
</tr>
<tr>
<td>replicator</td>
<td>$i = [x for y]$</td>
<td>${x, y}$</td>
<td>${i}$</td>
<td>$\emptyset$ ($rep, i, x, y$)</td>
</tr>
</tbody>
</table>

Table 4.3.

Sample OCCAM program fragment and corresponding Intermediate Form Table.

seq

\[
\begin{align*}
\text{x} & := x + 1 \\
\text{y} & := x
\end{align*}
\]

The definitions of $U$ and $D$ presented here are a generalization of those presented in [Allan 1979] and [Allan 1980]. This method is different in that the elements of the $O$ set have both a $U$ and $D$ set associated with them, whereas [Allan 1979] and [Allan 1980] use only a $U$ set.

The third and final step in the data-flow analysis procedure is live-value analysis, which associates with each value in every $O$ set a Boolean-valued tag which indicates whether the value
Table 4.2.

Interface Intermediate Form Table (IFT) entries.

<table>
<thead>
<tr>
<th>construct</th>
<th>$I, O, E$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>seq</strong> $P_1$</td>
<td>$I = I(P_1) \cup \bigcup_{i=2}^{n} (I(P_i) - \bigcup_{j=1}^{i-1} O(P_j))$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$O = \bigcup_{i=1}^{n} O(P_i)$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$E = {\rho_1, \rho_2, \ldots, \rho_n}$</td>
</tr>
<tr>
<td>$P_n$</td>
<td>where $\rho_i$ is the IFT index of $P_i$.</td>
</tr>
<tr>
<td><strong>par</strong> $P_1$</td>
<td>$I = \bigcup_{i=1}^{n} I(P_i)$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$O = \bigcup_{i=1}^{n} O(P_i)$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$E = {\rho_1, {\rho_2}, \ldots, {\rho_n}}$</td>
</tr>
<tr>
<td>$P_n$</td>
<td></td>
</tr>
<tr>
<td><strong>if</strong> $C_1$ $P_1$</td>
<td>$I = \bigcup_{i=1}^{n} {I(C_i) \cup (I(P_i) - O(C_i))}$</td>
</tr>
<tr>
<td>$C_2$ $P_2$</td>
<td>$O = \bigcup_{i=1}^{n} O(C_i) \cup O(P_i)$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$E = {\gamma_1, \rho_1, \gamma_2, \rho_2, \ldots, \gamma_n, \rho_n}$</td>
</tr>
<tr>
<td>$C_n$ $P_n$</td>
<td>where $\gamma_i$ is the IFT index of $C_i$.</td>
</tr>
<tr>
<td><strong>while</strong> $C_1$ $P_1$</td>
<td>$I = I(C_1) \cup (I(P_1) - O(C_1))$</td>
</tr>
<tr>
<td></td>
<td>$O = O(C_1) \cup O(P_1)$</td>
</tr>
<tr>
<td></td>
<td>$E = {\gamma_1, \rho_1}$</td>
</tr>
<tr>
<td><strong>seq</strong> $R_1$ $P_1$</td>
<td>$I = I(R_1) \cup (I(P_1) - O(R_1))$</td>
</tr>
<tr>
<td></td>
<td>$O = O(P_1)$</td>
</tr>
<tr>
<td></td>
<td>$E = {r_1, \rho_1}$</td>
</tr>
<tr>
<td>where $r_1$ is the IFT index of $R_1$.</td>
<td></td>
</tr>
<tr>
<td><strong>par</strong> $R_1$ $P_1$</td>
<td>$I = I(R_1) \cup (I(P_1) - O(R_1))$</td>
</tr>
<tr>
<td></td>
<td>$O = O(P_1)$</td>
</tr>
<tr>
<td></td>
<td>$E = {r_1, \rho_1}$</td>
</tr>
</tbody>
</table>

has a subsequent use in the program.

The $U$ and $D$ sets are constructed using the algorithm of Figure 4.11. The UseAndDef procedure is a top-down recursive descent algorithm which calls FindDef to link the elements of the $I$ and $O$ sets of IFT entries. The FindDef procedure scans through the ordered list of IFT indices $P$ (corresponding to statements preceding $H_j$ in the current scope) searching for an entry whose output set contains the value $x$. If it is not found in $P$, then FindDef checks the input set of $H$, the interface entry for this scope, to see if the value is imported from the enclosing scope. If the definition is found, the appropriate $U$ and $D$ sets are modified to reflect the dependency.

The procedure LiveAnalyze shown in Figure 4.12 does the live-value analysis. The rules for determining whether a value is live are as follows:
**procedure** UseAndDef \((H: \text{IFTindex})\)

**local var** \(P: \text{ordered list of IFTindex}\)

**for each** \(E_i \in E(H)\)

\(P \leftarrow \emptyset\)

**for each** \(H_j \in E_i\)

**for each** \((x, D, U) \in I(H_j)\)

FindDef \((x, H_j, H, P, D)\)

**end for each**

UseAndDef \((H_j)\)

\(P \leftarrow \text{concatenate} \left\{H_j\right\}, P\)

**end for each**

**for each** \((x, D, U) \in O(H)\)

FindDef \((x, H, H, P, D)\)

**end for each**

**end for each**

**end procedure** UseAndDef

**procedure** FindDef \((x: \text{symbol}, H_j, H: \text{IFTindex},\) \(P: \text{ordered list of IFTindex, D: set of IFTindex})\)

**for each** \(H_k \in P\)

if \((x, D', U') \in O(H_k)\) then

\(U' \leftarrow U' \cup \{H_j\}\)

\(D \leftarrow D \cup \{H_k\}\)

return

**end if**

**end for each**

if \((x, D', U') \in I(H)\) then

\(U' \leftarrow U' \cup \{H_j\}\)

\(D \leftarrow D \cup \{H\}\)

**end if**

**end procedure** FindDef

Figure 4.11.

Algorithm for generating Use and Definition sets.

1. A value which has a non-empty \(U\) set which is not equal to \(\{H\}\), the containing interface IFT entry, is live.
2. A value whose \(U\) set is \(\{H\}\) and the entry type of entry \(H\) is a loop (i.e. while or replicated seq) and the value is in the input set of the loop is live. Otherwise, the liveness of the value is obtained from the enclosing scope.
3. All var formal procedure arguments are live.

Once the IFT has been constructed and data-flow analysis completed, the data-flow graph can be easily constructed. The augmented IFT itself contains a representation of the data-flow graphs corresponding to the program. The process of constructing the actual graphs is a simple, but tedious job of translating the IFT to a suitable representation.
procedure LiveAnalyze (H: IFTindex)
  for each $E_i \in E(H)$
    for each $H_j \in E_i$
      for each $(x, D, U, Live) \in O(H_j)$
        if $U \neq \emptyset$ then
          if type($H$) is a loop and
            $x \in I(H)$ then
            Live $\leftarrow$ true
          else
            find $(x, D', U', Live') \in O(H)$
            Live $\leftarrow$ Live'
          end if
        else
          if $x$ is a var formal then
            Live $\leftarrow$ true
          else
            Live $\leftarrow$ false
          end if
        end if
      end for each
    end for each
  end for each
end procedure LiveAnalyze

Figure 4.12.
Algorithm for live-value analysis

4.21. Intercontext Communication

The basis of the dynamic data-flow graph splicing mechanism is the partitioning of a program into a collection of acyclic data-flow graphs which are spliced together at execution time via the mechanism of intercontext communication over channels. In the preceding sections, it was shown how OCCAM programs can be partitioned on the basis of certain language structures. In addition, it was shown how data-flow graphs can be constructed from such OCCAM program fragments. In this section, the issue is how to coordinate the communication of values (operands and results) between contexts. Since the dynamic data-flow splicing mechanism relies on the communication of values between the calling context and the called context over a single channel, the values transmitted must be sent in some sequence. Thus, the issue in this section is the method of selecting an appropriate sequence.

The optimum sequence of values for intercontext communication is that sequence which results in the minimum total execution time for a given program. Unfortunately, this definition is unworkable because it requires that a global optimization be performed on all the graphs associated with a given program. Since a program is partitioned into many small graphs, each of which transmits and receives many values, the search space of possible value sequences becomes large. Instead, the following heuristic assumption is made: The preferred sequence of values for intercontext communication is that sequence which maximizes the amount of computation possible in a context before the context must wait for another input.
For example, consider a graph that has two inputs \( a \) and \( b \) in which the amount of computation possible after \( a \) is received but before \( b \) is needed is \( C_a \) and the amount of computation possible after \( b \) is received but before \( a \) is needed is \( C_b \) such that \( C_a > C_b \). Then the inputs to the graph should consist of \( a \) followed by \( b \) since this sequence maximizes the amount of work done before the context executing the graph must wait for another input.

The preceding concepts will now be presented more formally. First, a partial order relation on the inputs of an acyclic data-flow graph will be defined. This partial order relation describes the sequencing constraints on the inputs to the graph. Finally, an algorithm to compute a sequence of the inputs to the graph which satisfies the partial order constraints will be presented.

**Definition:** The set of immediate predecessors of vertex \( v \in V \) of a DAG \( G = (V, E) \) is the set \( P(v) \subseteq V \) given by \( P(v) = \{ u \in V : (u, v) \in E \} \).

**Definition:** The set of immediate successors of vertex \( v \in V \) of a DAG \( G = (V, E) \) is the set \( S(v) \subseteq V \) given by \( S(v) = \{ w \in V : (v, w) \in E \} \).

**Definition:** An acyclic data-flow graph is a DAG \( G = (V, E) \) with the following properties:

i) its vertices are either inputs \( I \) or operators \( O \), i.e. \( (V = I \cup O) \land (I \cap O = \emptyset) \).

ii) \( \forall v \in I: P(v) = \emptyset \).

iii) \( \forall v \in O: |P(v)| = A(v) \) where \( A(v) \) is the arity of operator \( v \) (i.e., the number of operands).

The preceding definition of acyclic data-flow graph is a modification of the one presented in the previous chapter. In addition to operators, some of the vertices of a data-flow graph are inputs. No input vertex has an edge entering it. Also, the operators can have an arbitrary (but finite) number of operands.

**Definition:** The predecessor set \( P^*(v) \) of a vertex \( v \) of a DAG \( G = (V, E) \) is the set of all the nodes in \( G \) preceding \( v \) and including \( v \) itself, i.e. \( P^*(v) = \{ v \} \cup \bigcup_{u \in P(v)} P^*(u) \).

**Definition:** \( C(v) \) is the cost of the computation associated with a node \( v \) of an acyclic data-flow graph \( G = (V, E) \), given by \( C(v) = |P^*(v)| \).

**Definition:** The required input set \( I^*(v) \) of a node \( v \) of an acyclic data-flow graph \( G = (V, E) \) is the set \( I^*(v) = P^*(v) \cap I \).

The preceding definitions associate a cost of computation and the inputs required for that computation with every node of an acyclic data-flow graph. In this discussion, the cost associated with a node is simply the size of its predecessor set. In practice, the cost will be the sum of the execution times of the operators in the node’s predecessor set. Using these concepts, it is now possible to define a relation on the set of inputs \( I \) of an acyclic data-flow graph.

**Definition:** \( \pi_I \) is a relation defined on the set of input nodes of an acyclic data-flow graph \( G = (V, E) \) given by:

\[
\forall a, b \in I: a \pi_I b \iff (a = b) \lor ((a \neq b) \land (\sum_{v \in V: a \in \Gamma(v)} C(v) > \sum_{w \in V: b \in \Gamma(w)} C(w)))
\]

**Lemma:** \( \pi_I \) is a partial order.
Proof: It will be shown that $\pi_I$ is reflexive, antisymmetric, and transitive:

1) (reflexive) By definition, $\forall a \in I: a \pi_I a$.

2) (antisymmetric) Given $a \pi_I b \land b \pi_I a$. Let $x = \sum_{v \in V: a \notin I'(v)} C(v)$, and $y = \sum_{v \in V: b \notin I'(v)} C(v)$.

   Assume that $a \neq b$. Then $x > y \land y > x$ (contradiction).

   $\therefore a = b$.

3) (transitive) Given $a \pi_I b \land b \pi_I c$, $a \neq b \neq c \neq a$. Let $x = \sum_{v \in V: a \notin I'(v)} C(v)$, and $y = \sum_{v \in V: b \notin I'(v)} C(v)$, and $z = \sum_{v \in V: c \notin I'(v)} C(v)$. Then $x > y \land y > z => x > z$.

   $\therefore a \pi_I c$. \hfill $\square$

Definition: A sequence $\{v_1, v_2, \cdots v_{|V|}\}$ of the inputs $I$ of an acyclic data-flow graph $G = (V, E)$ satisfies the relation $\pi_I$ if and only if

$\forall i, j, 1 \leq i < j \leq |V|: \neg(v_j \pi_I v_i)$.

An algorithm for constructing a sequence of the inputs to an acyclic data-flow graph satisfying the $\pi_I$ relation will now be described. The first step of the algorithm is to construct a depth-first list $L = \{l_1, l_2, \cdots, l_{|V|}\}$ of the vertices of the graph $G = (V, E)$. A depth-first list has the property that all the successors of a node precede the node in the list. Conversely, all the predecessors of a node follow the node in the list. An algorithm adapted from [Aho 1974] to compute the depth-first list is shown in Figure 4.13.

```
procedure ConstructDepthFirstList
  initially all nodes $v \in V$ are unmarked
  $i \leftarrow 1$
  loop
    exit when all nodes $v \in V$ are marked
    choose an unmarked $v \in V$
    Search ($v$)
  end loop
end procedure ConstructDepthFirstList

procedure Search ($n$: node)
  mark $n$
  for each $m \in S(n)$
    if $m$ is not marked then
      Search ($m$)
    end if
  end for each
  $l_i \leftarrow n$
  $i \leftarrow i + 1$
end procedure Search
```

Figure 4.13.

Algorithm for constructing a depth-first list of the nodes of a DAG.
As an example, the data-flow graph for the expression $e := ((a + b) \times (-c)) \div d$ is shown in Figure 4.14.(a). The depth-first list of the nodes of this graph is $L = \{e, \div, \times, +, a, b, -, c, d\}$, which is obtained by ordering the nodes as shown in figure Figure 4.14.(b).

The second step of the algorithm is to compute for each vertex $v \in V$ of the data-flow graph the sets $P^*(v)$ and $I^*(v)$ and the computational cost $C(v)$. This is easily done using the depth-first list of nodes. The algorithm to do this computation is shown in Figure 4.15.

The results of applying the algorithm of Figure 4.15 to the acyclic data-flow graph of Figure 4.14.(a) are summarized in Table 4.4.

The final step of the algorithm involves computing the values $W(v) = \sum_{u \in V; v \in I^*(u)} C(u)$ for each of the inputs $v$ to the data-flow graph and then sorting the input set $I$ according to $W$. An algorithm to do this is shown in Figure 4.16.

The results of applying the algorithm of Figure 4.16 to the example of Figure 4.14.(a) are summarized in Table 4.5. From this it is possible to determine that there are two suitable sequences of the inputs, namely $\{a, b, c, d\}$ and $\{b, a, c, d\}$.

4.22. Dealing with Side-Effects

Conventional data-flow systems are based solely on actors that are both functional and free from side-effects. These requirements cause difficulties in handling input/output operations, since such operations are neither functional nor side-effect free. In addition, these restrictions make it impossible to modify vectors, matrices, and data bases since such modification is itself a side-effect. In this section, a technique for handling operators with side-effects will be described. The
for $i: |V| \cdots 1$

$P^*(l_i) \leftarrow \{l_i\}$

if $l_i \in I$ then
   $I^*(l_i) \leftarrow \{l_i\}$
else
   $I^*(l_i) \leftarrow \emptyset$
end if

for each $m \in P(l_i)$
   $P^*(l_i) \leftarrow P^*(l_i) \cup P^*(m)$
   $I^*(l_i) \leftarrow I^*(l_i) \cup I^*(m)$
end for each

$C(l_i) \leftarrow |P^*(l_i)|$
end for

Figure 4.15.
Algorithm to compute $P^*(v)$, $I^*(v)$, and $C(v)$ for the nodes $v \in V$ of an acyclic data-flow graph.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$l_i$</th>
<th>$P^*(l_i)$</th>
<th>$I^*(l_i)$</th>
<th>$C(l_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>d</td>
<td>{d}</td>
<td>{d}</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>{c}</td>
<td>{c}</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>{–, c}</td>
<td>{c}</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>{b}</td>
<td>{b}</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>{a}</td>
<td>{a}</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>{+, a, b}</td>
<td>{a, b}</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>×</td>
<td>{×, +, a, b, –, c}</td>
<td>{a, b, c}</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>÷</td>
<td>{÷, ×, +, a, b, –, c}</td>
<td>{a, b, c, d}</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
<td>{e, ÷, ×, +, a, b, –, c}</td>
<td>{a, b, c, d}</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 4.4.
$P^*(v)$, $I^*(v)$, and $C(v)$ for the nodes $v$ of the data-flow graph of Figure 4.14.(a).

The goal is to ensure that the data-flow graphs derived from OCCAM programs do not violate the OCCAM semantics.

The approach taken here is to identify all those actors which can have a side effect and to force the sequencing of those actors using control tokens. An actor with a control token input cannot proceed until that token arrives. An actor with a control token output emits a control token after it has completed execution. Operations having no control token inputs can be executed in any sequence based solely on the availability of operands. Operations having side effects in addition to the availability of operands are sequenced by the availability of the control token. Figure 4.17 shows a basic repertoire of actors with side effects.
for each $v \in I$

$W(v) \leftarrow 0$

for each $u \in V$

if $v \in I^*(u)$ then

$W(v) \leftarrow W(v) + C(u)$

end if

end for each

end for each

sort the elements of $I$ according to $W$

Figure 4.16.

Algorithm to form the sequence of the inputs to an acyclic data-flow graph satisfying the relation $\pi$.

<table>
<thead>
<tr>
<th>$v \in I$</th>
<th>$W(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>27</td>
</tr>
<tr>
<td>$b$</td>
<td>27</td>
</tr>
<tr>
<td>$c$</td>
<td>26</td>
</tr>
<tr>
<td>$d$</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 4.5.

$W(v)$ for the input nodes $v$ of the data-flow graph of Figure 4.14.(a).

The channel actors, send (!) and receive (?), have been described earlier in connection with dynamic data-flow graph splicing. The memory actors are store (mem !) and fetch (mem ?). These actors either store or retrieve a word from memory. (There are also actors which store and receive bytes from memory — storeb (memb !) and fetchb (memb ?)) The real-time actors are now, which returns the time $t$ of arrival of the control token, and wait, which delays execution until the current time exceeds $t$. This use of the control token to control in real time is similar to the notion of hiatonic data-flow [Faustini 1986], in which special tokens (called hiatons) are used to control real-time processes. The difference is that the control token approach is based on the realization that all side-effects must be sequenced (not just those related to real time).

To illustrate the importance of the control token concept, Figure 4.18 shows the data-flow graphs corresponding to an OCCAM program fragment with and without control tokens. Note that without the control token arcs, the graph does not reflect the sequencing implied in the OCCAM program. Even the conventional data-flow approach in which the channel identifier is conceptually modified by input/output statements would not enforce the correct sequencing, since these statements refer to different channels. However, the control token approach ensures that the data-flow graphs reflect the semantics of the OCCAM program.

The control token is like a program counter in a Von Neumann architecture in that it represents the locus of control of the program. The control token differs from a program counter in two respects. First, it can be replicated as a result of the par construct, resulting in several control tokens in one program. Second, the control token only visits those instructions having side-
Figure 4.17.

Data-flow actors with side-effects.

A strict interpretation of OCCAM semantics requires that there exists only one control token within each process and that the token is replicated only as a result of the \texttt{par} construct, with only one copy going to each parallel process. In particular, this means that I/O, memory access, and timing synchronization all use the same control token. To provide the ability to exploit parallelism more effectively, OCCAM semantics are violated in the following way: A separate control token is used for every OCCAM vector (array) in the program. This means that array accesses to different arrays need not be strictly sequenced since they affect different objects. Furthermore, the multiple readers/single writer paradigm is used. That is, consecutive fetch
operations from the same array can occur in any order without affecting the outcome of the computation. However, write operations to an array must be strictly sequenced to ensure deterministic execution. This concept is illustrated for an OCCAM program fragment in Figure 4.19. Notice that the fetches can occur in any order. However, the store must follow all the fetches. A synchronizing actor (Λ, i.e., the bitwise and actor) is included to accomplish this.

4.23. Generating Queue Machine Instruction Sequences

The method of constructing a collection of acyclic data-flow graphs representing an OCCAM program has been outlined in the preceding sections. In a preceding chapter, it was shown that an acyclic data-flow graph can be considered as a generator of valid queue machine
var c [byte 3], w, x, y, z:

seq
  x := c [byte 0]
  y := c [byte 1]
  z := c [byte 2]
  c [byte 0] := w

Figure 4.19.
Sequencing array accesses.

instruction sequences. In this section, the issue is the method of constructing a suitable instruction sequence.
The major criterion of a suitable instruction sequence is that it satisfies the partial order constraints imposed by the arcs of the data-flow graph. In general, there are many possible instruction sequences satisfying the partial order constraints. The approach taken here is to attempt to maximize parallelism and to minimize total execution time. Since the optimal sequence is difficult to determine directly, a heuristic approach is developed.

The basic algorithm to construct a sequence \( S = \{s_1, s_2, \ldots, s_{|V|}\} \) of the nodes \( s_i \in V \) of an acyclic data-flow graph \( G = (V, E) \) is shown in Figure 4.20. The set \( R \) is the set of instructions ready to execute. It initially contains those nodes which have no predecessors in \( G \). A node is selected from \( R \) (according to a heuristic described below) and added to the instruction sequence. Then, all the successors of that node are checked to see if they have all their operands and can thus be placed in \( R \). These steps are repeated until \( R \) becomes empty. For a correctly constructed graph, this occurs when all the nodes have been exhausted.

\[
i \leftarrow 1
\]

\[
\text{initially all edges } e \in E \text{ are unmarked}
\]

\[
\text{initialize } R \leftarrow \{v \in V: P(v) = \emptyset\}
\]

\[
\text{loop}
\]

\[
\text{exit when } R = \emptyset
\]

\[
\text{select a node } v \in R
\]

\[
s_i \leftarrow v
\]

\[
i \leftarrow i + 1
\]

\[
\text{for each } w \in S(v)
\]

\[
\text{mark edge } (v, w) \in E
\]

\[
\text{if } \forall u \in P(w): (u, w) \text{ is marked then}
\]

\[
R \leftarrow R \cup \{w\}
\]

\[
\text{end if}
\]

\[
\text{end for each}
\]

\[
\text{end loop}
\]

**Figure 4.20.**

Algorithm to generate a queue machine instruction sequence from an acyclic data-flow graph.

The key step in the algorithm is the selection of the next instruction to emit. A heuristic is proposed which is based on the following assignment of priorities to the instructions:

1. rfork and ifork (highest priority),
2. send,
3. store and storeb,
4. others not explicitly mentioned,
5. fetch and fetchb,
6. receive, and
7. wait (lowest priority).

The highest priority instruction in \( R \) is chosen at each step of the algorithm of Figure 4.20. This assignment of priorities was chosen to attempt to maximize the number of parallel contexts and to minimize the total execution time. The fork actors are given the highest priority because
they create new contexts for parallel execution. The send is next because it transmits arguments
to newly created contexts and thus enables them to proceed. The receive and wait actors were
given the lowest priorities because they can result in a context becoming blocked or suspended.
Thus, it is more efficient to do work before possibly becoming suspended. Finally, memory store
operations have higher priority than “other” instructions since it is more efficient to remove values
from the operand queue (thus making it smaller) as soon as possible. Similarly, “others” pre-
cede memory fetch operations because it is more efficient to do as much computation with the
values currently in the queue before fetching new operands from memory (making the queue
larger).

4.24. Software System Overview

In this section, a prototype compiler that translates OCCAM programs into queue machine
instruction sequences will be described. This compiler embodies the concepts and algorithms
described in the preceding sections of this chapter.

The compiler system consists of the eleven programs shown in Figure 4.21 together with a
controlling program which coordinates the activities of the others. The input to the compiler is a
text file containing an OCCAM source program. The input to each component of the compiler is
a representation of the source program. Each component of the compiler modifies the representa-
tion in some way. All the component programs were constructed by modifying a prototype com-
piler pass. The prototype compiler pass consists of a parser for the internal program representa-
tion language. The prototype pass constructs a parse tree of the program and then traverses it.
The output of the prototype pass is exactly the same as its input. This approach greatly simplified
the construction of the compiler and provided a means to check the output from each pass of the
compiler.

The scanparse pass of the compiler does the lexical and syntactic analysis of the OCCAM
source program. It translates the text file into the binary token stream representation used by the
remainder of the compiler. This representation is a machine-readable pre-order traversal of the
program parse tree.

The semantic pass of the compiler does the semantic analysis of the program. This pass
constructs a symbol table which is used by the remaining passes of the compiler. This pass also
does range and type checking as well as variable scope checking.

The dataflow pass of the compiler constructs the intermediate form table (IFT) as described
in a preceding section. This involves the construction of the use and definition sets as well as
live-value analysis.

The grapper pass constructs the data-flow graphs as described in a preceding section. This
pass also determines the sequence for intercontext communications according to the method
described earlier. The output of this pass is a text file containing a description of the resulting
data-flow graphs.

The sequencer pass selects a sequence of the nodes of the data-flow graph according to the
heuristic actor priorities described earlier. The output of this pass is a text file containing a
description of the data-flow graph. This pass does not alter the data-flow graph itself. It merely
reorders the sequence of the nodes in the description of the graph.

Several programs which are not part of the compiler were written to aid the development of
the prototype compiler system. The dump program produces a human-readable representation of
the machine-readable symbol table produced by the semantic pass. The display program pro-
duces a human readable representation of the binary token stream format used between passes of
the compiler. The draw and drawpic programs produce appropriately formatted graphical repre-
sentations of the data-flow graphs produced by the compiler. These programs contain
Franklin 4.21. Software system components.

ruderentary layout and routing routines. The draw program produces output suitable for use with an x-y plotter. The drawpic program produces output suitable for use with the PIC
The two remaining programs, *coder* and *assembler* generate queue machine assembly language and queue machine object code files for the proposed queue machine architecture described in the following chapter. The details of the queue machine assembly language and object code format will also be described there.

All these programs were written in the C programming language [Kernighan 1978] with the aid of the *YACC* [Johnson 1979] parser generator and the *LEX* [Lesk 1979] lexical analyzer generator utilities. A controlling program written for the BSD 4.2 UNIX operating system coordinates the activities of the component programs.

### 4.25. Conclusions

The issues associated with generating acyclic data-flow graphs and queue machine instruction sequences from OCCAM programs have been presented in this chapter. In particular, it was shown that the dynamic data-flow program splicing mechanism can be used to connect contexts executing acyclic data-flow graphs at execution time to implement subroutine calls, iteration, and conditional execution.

It was shown that OCCAM programs have constructs which facilitate the partitioning of programs into acyclic data-flow graphs. The procedures of data-flow analysis of OCCAM programs, namely the construction of use and definition sets and live-value analysis, were described in detail. In addition, algorithms for sequencing the inputs of an acyclic data-flow graph and for choosing a sequence of the nodes of the graph suitable for execution on a queue machine were described.

Finally, the prototype compiler system for translating OCCAM programs into queue machine instruction sequences was described. This compiler system embodies the concepts and algorithms presented in this chapter.

### Chapter 5

**On the Design of a Practical Queue Machine**

#### 5.26. Introduction

In Chapter 3, the indexed queue execution model was presented. The design of such a queue machine, suitable for practical implementation will be described in this chapter. It will also be shown how such a machine can be used as the basic processing element in a multiprocessor system designed for executing OCCAM programs that have been decomposed into a set of communicating, acyclic data-flow graphs. Such graphs were shown to be generators of queue machine instruction sequences. An approach to the compilation of such graphs was described in the preceding chapter.

First, a scheme for implementing an indexed queue based on the use of a set of special purpose registers, called window registers, will be presented. The effect of these registers is to automatically transform the basic instruction execution mechanism from the memory-to-memory paradigm to the register-to-register paradigm.

Next, the architecture of the proposed queue machine processing element will be described in some detail. The basic elements of the programmer’s model will be presented first. These include a discussion of the basic data objects handled by the processing element, a description of the register set, a specification of the instruction format, a list of the instruction set, and a brief discussion of the assembly language syntax for the proposed processing element. An important feature of the proposed design is the fact that it supports the conventional Von Neumann execution paradigm as well as the indexed queue machine execution model.
Following the specification of the programmer’s model, a data path architecture for the queue machine processing element is proposed. In addition, the detailed timing specifications for the various instruction classes are presented.

A fundamental element of the context-based indexed queue machine execution model described in the preceding chapter is the notion of communication channels. The queue machine processing element/channel interface will be specified and an approach to implementing channels using dedicated message processing hardware will be described.

Finally, a multiprocessor queue machine system architecture is proposed. This architecture is based on a partitioned bus that is configured in a ring topology.

5.27. Window Registers

In an earlier chapter, the evaluation of an indexed queue machine instruction sequence was described in terms of an abstract queue machine execution model. In this section, a practical approach for implementing the queue machine execution model will be described. In particular, a simple mechanism for efficient management of the queue of operands will be presented.

The basis of the indexed queue machine execution model is the use of a queue data structure to handle operands. Each instruction removes the required number of operands from the front of the queue of operands, performs some computation, and stores the results of the computation into the queue of operands at specified offsets from the front of the queue (overwriting previous values at those locations). This execution mechanism was specifically chosen because it can be easily implemented using a random-access, read/write memory. The queue of operands occupies contiguous memory locations. A special-purpose register, called the queue pointer (QP), contains the address of the first operand in the operand queue. Operands are retrieved from the front of the queue by reading the memory location indicated by the QP. Immediately after retrieving an operand, the QP is incremented so that it points at the next operand in the queue. Results are returned to the queue by storing them at the address which is the sum of the offset associated with the result and the contents of the QP.

The obvious problem with this approach is that the address of the front of the queue is always increasing. That is, the operand queue migrates through the memory. Furthermore, memory locations preceding the front of the queue no longer contain useful data, yet they will never be reused. This problem can be easily solved by restricting the operand queue to a page of memory. Thus, the contents of the QP are to be interpreted as a page number and a page offset. The page number remains constant, whereas the page offset is incremented modulo the size of the page. The size of the page, a power of two, is an architectural parameter. It places a restriction on the maximum queue length. The page size also affects the utilization of operand queue memory, where utilization is defined as the fraction of memory locations in use that contain valid data. The average utilization is simply the average queue length divided by the page size. Thus, the choice of a page size is a trade-off between the maximum queue length and average memory utilization.

A memory-based implementation of the queue machine execution model has the property that all the instructions executed by the machine are memory-to-memory instructions. This is not surprising since most data-flow-based machines also rely on memory-to-memory instructions. However, most implementation technologies have the property that memory operations (reads and writes) execute more slowly than most ALU operations. For this reason, it is desirable to use registers for operands wherever possible.

Suppose there are \( n \) CPU registers, called window registers corresponding to the first \( n \) elements of the operand queue. That is, the window registers are associated with the memory locations \( QP, QP + 1, \ldots, QP + n - 1 \) (with appropriate wrap around at page boundaries). As the QP
is incremented, the register window slides through memory. By choosing the size of the register window \( n \) such that \( n = 2^k \) for some \( k \), the least significant \( k \) bits of the page offset field of the QP select the physical register corresponding to the front of the queue. The relationship between the (virtual) queue, its state in a given page of physical memory, and the associated physical register window is shown in Figure 5.1. Note that associated with each window register is a presence bit which indicates either that the operand is available in the register, or that it must be retrieved from memory.

![Figure 5.1.](image)

Relationship between the operand queue, the queue pointer (QP), the page base address and the page offset, and the register window.

The operand-fetch phase of instruction execution begins by examining the presence bit of the window register associated with the front of the operand queue. If the bit is set, indicating the register contains a valid operand, the operand is simply retrieved from the register. If the bit is not set, a memory fetch operation must be done to retrieve the operand from memory. In both cases, once the operand has been obtained, the presence bit is cleared and the QP is incremented so that it points at the next operand.
The result-store phase of instruction execution begins by determining whether the offset associated with the result is smaller than $n$, the size of the register window. If it is, the result is stored in the appropriate window register and its associated presence bit is set. Otherwise, the result is stored in the appropriate memory location. If the instruction sequence has the property that most of the offsets associated with the results of instructions are less than the size of the register window, then many instructions will automatically execute in the register-to-register mode.

When a processor does a context switch, it must first save the state of the currently executing context. It does so by rolling out the window registers whose presence bits are set by storing the register contents into their associated memory locations. Note that the presence bits themselves are not stored in memory. When a context resumes executing, it begins with all the presence bits cleared. In this way, operands are automatically restored by the normal execution mechanism without the need for explicitly restoring the state of the register window after a context switch. (This is an improvement over earlier work which required explicit presence bits in memory [Preiss 1984] and [Preiss 1985].)

The size of the register window is an architectural parameter. The larger the register window, the more efficient the execution mechanism. This is because more instructions will be executed in the register-to-register mode. However, as the register window becomes larger, context switch overhead increases since more queue entries must be rolled out into memory.

The register window clearly acts as a cache for the operand queue. However, it differs from a conventional cache in two ways. First, at every point in the execution, there is a unique one-to-one mapping from memory locations to cache locations (as opposed to the usual case of many-to-one). Second, values are only written into memory on a context switch. As a result, it is possible that some values are never written through to memory.

In this scheme, values stored in window registers are read exactly once. This means that the maximum achievable register utilization efficiency is lower than that for a conventional, Von Neumann architecture. However, the ability of this scheme to utilize registers at all makes it more efficient than other data-flow architectures.

### 5.28. Queue Machine Processing Element — Programmer’s Model

In the preceding section, it was shown how a register-based processing element can be used to efficiently implement the indexed queue machine execution model. In this section, a proposed queue machine processing element architecture will be introduced. In particular, the elements of the programmer’s model will be described. These elements include a description of the basic data types, the register bank, the instruction format, and the instruction set. The details of the proposed implementation are described in the following section.

#### 5.28.1. Basic Data Objects

The basic data objects are a 32-bit word and an 8-bit byte. Memory is both word and byte addressable. The contents of a word can be interpreted in three ways: as a signed, two's-complement integer; as an unsigned value, usually a memory address; or as a Boolean value, with all zeroes corresponding to false, and all ones corresponding to true.

Although memory is byte addressable, all internal operations are done on words. Bytes are interpreted as unsigned quantities and are right-justified without sign extension when assigned to words. Also, when addressing words, the address must be word-aligned (i.e., the least significant two bits of the address must be zero).
5.28.2. Registers

The state of a queue machine processing element is contained entirely in a collection of 32 one-word registers. Figure 5.2 shows that the collection of registers is split into two banks of 16 registers — virtual window registers and global registers.

Figure 5.2.
Queue machine processing element register set.

Registers $R_0$ to $R_{15}$ are the virtual register names representing the sliding window registers. These registers correspond to the first 16 elements of the operand queue. (Recall that the general scheme of mapping from the virtual queue to the physical memory was shown in Figure 5.1). The register $R_0$ always corresponds to the first element of the operand queue (i.e., the front of the queue), $R_1$ corresponds to the second, etc. This means that the actual physical register selected depends on two things — the virtual register number and the queue pointer (QP), $R_{30}$. The mapping from virtual register numbers, $R_0$ to $R_{15}$, to physical register numbers is shown in Figure 5.3. It is simply the modulo 16 sum of the virtual register number and bits 5-2 of the QP. Since the operand queue contains only word operands, the address of the front of the queue is always word aligned. Therefore, the least significant two bits of the QP are always zero.

Registers $R_{16}$ to $R_{31}$ are (physical) global registers. These registers are directly selected; i.e., there is no virtual-to-physical register number translation done. This bank is further subdivided into two parts — general-purpose registers and special-purpose registers. Registers $R_{16}$ to $R_{27}$ are the general-purpose registers and may be freely used by the programmer. (In terms of executing programs from acyclic data-flow graphs, such registers are not needed. They have been included in the design of the machine to broaden its applicability to include Von Neumann-style
Virtual register number to physical register number translation.

Programming.) Registers R\textsubscript{28} to R\textsubscript{31} are special-purpose registers reserved for hardware and operating system support.

Register R\textsubscript{31} is the program counter (PC). It contains the address of the instruction immediately following the instruction currently being executed.

Register R\textsubscript{30} is the queue pointer (QP). It contains the address of the front of the operand queue. As stated earlier, this address is always word-aligned. The QP register consists of two parts — the page number and page offset as shown in Figure 5.4. The least significant 10 bits of the QP form the page offset. Thus, the maximum queue page size is 256 words. (The least significant two bits of the QP are always zero.) The most significant 22 bits of the QP form the page number.

Register R\textsubscript{29} is an 8-bit register called the page offset mask (POM). It is used to support variable-size queue pages. The details of its use will be described below.

Register R\textsubscript{28} is the NAK address register (NAR). When a memory or channel access fails, the instruction is aborted and the offending address is loaded into the NAR. This register is provided to facilitate the implementation of the operating system.
Each of the window registers has a presence bit associated with it. This bit indicates whether the window register contains a valid value. The presence bit is set when a value is stored into a window register. The presence bit is automatically cleared when an operand is removed from the front of the queue. If the presence bit is not set when a queue operand fetch is done, the operand is retrieved from memory. The memory address of the operand is computed using the QP, the POM, and the virtual register number as shown in Figure 5.5.

![Diagram](image)

**Figure 5.5.**
Virtual register number to memory address translation.

Each of the bits of the POM controls one bit in the calculation of the page offset. If the $i^{th}$ bit of the POM is set, then the $i^{th}$ bit of the page offset field of the QP is selected. If the $i^{th}$ bit of the POM is not set, then the $i^{th}$ bit of the sum of the page offset field of the QP and the virtual register number is selected. By loading the appropriate values into the POM, the page size can be varied by powers of two from 1 to 256. (It will be shown below that the minimum queue page is actually 32). Valid POMs begin with a block of ones, followed by a block of zeroes to the right. For example, if the POM value consists of $m$ zeroes on the right and $8 - m$ ones on the left, the page size of $2^m$ is selected.

### 5.28.3. Instruction format
The goal in the design of the instruction format is to provide a means for implementing the indexed queue machine execution model, while at the same time providing the ability to execute
conventional (Von Neumann-style) instruction sequences. To support the indexed queue machine model, instructions must be able to store their results in more than one position in the operand queue. To support conventional execution, instructions must be able to use any registers as operands (not just the front of the operand queue). Consequently, a four-address instruction format was selected — two source specifiers and two destination specifiers. The basic instruction format is shown in Figure 5.6.

![Instruction Format Diagram]

Figure 5.6. Basic instruction format.

All instructions are 32 bits wide. Two source operand fields, \( src_1 \) and \( src_2 \), specify the operands of the instruction. Two result destination fields, \( dst_1 \) and \( dst_2 \), specify where the result of the instruction is to be placed. The source fields can be interpreted in four ways. The various interpretations are summarized in Table 5.1. If the most significant two bits of the source field are 00, the least significant four bits of the source field select one of the 16 (virtual) window registers. If the most significant two bits of the source field are 01, the least significant four bits select one of the 16 global registers. If the source field is 110000, then immediately following the current instruction is a word constant to be used as the operand. For the remaining combinations, the least significant five bits represent a signed, twos-complement immediate operand in the range \(-15\) to 15 (inclusive).

<table>
<thead>
<tr>
<th>source field</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00nnnn2</td>
<td>window register ( nnnn2 )</td>
</tr>
<tr>
<td>01nnnn2</td>
<td>global register ( 1nnnn2 )</td>
</tr>
<tr>
<td>1100002</td>
<td>immediate word</td>
</tr>
<tr>
<td>1nnnnn2</td>
<td>small immediate ((-15) to 15) for ( nnnn2 \neq 100002 )</td>
</tr>
</tbody>
</table>

Table 5.1. Table of source operand modes.

The destination fields are 5-bit fields which specify one of the 32 registers. The number of destinations used is a property of the program being executed and not of the particular operation being done. By convention, when fewer than two destinations are used, the remaining fields are set to specify register \( R_{16} \) (DUMMY). This means that \( R_{16} \) should not be used by the programmer for any other purpose.
The three-bit QP increment field is the key to the queue machine execution mode. It specifies a number in the range 0 to 7 (inclusive). This number specifies how many words are removed from the front of the operand queue. The queue pointer is adjusted after the source operands have been obtained, but before the result destinations are determined. The QP increment also specifies how many registers’ presence bits are cleared. For example, a QP increment of two means that two words are removed from the operand queue. Thus, R2 becomes R0 after the instruction and the presence bits of R0 and R1 (which become R14 and R15, respectively) are cleared.

There are two problems with the basic instruction format. First, the instructions are limited to an operand-store fan-out of two. Second, the destination offset into the operand queue must be less than 16. To solve both of these problems, a second instruction format is used as shown in Figure 5.7. Instructions in this format have no source fields but two destination offsets. There are two instructions with this format, dup1 and dup2, having one and two destinations respectively. The dup instructions merely duplicate the result of the previous instruction. The result is stored in the operand queue at the word offset from the queue pointer specified by the destination fields. Note that offsets less than 16 also cause the result to be stored in the memory resident operand queue, not into the corresponding window register.

Since the result of an instruction is part of the internal state of the processing element, it must be guaranteed that an interrupt does not occur immediately preceding a dup instruction. To ensure that this does not happen, both instruction formats have a continue flag which, when set, indicates that the next instruction requires the result of the current instruction. In effect, the continue flag suspends interrupts for the duration of an instruction and all subsequent dup instructions so that they may execute atomically. Note that the last dup instruction in a sequence does not have its continue bit set.

5.28.4. Assembly language syntax

Before discussing the details of the instruction set, the syntax of the assembly language will be described. The assembly language instructions have the following format:

```
opcode[\{ \} + n] [src1 [, src2] ] [: dst1 [, dst2] ] [ >]
```

where [x] means an optional x, \{x\} means one or more occurrences of x, and x|y means an occurrence of either x or y (but not both). The valid opcode mnemonics are listed in Table 5.2 of the following section. The QP increment is specified by appending the appropriate number of + symbols to the opcode. The short form +n where n is a number is also supported.

The source operands are either registers or immediate constants. A register is specified as \texttt{rn} where n is the number of the register. Registers can also be referred to by name as shown in Figure 5.2. Immediate constants are specified as \#n where n is a (signed) decimal integer.
Destinations are specified as $rn$ where $n$ is a register number. For basic format instructions, $0 \leq n \leq 31$. For dup instructions, $0 \leq n < 256$ (the maximum size of a queue page).

Finally, the symbol $>$ may follow an instruction to indicate that the continue flag is set. This means that the next instruction uses the result of this one.

For example, the following instruction sequence removes two elements from the front of the operand queue, computes their sum, and stores the result at offsets 0, 2, and 30 from the front of the resulting operand queue:

\begin{verbatim}
plus++ r0,r1 :r0,r2 >
dup1 :r30
\end{verbatim}

5.28.5. Instruction Set

The instruction set of the proposed queue machine processing element is summarized in Table 5.2. The instructions have been divided into eight classes as indicated by the first octal digit of the opcode. The classes are: duplicate operations ($0_8$), memory operations ($1_8$), logical operations ($2_8$), arithmetic operations ($3_8$), two’s-complement comparison operations ($4_8$), unsigned comparison operations ($5_8$), branch operations ($6_8$), and trap operations ($7_8$).

As explained in the preceding section, the duplicate operations have a special instruction format. All other operations use the basic instruction format. The duplicate instruction opcodes explicitly encode the number of destinations in which the result is to be stored. This is done to eliminate unnecessary memory write operations.

The memory operations are essentially fetches and stores. The least significant three bits of the opcode are interpreted as follows: Bit 0 specifies whether the operation is a memory or channel operation. Bit 1 specifies whether the operation is a word or byte operation. Bit 2 specifies whether it is a read or write operation.

The instructions starting with octal digit $2_8$ are the logical operations. The two shift operations are arithmetic shifts. In particular, the right shift operation causes sign extension of the result. The unary bitwise complement is conspicuous by its absence. Nevertheless, it can be achieved by using the exclusive or operation with a small immediate constant of $-1$. For example, the following instruction computes the bitwise complement of $R_0$:

\begin{verbatim}
xor r0,#-1 :r0
\end{verbatim}

Only two arithmetic operations have been included in the instruction set — plus and minus. However, there is room for adding multiplication and division if needed. The unary arithmetic negation operation (two’s-complement) is also absent. It can be achieved by subtraction from the immediate constant 0. For example, the following instruction negates the contents of $R_0$:

\begin{verbatim}
minus #0,r0 :r0
\end{verbatim}

Register-to-register transfers (move operations) can be accomplished by using (among other instructions) the plus operation with a small immediate constant of 0. For example, to transfer $R_0$ to $R_1$, the following instruction can be used:

\begin{verbatim}
plus r0,#0 :r1
\end{verbatim}

Similar constructions can be used to implement register clear, increment, and decrement operations.

The comparison operations produce a Boolean result. In all cases, this Boolean result is determined the same way. $src_2$ is subtracted from $src_1$ and the usual condition code $C$ (carry), $N$ (negative), $V$ (overflow), and $Z$ (zero) are computed. The opcodes of the comparison operations
<table>
<thead>
<tr>
<th>Operation</th>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>duplicate</td>
<td>dup1</td>
<td>00₈</td>
<td>result ← previous result</td>
</tr>
<tr>
<td>duplicate</td>
<td>dup2</td>
<td>04₈</td>
<td>result ← previous result</td>
</tr>
<tr>
<td>channel send</td>
<td>send</td>
<td>10₈</td>
<td>src₁ ! src₂</td>
</tr>
<tr>
<td>store word</td>
<td>store</td>
<td>11₈</td>
<td>memory[src₁] ← src₂</td>
</tr>
<tr>
<td>store byte</td>
<td>storb</td>
<td>13₈</td>
<td>memory[byte src₁] ← src₂</td>
</tr>
<tr>
<td>channel receive</td>
<td>recv</td>
<td>14₈</td>
<td>src₁ ? result</td>
</tr>
<tr>
<td>fetch word</td>
<td>fetch</td>
<td>15₈</td>
<td>result ← memory[src₁]</td>
</tr>
<tr>
<td>fetch byte</td>
<td>fchb</td>
<td>17₈</td>
<td>result ← memory[byte src₁]</td>
</tr>
<tr>
<td>bitwise or</td>
<td>or</td>
<td>20₈</td>
<td>result ← src₁ ∨ src₂</td>
</tr>
<tr>
<td>bitwise and</td>
<td>and</td>
<td>21₈</td>
<td>result ← src₁ ∧ src₂</td>
</tr>
<tr>
<td>bitwise exclusive or</td>
<td>xor</td>
<td>22₈</td>
<td>result ← src₁ ⊕ src₂</td>
</tr>
<tr>
<td>left shift</td>
<td>lshift</td>
<td>23₈</td>
<td>result ← src₁ &lt;&lt; src₂</td>
</tr>
<tr>
<td>right shift</td>
<td>rshift</td>
<td>24₈</td>
<td>result ← src₁ &gt;&gt; src₂</td>
</tr>
<tr>
<td>plus</td>
<td>plus</td>
<td>30₈</td>
<td>result ← src₁ + src₂</td>
</tr>
<tr>
<td>minus</td>
<td>minus</td>
<td>31₈</td>
<td>result ← src₁ − src₂</td>
</tr>
<tr>
<td>≥ (twos-complement)</td>
<td>ge</td>
<td>41₈</td>
<td>result ← src₁ ≥ src₂</td>
</tr>
<tr>
<td>≠</td>
<td>ne</td>
<td>42₈</td>
<td>result ← src₁ ≠ src₂</td>
</tr>
<tr>
<td>&gt; (twos-complement)</td>
<td>gt</td>
<td>43₈</td>
<td>result ← src₁ &gt; src₂</td>
</tr>
<tr>
<td>&lt; (twos-complement)</td>
<td>lt</td>
<td>45₈</td>
<td>result ← src₁ &lt; src₂</td>
</tr>
<tr>
<td>=</td>
<td>eq</td>
<td>46₈</td>
<td>result ← src₁ = src₂</td>
</tr>
<tr>
<td>≤ (twos-complement)</td>
<td>le</td>
<td>47₈</td>
<td>result ← src₁ ≤ src₂</td>
</tr>
<tr>
<td>≥ (unsigned)</td>
<td>his</td>
<td>50₈</td>
<td>result ← src₁ ≥ src₂</td>
</tr>
<tr>
<td>&gt; (unsigned)</td>
<td>hi</td>
<td>52₈</td>
<td>result ← src₁ &gt; src₂</td>
</tr>
<tr>
<td>&lt; (unsigned)</td>
<td>lo</td>
<td>54₈</td>
<td>result ← src₁ &lt; src₂</td>
</tr>
<tr>
<td>≤ (unsigned)</td>
<td>los</td>
<td>56₈</td>
<td>result ← src₁ ≤ src₂</td>
</tr>
<tr>
<td>branch if true</td>
<td>bne</td>
<td>62₈</td>
<td>see text</td>
</tr>
<tr>
<td>branch if false</td>
<td>beq</td>
<td>66₈</td>
<td>see text</td>
</tr>
<tr>
<td>fast trap</td>
<td>ftrap</td>
<td>70₈</td>
<td>see text</td>
</tr>
<tr>
<td>trap</td>
<td>trap</td>
<td>71₈</td>
<td>see text</td>
</tr>
<tr>
<td>return from fast trap</td>
<td>fret</td>
<td>74₈</td>
<td>see text</td>
</tr>
<tr>
<td>return from trap</td>
<td>rett</td>
<td>75₈</td>
<td>see text</td>
</tr>
</tbody>
</table>

Table 5.2.
Queue machine processing element instruction set.

are designed so that the result of the instruction is true if and only if the following is true:

\[ b₂ = (b₃N) ∨ (b₁NZ) ∨ (b₀NjV) \]

where \( b_i \) is the \( i^{th} \) bit of the opcode. Note that the queue machine does not have a condition code register. This is because all data-flow actors are purely functional and have no side-effects. Instead, the results of comparison operations are always returned to the operand queue.

The branch instructions are two-operand instructions. They have the same format as all other basic instructions. The first operand is the controlling operand which determines whether the branch is taken. The second operand is the offset added to the program counter if the branch
is taken. The branch offset is represented in the same way as a source operand. The branch instructions are included to support the conventional execution model. They are not strictly required to support the indexed queue machine execution model.

Finally, the trap operations are used to provide operating system support, to enable context switching, and to provide a mechanism for emulating unimplemented instructions such as multiplication and division. The trap and ftrap operations take a single operand which specifies the word address of a trap vector. These instructions first save the processor state in the current queue page and then begin executing instructions from the address specified in the selected vector. The trap instruction rolls out the window registers and saves the global registers as shown in Figure 5.8.(a). The ftrap (fast trap) does not roll out the window registers and it only saves the program counter as shown in Figure 5.8.(b). Note that, on a trap, the global registers are stored in the last 16 memory locations of the current queue page. This means that the effective size of the queue page is 16 less than its actual size. Note also that in order to save the entire processor state, the minimum queue page is 32 words. The rett (return from trap) and frett (return from fast trap) instructions are used to return from the trap handler routines. The rett instruction restores all the global registers and the frett instruction only restores the PC. Neither routine restores the window registers. These registers are restored on demand via the presence bit mechanism.

5.29. Queue Machine Processing Element — Architecture

In the preceding section, the programmer’s model of the proposed queue machine processing element was described. In this section, an architecture for implementing the queue machine processing element is proposed. In particular, the timing of instructions and instruction cycle usage as well as the data path architecture will be described.

The goal in the design of the timing of instructions is to create a scheme in which the simplest and most common instruction type, namely operations whose source operands and result destinations are all registers, can be executed at a rate of one instruction per processor cycle. In order to accomplish this, a three-stage pipelined architecture, loosely based on that used in the Dorado [Pier 1983], was chosen. Figure 5.9 shows the elements of the basic instruction cycle sequence and how consecutive instructions overlap. The basic instruction cycle sequence consists of four parts — instruction fetch (IF), register fetch (RF), execute (E), and register store (RS). The instruction fetch takes one cycle to complete. When a sequence of simple instructions is executed, instructions are fetched at the rate of one per cycle. During the first half of every cycle, operands are retrieved from the registers for a particular instruction. During the second half of every cycle, the results of the preceding instruction are returned to the registers. The technique of register bypassing is used when the result of one instruction is an operand of the next. That is, when the output of one instruction is used by the next, the output from the arithmetic/logic unit (ALU) is fed back into its input, bypassing the registers.

Not all instructions can be completed in three cycles. More cycles are needed when an instruction requires an immediate operand and when, as the result of the presence bit of a window register being clear, an operand has to be retrieved from the operand queue in memory. In addition, operations directly involved with memory access such as fetches and stores, as well as duplicate instructions, require more cycles. The instruction cycle usage for various instruction classes is shown in Figure 5.10.

a) Instructions of this class have all their operands available in registers. That is, all the window register operands have their presence bits set.

b) Instructions of this class have one immediate word operand. An operand fetch (OF) cycle is inserted in the instruction cycle immediately after the instruction fetch.
Figure 5.8.
Contents of the queue page after \textit{trap} (a) and \textit{ftrap} (b) instructions.

\begin{table}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{cycle number} & 1 & 2 & 3 & 4 & 5 \\
\hline
\textbf{instruction} & IF & RF & E & RS &  \\
\hline
& IF & RF & E & RS &  \\
\hline
& IF & RF & E & RS &  \\
\hline
\end{tabular}
\end{table}

Figure 5.9.
Three-stage instruction pipelining (simple instructions).

c) Instructions of this class require two immediate word operands. Thus, two operand fetch cycles are required before execution can proceed.
d) Instructions of this class have the property that one of the window register operands has a clear presence bit. Thus, the operand must be fetched from the operand queue. After an idle cycle (∅), a queue fetch (QF) cycle is performed.

e) Instructions of this class require two window register operands, neither of which has its presence bit set. Thus, two queue fetch cycles must be performed before execution can proceed.

f) Instructions of this class require both an immediate word operand fetch cycle and a queue fetch cycle.

g,h) These two classes correspond to the \textit{dup1} and \textit{dup2} duplicate instructions. These instructions perform one or two (respectively) queue store (QS) cycles in which the result of the
preceding instruction is stored in the operand queue.

i. j) These two classes correspond to memory fetch (MF) and memory store (MS) operands (respectively). The figure shows the case in which the address (and the data in the case of the store) is available in a register. Immediate word operand fetch and queue fetch cycles can be inserted where needed.

The architecture of the data paths of a queue machine processing element that can execute instructions as described above is shown in Figure 5.11. The key element of this architecture is a dual-ported register bank. The register bank has the capability to support simultaneous reads or writes to a pair of registers. The registers are always read during the first half of the cycle and written during the second.

Figure 5.11.
Queue machine processing element data path architecture.

The execution phase of instructions is performed by the arithmetic/logic unit (ALU). In order to support the instruction set described in the preceding section, the ALU must provide the
following operations: addition, subtraction, bitwise or, bitwise and, left and right bit shifting, identity $X$ (i.e. $Z \leftarrow X$), and identity $Y$ (i.e. $Z \leftarrow Y$). Since the ALU execution cycle is a half cycle out-of-phase with respect to the basic system cycle, its inputs, outputs, and control signals are latched in registers $X$, $Y$, $Z$, and $ALUOP$. The register $PC2$ is used to latch instruction addresses to provide a means to abort and restart instructions when memory or channel operations fail.

Part of the circuitry required to decode the queue pointer (QP) and instructions is shown in Figure 5.12. In particular, this figure shows how queue addresses and register operands are selected. Two instruction registers, $IR_1$ and $IR_2$, are used to support the overlapped execution due to pipelining. The purpose of the second instruction register is to detect the situation where the result of one instruction is used by the next. This is necessary to allow the bypassing of the register bank. Four 5-bit comparators are required to check all source operand and result destination pairs.

The register operands are selected as follows: During the register fetch (RF) half of the basic cycle, the appropriate bits of $IR_1$ are used to select the desired source operand registers. If the register is a window register, then the virtual-register-number-to-physical-register-number translation is performed using the 4 bit adders to add the appropriate bits of the queue pointer. During the register store (RS) half of the basic cycle, the appropriate bits of $IR_2$ are used to select the result destination registers in exactly the same way. In order to support the trap and return from trap instructions, a counter, $STEP$, is used to step through the register numbers as the registers are saved or restored.

Queue addresses are determined as follows: An 8-bit adder is used to compute the sum of the queue page offset and the appropriate result destination offset obtained from $IR_1$. There are four possibilities corresponding to each of the two destination fields in each of the two instruction formats. The $POM$ register is used to select either the bits of the sum or the page offset field of the $QP$ as described earlier.

5.30. Implementing Channels

In the preceding chapter, channels were described as abstract entities providing an unbuffered, simplex communication link between two contexts. In this section, the queue machine processing element/channel interface will be described. In addition, an implementation strategy based on the use of dedicated message processing hardware will be presented.

The proposed queue machine processing element/channel interface is a generalization of the conventional processor/memory interface. In addition to the usual memory operations, fetch and store, this interface also provides the two channel operations, receive and send. This generalization of the processor/memory interface is loosely based on the $fetch$ – $and$ – $\phi$ operations proposed for the NYU Ultracomputer [Edler 1985]. This class of operations is simply a special case of a read-modify-write operation.

Figure 5.13 shows the components of the queue machine processing element/memory interface. This interface supports four operations — memory read (fetch), memory write (store), channel read (receive), and channel write (send). The operation is specified by the $read/write$ and $memory/channel$ lines. An $address/channel$ identifier bus specifies the channel identifier for channel operations and the memory address for memory operations. Memory operations are further divided into byte and word operations as specified by the $byte/word$ line. The $data$ bus is used to transmit the data between the queue machine processing element and the appropriate channel or memory location.

Two acknowledgement lines are used to signal the completion of the memory or channel operation. The $ACK$ line indicates successful completion of the requested operation. The $NAK$
Figure 5.12.
Decoding $QP$, $IR_1$, and $IR_2$ to select window register operands and queue addresses.

line (negative acknowledge) indicates that the requested operation could not be completed. In the case of memory operations, the NAK signal usually means that an illegal operation was attempted. For example, it could mean that a read or write was attempted to a non-existent memory location, or it could mean that a write operation was attempted on a read-only address. In the
In both cases, the NAK signal causes the queue machine processing element to abort the instruction and to execute a hardware trap, called a NAK trap, to an appropriate trap handling routine. A NAK trap differs from a regular trap in that it saves the address of the current instruction (which is latched in \( PC_2 \)), rather than the address of the next instruction, so that the aborted instruction can be restarted. In addition, the offending memory address or channel identifier is latched in the NAK address register (\( NAR \)). The trap handler may then perform a context switch if other contexts are available for execution or it may simply return from the trap. Upon returning from a NAK trap, the aborted instruction is restarted. Note that a simple spin-lock can be implemented by using a NAK trap handler that consists of the single instruction \texttt{rett} (return from trap).

An implementation of the memory system need not provide any special hardware to support the channel operations. The queue machine processing element/memory interface can be configured in such a way that channel operations immediately cause a NAK trap. In this way, the channels can be implemented entirely in software. However, the queue machine programming paradigm relies heavily on intercontext communication via channels. For this reason, external message processing hardware is needed to support channel operations efficiently.

There are various ways of organizing the hardware support for message processing. For example, an approach to providing message-based communications based on the use of a special-purpose coprocessor is used in the SYLVAN system architecture [Burkowski 1984]. In this scheme, a micro-coded instruction processor provides three primitives: \texttt{send}, \texttt{receive}, and \texttt{reply}. A message processor is tightly coupled with each processing element. An alternative approach is to use autonomous message processors as described in [Reghbati 1979]. In this scheme, message processors and processing elements are physically separate and are connected by a network (a cross-bar). The message processors provide two primitives: \texttt{send} and \texttt{receive}. Both of these schemes support variable-size messages with buffering. Since OCCAM channels transmit fixed-size messages and are unbuffered, a much simpler approach can be used.
In this thesis, an approach to implementing channels in hardware based on associating a message processor and message cache with each queue machine processing element is proposed. The goal in the design of this approach is to provide an easily expanded symmetric scheme in which messages can be transmitted between a pair of contexts, whether they execute on a single processing element or on two different processing elements. In addition, the approach should minimize the utilization of the interprocessor communication bandwidth.

The proposed approach to implementing channels uses dedicated hardware as shown in Figure 5.14. The hardware consists of a message processor and message cache which is placed between the queue machine processing element and the interprocessor communications medium, in this case a global bus.

![Figure 5.14. An approach to implementing channels using dedicated message processor and cache hardware.](image)

The send and receive channel operations are handled by the message processor. The message processors respond to channel operations initiated by the queue machine processing...
elements with appropriate acknowledgement signals. The message processors effect the channel transfers by using the system bus.

Each message cache has an entry reserved for every channel in the system. The message cache entry number is directly specified by the channel identifier. The format of a message cache entry is shown in Figure 5.15.

![message cache entry format](image)

**Figure 5.15.**

Message cache entry format.

Each entry consists of a one-word message field and a state field. The state field specifies the state of the channel with respect to the local queue machine processing element. The state field for a given channel is not necessarily the same in all message caches. A cache entry can be in one of the following five states:

1. **idle**: No transaction has been initiated on the channel.
2. **send pending**: A context has initiated the send half of a channel transfer.
3. **recv pending**: A context has initiated the receive half of a channel transfer.
4. **send ack pending**: A channel transfer has been completed. The sending context has not yet received acknowledgement.
5. **recv ack pending**: A channel transfer has been completed. The receiving context has not yet received acknowledgement.

The message processors effect channel transfers by communicating with each other in a broadcast mode over the system bus. In addition to memory reads and writes, the system bus supports two channel operations called \textit{fetch} – \textit{and} – \( \phi_1 \) and \textit{fetch} – \textit{and} – \( \phi_2 \). The message cache entry state transition tables for these two operations are shown in Table 5.3.

A message processor issues the \textit{fetch} – \textit{and} – \( \phi_1 \) operation to effect a channel send operation. In response to this operation, each message processor examines its message cache entry corresponding to the specified channel and performs the transition indicated in Table 5.3. If the cache entry is in state 3 (recv pending), the message is stored in the cache and a positive acknowledgement is returned to the message processor issuing the \textit{fetch} – \textit{and} – \( \phi_1 \) operation.

A message processor issues the \textit{fetch} – \textit{and} – \( \phi_2 \) operation to effect a channel receive operation. In response to this operation, each message processor examines its message cache entry corresponding to the specified channel and performs the transition indicated in Table 5.3. If the cache entry is in state 2 (send pending), the message is read from the cache and returned along with a positive acknowledgement to the message processor issuing the \textit{fetch} – \textit{and} – \( \phi_2 \) operation.

In both cases above, at most one message processor will respond to the \textit{fetch} – \textit{and} – \( \phi \) operation. This is because the semantics of channel operations ensures that only one context may send on a given channel and only one context may receive on a given channel. Those actions in Table 5.3 marked \textit{error} are conditions that should never occur. They result when two different contexts attempt to send or receive on the same channel, which violates the channel semantics. In addition, Table 5.3 shows that in response to a successful \textit{fetch} – \textit{and} – \( \phi \) operation, a message
processor may (but need not) interrupt its attached queue machine processing element to signal that a transfer has been completed.

The manner in which the \textit{fetch – and – φ} operations are used to effect the send and receive channel transfers is specified in Table 5.4. These tables show the cache entry state transitions that the message processor performs in response to send and receive operations issued by the attached queue machine processing element.

All possible state sequences that can result when two contexts executing on different processing elements perform a channel transfer are shown in Figure 5.16. In this figure, context A is sending a message to context B. The state of the channel transfer is represented as an ordered pair of message cache entry states. The first element of the pair corresponds to the state of the message cache entry in the message processor corresponding to the queue machine processing element executing context A. The second corresponds to the message cache entry state of the processing element executing B.

For example, a possible state sequence is as follows: The system begins in state (1, 1). First, context A executes a \textit{send} instruction on a given channel. Its message processor invokes a \textit{fetch – and – φ} operation, which returns a negative acknowledgement because no context has yet executed a receive instruction on the same channel. As a result, context A aborts the send
Table 5.4.
Message cache entry state transition tables for send and receive operations.

<table>
<thead>
<tr>
<th>send(chan, mesg, ack)</th>
<th>state number</th>
<th>state name</th>
<th>action</th>
<th>ack</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>idle</td>
<td>fetch − and − (\phi_1(chan, mesg, ack))</td>
<td>NAK</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>send pending</td>
<td>—</td>
<td>NAK</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>recv pending</td>
<td>cache(chan) ← mesg (interrupt)</td>
<td>ACK</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>send ack pending</td>
<td>—</td>
<td>ACK</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>recv ack pending</td>
<td>—</td>
<td>NAK</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>receive(chan, mesg, ack)</th>
<th>state number</th>
<th>state name</th>
<th>action</th>
<th>ack</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>idle</td>
<td>fetch − and − (\phi_2(chan, mesg, ack))</td>
<td>NAK</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>send pending</td>
<td>—</td>
<td>NAK</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>recv pending</td>
<td>mesg ← cache(chan) (interrupt)</td>
<td>ACK</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>send ack pending</td>
<td>—</td>
<td>NAK</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>recv ack pending</td>
<td>mesg ← cache(chan) (interrupt)</td>
<td>ACK</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

instruction. The system enters state (2, 1), which indicates that a valid message is stored in the cache associated with A. Then, context B executes a recv instruction. Its message processor invokes a fetch − and − \(\phi_2\) operation, which returns with the message and a positive acknowledgement. The system enters state (4, 1), which indicates that the transfer has completed and that the sending context should receive an acknowledgement. Finally, context A restarts the aborted send instruction which this time receives a positive acknowledgement. The system returns to state (1, 1).

All possible state sequences that can result when two contexts executing on the same processing element perform a channel transfer are shown in Figure 5.17. In this figure, the state of
the channel transfer is represented by the singleton consisting of the state of the message cache entry of the associated message processor.

An important property of this scheme is its efficient use of interprocessor communication bandwidth. In particular, a channel transfer between two processing elements involves exactly two system bus transfers, and a channel transfer between two contexts on the same processing element involves exactly one system bus transfer. All spin-locks or channel polling loops are handled by the message processor and do not place any additional load on the interprocessor communications medium.

5.31. System Architecture

In this section, a multiprocessor system architecture utilizing queue machine processing elements as the basic building block is proposed. This architecture was selected because it supports shared memory accesses as well as the channel implementation scheme described in the preceding section. In addition, the architecture attempts to address the bandwidth limitations due to a single shared bus implementation.

The basis of the system architecture is the P-bus concept [Loucks 1980] and [Rose 1985]. A P-bus is essentially a partitioned shared bus that is configured in a ring topology. Figure 5.18 shows a possible configuration of a 4-processor, 2-partition system architecture. The queue machine processing element (QMPE) and message processor (MP) are configured as described in

---

Figure 5.16.
State transition diagram for interprocessor channel transfers.
the preceding section. In addition, each bus partition has associated with it a node controller (NC).

The node controller arbitrates access to the associated bus partition. As bus master, it can grant access to the bus partition to an attached processing element or message processor. If a memory access is requested to a memory address on a different partition of the bus, the request is transferred to the next downstream node controller. The node controllers forward such requests until they reach the desired bus segment. The memory access is performed, and a response travels back around the ring to the issuing processor (QMPE or MP). A processor can have at most one request outstanding at any time. This guarantees that all processors have equal access to all memories. Furthermore, since the node controller is the master of its bus partition, a request is guaranteed to circulate (i.e., it cannot be indefinitely postponed).

In addition to memory references, the ring is also used to handle the channel operations $fetch - \phi_1$ and $fetch - \phi_2$. Since these operations are global, they must be broadcast to all message processors in the system. Thus, requests for these operations circulate around the ring in the same way that remote memory accesses do.

Earlier work on queue machine systems restricted the number of processing elements per bus partition to one [Preiss 1984] and [Preiss 1985]. Analytical and empirical results showed that this restriction caused performance degradation when the number of processing elements was increased beyond a threshold which depended on the kinds of program being executed. This

![State transition diagram for intraprocessor channel transfers.](image)

Figure 5.17.

State transition diagram for intraprocessor channel transfers.
performance degradation is due to the increased ring delay that results as bus partitions are added to the system at the rate of one per processor. For this reason, the restriction on the number of processing elements per bus partition has been lifted. Thus, by placing several processing...
elements on a single bus partition, the total ring delay can be reduced. The optimum number of processing elements per bus segment depends on the ability of the bus partition bandwidth to handle the load offered by the attached processing elements, the distribution of the local and remote memory references, and the frequency of channel transfers. The first factor depends on the implementation, whereas the latter two depend on the kinds of computation done on the system.

5.32. Conclusions

The issues associated with the design of a practical multiprocessor system based on the indexed queue machine execution model were described in this chapter. In particular, an architecture that implements the execution paradigm described in the preceding chapter was presented in detail in this chapter.

The architecture of a queue machine processing element that implements both the conventional Von Neumann execution model and the indexed queue machine execution model was specified. This architecture uses a special set of registers, called window registers, for efficient execution of queue operations.

The basic elements of the programmer’s model of the queue machine processing element were described. These descriptions specified the basic data objects, register set, instruction format, and instruction set of the proposed processing element. In addition, the data path architecture and instruction timing specifications for the proposed processing element were described.

An approach to the implementation of channel transfers based on the use of dedicated message processor and message cache hardware was described. This approach has the property that it is symmetric and easily expandable. In addition, the approach uses interprocessor communication bandwidth efficiently.

Finally, a system architecture based on the P-bus was proposed. This architecture attempts to overcome the bandwidth limitations of a single-memory/channel bus system, while at the same time provides cost effective global broadcast and shared memory capabilities.

Chapter 6
Software Simulation and Results

6.33. Introduction

In this chapter, the simulation of the queue machine multiprocessor system developed in the previous chapter will be described. This simulation was undertaken to demonstrate the feasibility of the new ideas for data-flow architectures and compilers introduced in the preceding chapters. Among the benefits of such a simulation are the following:

- The simulation program itself constitutes a formal specification of the proposed queue machine processing element.
- The simulation provides a means to test the proposed multiprocessor system without incurring the expense of actually constructing it.
- The simulation provides the ability to examine the impact of architectural design decisions on the performance of the proposed machine.
- The simulation provides the means to test and debug the OCCAM compiler by providing an environment in which real programs can be executed.
- The simulation allows the examination of the impact of the compiler optimization algorithms on the performance of real programs.

First, the organization of the simulation program will be described. It will be shown that the organization of the simulation program mirrors the proposed queue machine multiprocessor
architecture. Furthermore, it will be argued that the portion of the simulation software that emulates the behaviour of a queue machine processing element (QMPE) constitutes a formal specification of a QMPE.

Next, the queue machine multiprocessing kernel will be described. The kernel is a collection of software routines written in queue machine assembly language that create an environment in which test programs generated by the OCCAM compiler can be executed. The kernel is responsible for the management of processes and resources.

Next, some simulation results will be presented. These results are used to characterize the behaviour of a queue machine multiprocessor system having one to eight processing elements. The observed system-throughput ratios or speed-up factors are compared with a modified derivation of Amdahl’s law that attempts to take into effect kernel overhead.

Finally, the method used to confirm the correctness of the message processor protocols will be described. This involved the implementation of a program specifically to test the protocols. This was undertaken in addition to the simulation because a simulation can only confirm the presence of errors in the protocol — it cannot prove the correctness of the protocol.

### 6.34. On the Organization of the Queue Machine Multiprocessor Simulator

In this section, the organization of the queue machine multiprocessor simulation program will be described. The simulation program was written in the Concurrent Euclid programming language [Holt 1983]. The Concurrent Euclid programming language was chosen for several reasons: First, the rigid type-checking rules and the strict control of the scope of variables greatly increases the likelihood of the correctness of the program. Second, the ability to organize the program into separate, interacting modules can be used to model the architectural modularity of the system being simulated. Finally, the concurrency features of Concurrent Euclid can be used to facilitate the simulation of a system composed of many components acting in parallel.

The simulation program was designed specifically to simulate the P-bus-based queue machine multiprocessor architecture, an example of which is shown in Figure 5.18 of the preceding chapter. The organization of the simulation program directly corresponds to the architectural organization of the queue machine multiprocessor. For example, the simulation of the queue machine processing elements together with local memory is contained in one module of the simulator called the Processing module. The Message module of the simulator is responsible for the simulation of the message handler components of the multiprocessor system. Finally, the node controllers, global memory, and the P-bus are simulated by the Node module of the simulator.

The organization of the various components of the queue machine multiprocessor simulator is shown diagrammatically in Figure 6.1. As described above, the Processing module simulates the behaviour of the queue machine processing elements and local memory. There is a separate Concurrent Euclid process for each processing element in the simulated system. Each of these processes executes the same procedure called Element. This procedure emulates the behaviour of the queue machine processing element described in the preceding chapter. The Interface module encapsulates the interface between a processing element and its local memory, message handler, or node controller. The two modules, LocalRam and LocalRom, simulate the local memory associated with each processing element.

The Message module of the simulator simulates the behaviour of the message caches and message handlers. There is a separate Concurrent Euclid process for each message handler in the simulated system. There are as many message handlers as there are processing elements, and each handler is associated with exactly one processing element. Each of the message handler processes executes the same procedure called Handler. The ChanCache module simulates the local channel cache associated with each processing element/message handler pair.
The Node module of the simulator simulates the behaviour of the node controllers, global memory, and the P-bus. There is a separate Concurrent Euclid process for each node controller in the system being simulated. There are as many node controllers as there are P-bus segments.
Each of the node controller processes executes the same procedure called Controller. The Memory module simulates the global memory associated with each segment of the P-bus. The PBus monitor simulates the P-bus itself. A monitor is required in order to synchronize the interaction among the node controller processes.

The Processing, Message, and Node modules model the various components of the multiprocessor system. Above these modules are four monitors that model the interconnection and interaction of the various components. These are modelled in the form of requests and responses. The directed arcs in Figure 6.1 indicate the direction of the flow of requests. Responses flow in the opposite direction. The SysBus monitor models the interface between processing elements and the P-bus. The ChanBus monitor models the interface between a processing element and its associated message handler. The CacheBus monitor models the interface between the P-bus and the message handlers. Finally, the Interrupt monitor models the flow of interrupts between a message handler and its associated processing element. (Rather than using the ChanBus monitor for interrupts, a separate monitor is used because the direction of the flow of requests is reversed).

In addition to the aforementioned modules, the simulation program also contains a module that is not related to the actual simulation of the queue machine multiprocessor system. This module is called the Statistics module. The purpose of the Statistics module is to observe the activities of the simulator in order to gather various performance statistics. This module contains a single process, called StatsReporter, that waits until the simulation is complete and then prints a summary of the activity of the system. Each process in the system periodically reports its activity to the Statistics module. The Mutex monitor ensures mutually exclusive access to the statistics data.

An important goal in the implementation of the simulation was the confirmation of the design of the queue machine processing element presented in the preceding chapter. This was achieved by carefully writing the procedure that simulates the processing element so that it directly corresponds to the data path architecture shown in Figure 5.11. The following Concurrent Euclid code fragment is the skeleton of the Element procedure:

```euclid
var this : aProcessorState
var next : aProcessorState

InitializeState (this)
loop
    next := BasicCycle (this)
    this := next
end loop
```

Note that a single variable is used to encapsulate the current state of a processing element. Furthermore, the next state of a processing element is determined entirely from its current state. The definition of the state of a processing element taken directly from the simulation program is shown in Figure 6.2. Note the correspondence between the elements of the state and the registers of the queue machine processing element data path architecture described in the preceding chapter.

The function BasicCycle encapsulates a representation of the data-path architecture of the queue machine processing element as well as a specification for its control logic. This function emulates the behaviour of a processing element at the register transfer level. For example, data can be transferred between various elements of the state of a simulated processor only if the corresponding data path exists in the data-path architecture. Each data path can be used to effect at most one transfer per cycle. In effect, the BasicCycle function specifies the control logic for a queue machine processing element since it describes the actions to be taken as a function of the
**pervasive type** aProcessorState =

```plaintext
record
  { register bank }
| var R : array aRegisterNumber of aWord
  { presence bits }
| var P : array aWindowRegisterNumber of Boolean
  { hardware registers }
| var IR : aWord
| var IR2 : aWord
| var MAR : aWord
| var MDR0 : aWord
| var MDR1 : aWord
| var MDR2 : aWord
| var X : aWord
| var Y : aWord
| var Z : aWord
| var PC2 : aWord
| var ALUOP : anOperationCode
| var STEP : a6BitNumber
  { memory/channel operation }
| var MEMOP : aMemOp
  { target of memory op, i.e. MDR0, MDR1, or IR }
| var TARGET : aTarget
  { control logic states }
| var VALID1 : Boolean
| var VALID2 : Boolean
| var FETCH1 : Boolean
| var FETCH2 : Boolean
| var CYCLE : Boolean
end record
```

**Figure 6.2.**

*Concurrent Euclid* program fragment defining the state of a processing element.

Current state. The rigid semantics of Concurrent Euclid functions ensure that *BasicCycle* is a function in the mathematical sense (i.e., it has no side-effects) and the tight variable scope rules ensure that *BasicCycle* is purely a function of the current state (i.e., no other variable in the simulation will affect its outcome).†

Before the simulation program can be compiled, a number of architectural parameters must be specified. The architectural parameters are:

1. The number of processing elements (*numberOfProcessors*).

†Unfortunately, Concurrent Euclid does not allow record-valued functions. For this reason, *BasicCycle* was first written as a null-valued function to ensure its semantic integrity, and then it was changed into a procedure so that its result could be used.
2. The number of P-bus segments \((\text{numberOfSegments})\).

3. The size of the local read-only memory \((\text{localRomSize})\).

4. The size of the local read/write memory \((\text{localRamSize})\).

5. The size of each global memory segment \((\text{segmentMemorySize})\).

6. The size of each channel cache \((\text{numberOfChannels})\).

In order to minimize the complexity of the simulator, the architectural parameters are all implemented as compile-time constants. Unfortunately, this means that a separate simulator must be compiled for each set of parameters.

When the simulation program is executed, it is provided with two input files. The first of these files contains a queue machine program in object-code format as generated by the queue machine assembler. This program is loaded into the local read-only memory of every processor in the simulated system. The program is loaded into read-only memory to prevent it from accidentally modifying itself. The second file is a load map (also generated by the queue machine assembler). The load map specifies the starting address of each queue machine routine. The load map is loaded into the \textit{Statistics} module of the simulator. It is used to provide detailed information on the course of the simulated execution of the queue machine program.

After the input files have been loaded, each processing element starts executing the queue machine program. When all the processing elements have completed execution, the simulation is finished. (To signal that a processing element has completed execution, it attempts to execute an illegal instruction). When the simulation is finished, the \textit{StatsReporter} process prints out a detailed break-down of the number of cycles required by each processor to execute the various routines. In addition, the \textit{StatsReporter} prints a summary of the utilization of various system resources, such as the P-bus and the message caches.

\textbf{6.35. The Queue Machine Multiprocessing Kernel}

The \textit{Queue Machine Multiprocessing Kernel} is a collection of software routines that create an environment in which queue machine programs generated by the OCCAM compiler described in Chapter 4 can be executed. The kernel is like a primitive operating system. Its purpose is to manage contexts (i.e., processes) and resources (i.e., memory and channels), and to provide software routines for operations not directly implemented in hardware (such as multiplication and division). However, the kernel does not provide any of the higher-level functionality typically associated with operating systems such as device drivers, file systems, and input/output operations.

The kernel is interrupt driven. That is, kernel routines are invoked by hardware interrupts and NAK traps as well as by software interrupts which occur when a context executes a \textit{trap} or \textit{ftrap} instruction. An important feature of the kernel is that it does not rely on a real-time clock to generate interrupts in order to effect time-slicing. Instead, a context switch can only occur whenever a kernel routine is invoked. Since the amount of computation in a context is small, context switches are relatively frequent, obviating the need for time-slicing.
6.35.1. Memory Organization

The kernel is responsible for the management of the local memory associated with each processing element. The organization of memory is shown in Figure 6.3. The local memory consists of three parts: program memory, queue page memory, and kernel list memory. The program memory is used to hold the trap vectors, kernel routines, and the user program. In the simulation, the program memory is contained entirely in read-only memory.

![Figure 6.3.
Queue machine multiprocessing kernel memory map.](image)

The queue page memory contains a pool of pages from which free pages are allocated to contexts on context creation. In the version of the kernel used in the simulations, the queue page size was fixed at 512 bytes per page. Thus, the maximum queue length is 112 words (512 bytes per page ÷ 4 bytes per word − 16 words global register state). Immediately above the local memory is the global memory. The global memory is used to hold shared data. The shared data...
consists of both kernel data and user data.

### 6.35.2. Context and Memory Management

The basic purpose of the kernel is to orchestrate the concurrent execution of contexts. The kernel provides routines for the creation and destruction of contexts and for the allocation and recovery of memory and channel resources. Since every active context requires a page of queue memory, the routines for the management of contexts and queue memory pages are merged in the kernel.

The kernel maintains four lists for the management of queue memory pages (and the associated contexts). A queue memory page can be free or it can be allocated to an active context. An active context can be in one of three states: *running* (i.e., currently executing), *ready* (i.e., able to execute but not currently running), and *suspended* (i.e., waiting for a channel transfer to complete). The possible state transitions are shown in Figure 6.4. Note that when a running context finishes execution, it releases the queue page to the free list. This is shown in the figure as the context entering the *free* state.

**Figure 6.4.**

State transition diagram for contexts.

The ready list is actually a first-in, first-out (FIFO) queue. Ready contexts are dispatched from the front of the queue. When a running context executes a context switch, it is placed at the rear of the queue. Suspended processes are placed at the rear of the ready queue as the result of an interrupt signalling the completion of a channel transfer. Finally, new contexts created as the result of a fork operation are also placed at the rear of the ready queue. As a result, round-robin scheduling is performed. In effect, all contexts have equal priority.
6.35.3. Channel Management

In addition to the management of queue memory pages, the kernel is responsible for the management of channels. Recall that according to the dynamic data-flow graph splicing execution paradigm, each context has two channels (called in and out) associated with it. These channels are allocated to a context when it is activated and released by the context when it terminates. To facilitate the management of the channels, the kernel maintains two lists: a list of free channels and a list of busy channels.

The organization of the channel address space is shown in Figure 6.5. The channel address space is split into two parts: kernel channels and user channels. The user channel part of the channel address space is used for channels explicitly declared by the programmer in the OCCAM source program. These channels are allocated statically at compile time.

![Figure 6.5. Queue machine multiprocessor kernel channel address map.](image)

The kernel channels are divided among the processing elements. They constitute the pool from which channels are allocated on context creation. Note that some channels are reserved for kernel processes. These reserved channels are at known, fixed addresses. They allow the various kernel processes executing on different processing elements to communicate.

6.35.4. Kernel Routines

The queue machine multiprocessor kernel consists of ten routines. The various kernel routine entry points and trap vector addresses are shown in Table 6.1. The purpose of each of the kernel routines will now be briefly described:

The hardware trap routine is invoked by an interrupt generated by the message handler. This interrupt occurs when a channel transfer has been completed. This routine determines the
Table 6.1.

Table of queue machine multiprocessing kernel entry points.

<table>
<thead>
<tr>
<th>trap vector address</th>
<th>trap type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0_{10}</td>
<td>hardware trap</td>
</tr>
<tr>
<td>4_{10}</td>
<td>NAK trap</td>
</tr>
<tr>
<td>8_{10}</td>
<td>context switch trap</td>
</tr>
<tr>
<td>12_{10}</td>
<td>mul trap</td>
</tr>
<tr>
<td>16_{10}</td>
<td>div trap</td>
</tr>
<tr>
<td>20_{10}</td>
<td>mod trap</td>
</tr>
<tr>
<td>24_{10}</td>
<td>ifork trap</td>
</tr>
<tr>
<td>28_{10}</td>
<td>rfork trap</td>
</tr>
<tr>
<td>32_{10}</td>
<td>end-of-context trap</td>
</tr>
<tr>
<td>36_{10}</td>
<td>halt trap</td>
</tr>
</tbody>
</table>

channel number from the message handler and moves the appropriate context from the suspended list to the ready queue.

The NAK trap routine is invoked when a channel transfer cannot be completed. The running context is moved to the suspended list and a new context is dispatched from the ready queue.

The context switch trap routine is a software trap that is used by a context that wishes to relinquish the processing element. It moves the active context to the rear of the ready queue and dispatches a new context from the front of the ready queue.

The mul, div and mod trap routines are software routines for multiplication and division. The routines implement the following (non-existent) queue machine instructions (respectively):

mul+ r0, r1: r0
div+ r0, r1: r0
mod+ r0, r1: r0

These particular forms are implemented because they are easily generated by the OCCAM compiler.

The ifork and rfork trap routines are used to create a new context. These routines implement the following (non-existent) queue machine instructions (respectively):

ifork+ r0: r26, r27
rfork+ r0: r26, r27

The argument to the fork instruction is taken to be the address of the instruction sequence for the new context. The in and out channel numbers for the newly created context are returned in registers R_{26} and R_{27} (respectively). The effect of the fork instruction is to create a new context. This new context may be created on any processing element in the system. The processing element is selected according to a least-recent-fork algorithm. That is, the new context is created in the processing element which has executed the longest amount of time without having been required to create a new context. This algorithm is very easy to implement and it has been shown to be a good algorithm for a data-flow multiprocessor [Preiss 1984].
In order to facilitate the creation of contexts on other processing elements, a number of special hardware kernel processes are used. Each processing element has a scheduler context (in its pool of active contexts) for every processing element in the system. The in and out channels of the scheduler contexts have known, fixed addresses. Any processor can create a new context on any other processor by sending an appropriate message to the scheduler running on the desired processor. The scheduler returns the in and out channel addresses of the newly created context.

The end-of-context trap routine is a software trap that is invoked by a context when it has finished execution. It causes the running context to become inactive and places its queue memory page in the free list. It also returns the context’s in and out channels to the free channel list. A new context is dispatched from the front of the ready queue.

Finally, the halt trap routine is invoked by the main context of the user program to signal that the user program has completed execution. The halt trap routine contains an illegal instruction that is detected by the simulator. When the simulator detects this instruction, it terminates the simulation.

The queue machine multiprocessing kernel was written in a PASCAL-like high-level language and hand-compiled into queue machine assembly language. The kernel consists of a total of 485 instructions. The size of the kernel object code is 3028 bytes.

6.3.6. Some Simulation Results

In this section, some of the results obtained from the queue machine multiprocessor simulator will be described. The simulation measurements that are reported here were performed in order to characterize the behaviour of a queue machine multiprocessor having one to eight processing elements. The workload in each of these experiments consisted of a single OCCAM program together with the queue machine multiprocessing kernel. The results for four different OCCAM programs will be presented below. The tasks performed by the four test programs are: 1) matrix multiplication, 2) Fast Fourier Transformation, 3) Cholesky decomposition, and 4) congruence transformation. Before presenting the results for these test programs, a simple analytical performance estimation based on Amdahl’s law will be presented.

6.3.6.1. Amdahl’s Law Revisited

The principle performance metric used in this simulation study is the system throughput ratio. According to [Sanguinetti 1986],

“When measuring a particular workload on a variety of dissimilar computers, using a ratio of the system throughput is appropriate as the measure requiring the least amount of justification.”

The system throughput ratio \( R_n \) is defined as the ratio of the throughput on a system with \( n \) processors to the throughput on a system with 1 processor. Let \( X \) be the total work to be done in some particular workload area, measured in some appropriate units. It is important to note that \( X \) depends only on the application-related functions to be performed. In particular, any operating system overhead associated with a specific machine that executes the workload is excluded. (This is important to keep in mind in interpreting the quantitative results developed below, especially when considering the notion of super-linear speed-up.) Let \( T_n \) be the total execution time on a system with \( n \) processors. Then \( R_n = \frac{(X/T_n)}{(X/T_1)} = \frac{T_1}{T_n} \). Since the quantity \( R_n \) is simply the ratio of two execution times, it is often called a speed-up factor.

To derive Amdahl’s law, it is assumed that the workload consists of two parts: 1) a sequential part that requires a fixed amount of execution time, regardless of the number of processes, and 2) a parallel part, the execution time of which is inversely proportional to the number of
Accordingly, the execution time on a system with \( n \) processors is given by 
\[ T_n = \frac{P}{n} + S, \]
where \( S \) is the time to execute the sequential part, and \( \frac{P}{n} \) is the time to execute the parallel part. Then, 
\[ R_n = \frac{1}{1 + (1/n - 1)f}, \]
where 
\[ f = \frac{P}{P + S}. \]
Note that 
\[ \lim_{n \to \infty} R_n = \frac{1}{1 - f}. \]
That is, the system throughput ratio has a finite upper bound. Also, 
\[ \frac{dR_n}{dn} \bigg|_{n=1} = f. \]
That is, the slope of the curve of \( R_n \) vs. \( n \) is \( f \leq 1 \) for \( n = 1 \). As an example, the curve of \( R_n \) vs. \( n \) for \( f = 0.93 \) is shown in Figure 6.6.

\[ \text{Figure 6.6. Amdahl’s law (} f = 0.93 \text{).} \]

The problem with the preceding formulation for the system throughput ratio, \( R_n \), is that it is assumed that the workload is independent of the number of processing elements, \( n \). In this simulation study, the workload consists of two parts: 1) an OCCAM program, and 2) the kernel. Whereas the workload due to the OCCAM program is indeed independent of \( n \), the workload due to the kernel need not be.

The total workload, \( X \), consists of the sum of the user workload, \( X^U \), and the kernel workload, \( X^K \). Similarly, the total execution time, \( T_n \), is the sum of the user execution time, \( T^U_n \), and the kernel execution time \( T^K_n \). The throughput of the system is the ratio of the user workload to the total execution time, \( \frac{X^U}{T_n} \). The contribution to the workload due to the kernel is overhead and is, therefore, excluded from the throughput calculation. Thus, 
\[ R_n = \frac{(X^U/T_n)}{(X^U/T_1)} = \frac{T_1}{T_n}, \text{ as before.} \]

Also, \( T^U_n = \frac{P^U}{n} + S^U \), as before. The kernel workload consists of a large number of calls to kernel primitives. It will be assumed that these calls can execute independently. Therefore, \( T^K_n = \frac{Y}{n} \), where \( Y \) is proportional to the total work done by all the kernel calls. It is important to note that some of the workload due to the kernel primitives running on a given processor is roughly inversely proportional to the number of processes running on that processor. Thus, it is argued that 
\[ Y = \frac{P^K}{n} + S^K. \]
Hence, 
\[ R_n = \frac{1}{1 + (1/n - 1)f + (1/n^2 - 1)g}, \]
where 
\[ f = \frac{S^K + P^U}{P^K + S^K + P^U + S^U}, \]
and 
\[ g = \frac{P^K + S^K + P^U + S^U}{P^K + S^K + P^U + S^U}. \]
This formula for \( R_n \) will be referred to as the modified Amdahl’s law.

Note that 
\[ \lim_{n \to \infty} R_n = \frac{1}{1 - (f + g)}. \]
That is, the system throughput has a finite upper bound. Also, 
\[ \frac{dR_n}{dn} \bigg|_{n=1} = f + 2g. \]
That is, the slope of the curve \( R_n \) vs. \( n \) is \( f + 2g \) for \( n = 1 \). An impor-
The important result of this derivation is that when $S^U < P^K$, $f + 2g > 1$. That is, for small values of $n$, the system throughput ratio can be greater than $n$. This situation is called super-linear speed-up. As an example, the curve $R_n$ vs. $n$ for $f = 0.63$ and $g = 0.3$ is shown in Figure 6.7.

![Modified Amdahl’s law (f = 0.63, g = 0.3).](image)

Although it may seem unlikely, this kind of super-linearity is in fact a real phenomenon. Super-linear results have very recently been reported in the literature [Sanguinetti 1986]. These results are actual measurements on a message-based multiprocessor system, the ELXSI 6400. One way to view super-linearity is that it is the manifestation of a kernel implementation that penalizes systems with a small number of processing elements and that becomes more efficient only when the number of processing elements is increased.

### 6.36.2. Matrix Multiplication

The matrix multiplication program computes the product of two $n \times n$ matrices. The OCCAM source for the matrix multiplication program is listed in Appendix A.1. The program is a straightforward implementation of matrix multiplication. A pair of nested **par** constructs is used to compute the elements of the result matrix in parallel. The matrix multiplication program was chosen because it is a simple computation that contains sufficient parallelism to be of interest. It also requires synchronized access to shared memory and contains both replicated **seq** and **par** constructs.

Some statistics on the matrix multiplication program are summarized in Table 6.2. The **source** statistic is the number of non-blank lines (including comments) in the OCCAM source program. The **partition** statistic is the number of distinct acyclic data-flow graphs generated by the compiler to represent the computation. The **object** statistic is the number of queue machine instructions (including **dup**s) needed to represent the computation. Finally, the **fan-out** statistic specifies the average fan-out of the queue machine instructions. (The fan-out is computed on the basis of non-**dup** instructions. **Dup** instructions contribute to the fan-out of their predecessor.)

<table>
<thead>
<tr>
<th>Program:</th>
<th>Matrix Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source:</td>
<td>15 lines</td>
</tr>
<tr>
<td>Partition:</td>
<td>8 graphs</td>
</tr>
<tr>
<td>Fan-out:</td>
<td>0.88</td>
</tr>
<tr>
<td>Object:</td>
<td>236 instructions</td>
</tr>
</tbody>
</table>

**Table 6.2.**

Statistics for the matrix multiplication program.
The results obtained from simulating the execution of the matrix multiplication program on a queue machine multiprocessor are shown in Figure 6.8. The number of processing elements was varied from one to eight. In all of the simulations, the number of P-bus segments was one. Note the super-linearity of the speed-up curve. This is in agreement with the super-linearity predicted by the modified Amdahl’s law.

Figure 6.8.
Graph of system throughput ratio vs. number of processors for the matrix multiplication program.

6.36.3. Fast Fourier Transform.

The Fast Fourier Transform (FFT) program efficiently computes the discrete Fourier transform of $n$ data points, where $n$ is a power of two. The OCCAM source for the FFT program is listed in Appendix A.2. The program is based on the binary recursive FFT algorithm described in [Lipson 1981].

The FFT program is an interesting one in that it illustrates how a binary-recursive algorithm can be written in OCCAM. Unfortunately, OCCAM semantics do not admit recursive procedures. This is because OCCAM strictly enforces the definition-before-use rule [INMOS 1984]. This restriction is also evidenced in certain compiler algorithms. For example, the algorithm for sequencing the input arcs to a data-flow graph requires that the data-flow graph be entirely constructed. A data-flow graph which calls itself cannot be constructed until its input arcs have been sequenced. (Note that this is a compiler implementation restriction. The dynamic data-flow graph splicing paradigm does not preclude recursion).

Fortunately, it is possible to transform binary-recursive OCCAM programs into non-recursive ones that use replicated par and seq constructs. For example, an invalid, binary-recursive OCCAM program is shown in Figure 6.9(a). The procedures Pre and Post are called before and after the recursion step (respectively). If $n$ is greater than one, two recursive calls to procedure A are executed. Otherwise, the procedure Base is executed.

The OCCAM program fragment of Figure 6.9(b) calls the procedures Pre, Base and Post with exactly the same arguments and in the same order as the program of Figure 6.9(a). In addition, the non-recursive version exhibits exactly the same degree of parallelism as the binary-recursive one does.

The non-recursive FFT program was constructed from a recursive version in exactly the same manner as above. In this case, there were no Pre or Base procedures. The Post procedure consists of $\frac{n}{2}$ FFT butterfly calculations done in parallel. The statistics on the FFT program are summarized in Table 6.3.

The results obtained from simulating the execution of the FFT program on a queue machine multiprocessor are shown in Figure 6.10. As before, the number of processors was varied from one to eight and the number of P-bus segments was fixed at one. This program also exhibits the super-linear behaviour. In addition, this program exhibits an asymptotic approach to a maximum system throughput ratio.
(a) \textbf{proc} \textbf{A} (\textbf{value} n, i) =
\begin{verbatim}
  seq
  Pre (n, i)
  if
  n > 1
  par
  A (n / 2, i)
  A (n / 2, i + (n / 2))
  true
  Base (i)
  Post (n, i)
\end{verbatim}

(b) \textbf{proc} \textbf{A} (\textbf{value} n, i) =
\begin{verbatim}
  seq
  \begin{verbatim}
  seq i = [0 for \log_2 n + 1]
  \end{verbatim}
  \begin{verbatim}
  par j = [0 for 2^i]
  \end{verbatim}
  Pre (n / 2^i, j * (n / 2^i))
  \begin{verbatim}
  par i = [0 for n]
  \end{verbatim}
  Base (i)
  \begin{verbatim}
  seq i = [0 for \log_2 n + 1]
  \end{verbatim}
  \begin{verbatim}
  par j = [0 for n / 2^i]
  \end{verbatim}
  Post (2^i, j * 2^i):
\end{verbatim}

\textbf{Figure 6.9.}
Binary Recursive OCCAM procedure (a) and the corresponding non-recursive
OCCAM procedure (b).

\begin{tabular}{|l|}
  \hline
  Program: & Fast Fourier Transform \\
  Source: & 63 lines \\
  Partition: & 17 graphs \\
  Fan-out: & 0.96 \\
  Object: & 587 instructions \\
  \hline
\end{tabular}

\textbf{Table 6.3.}
Statistics for the Fast Fourier Transform program.

\textbf{6.36.4. Cholesky Decomposition}
The Cholesky decomposition program is an algorithm for the factoring of a symmetric positive
definite matrix $A$ of order $n$ into the product $LL^T$ of a lower triangular matrix and its transpose. The OCCAM source for this program is listed in Appendix A.3. This algorithm is a
transliteration of the data-flow-based algorithm presented in [O’Leary 1985].

The Cholesky decomposition algorithm was chosen because it was specifically designed for execution on a message-based multiprocessor system. In particular, it makes extensive use of
channels to communicate values between concurrently executing processes. The computation is in effect driven by the flow of data on channels between contexts.

The statistics on the Cholesky decomposition program are shown in Table 6.4. The results obtained from simulating the execution of the Cholesky decomposition program are shown in Figure 6.11. Again, the number of processors was varied from one to eight and the number of P-bus segments was fixed at one.

6.36.5. Congruence Transformation

The congruence transformation program computes the congruence transformation of a matrix $A$. That is, given two $n \times n$ matrices $A$ and $Q$, it computes the matrix $C = QAQ^T$. The OCCAM source for the congruence transformation program is listed in Appendix A.4. This algorithm is based on an algorithm presented in [O'Leary 1985].

Like the preceding program, the congruence transformation program was also designed specifically for execution on a message-based multiprocessor. The program is altered somewhat from that presented in [O'Leary 1985]. There it was assumed that channels were non-blocking.
and have infinite buffer capacity. OCCAM channels are blocking and have zero buffer capacity. The statistics for the congruence transformation program are summarized in Table 6.5.

<table>
<thead>
<tr>
<th>Program:</th>
<th>Congruence Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source:</td>
<td>66 lines</td>
</tr>
<tr>
<td>Partition:</td>
<td>18 graphs</td>
</tr>
<tr>
<td>Fan-out:</td>
<td>0.85</td>
</tr>
<tr>
<td>Object:</td>
<td>434 instructions</td>
</tr>
</tbody>
</table>

Table 6.5.

Statistics for the congruence transformation program.

The results obtained from simulating the execution of the congruence transformation program on a queue machine are shown in Figure 6.12. Again, the number of processors was varied from one to eight, and the number of P-bus segments was fixed at one.

6.36.6. Compiler Algorithms

In Chapter 4, a number of heuristic compiler algorithms were proposed. In particular, two algorithms were described: 1) an algorithm for sequencing input arcs to a data-flow graph, and 2) an algorithm for sequencing the actors within a data-flow graph. In this section, some of the measurements made to determine the efficacy of these algorithms will be described.

The problem with determining the efficacy of these algorithms is that there is nothing with which to compare them. These algorithms are intended to generate the optimum instruction sequence (i.e., that which minimizes total execution time on a multiprocessor). However, the optimum is not known. Thus, it is not possible to determine in absolute terms how close the algorithms are to the optimum.

In order to characterize the effect of the algorithms, the following experimental procedure was adopted. It is assumed that since the algorithms generate an input arc sequence and actor sequence that is the “optimum”, then the reverse of those sequences is the worst case. Four versions of the OCCAM compiler were constructed combining the various combinations of optimum and worst-case input arc and actor sequencing. The execution time of the four test programs described in the preceding sections was determined on a system having two, four, and eight processors. The number of P-bus segments was fixed at one.

The results obtained from these simulations are summarized in Table 6.6. This table lists the speed-up factors due to the compiler algorithms. These speed-up factors were obtained by
taking the ratio of the slowest execution time to the execution time for the “optimum” programs. This metric also demonstrates the variability in the total execution time due to input-arc and actor sequencing effects. Note that in all cases, the ratio is greater than one.

<table>
<thead>
<tr>
<th>Compiler optimization speed-up factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>Cholesky decomposition</td>
</tr>
<tr>
<td>Congruence transformation</td>
</tr>
</tbody>
</table>

Table 6.6.
Table of compiler optimization speed-up factors.

6.37. Verification of the Message Processor Protocols

The message processing system is an important component of the queue machine multiprocessor architecture. The dynamic data-flow graph splicing paradigm relies heavily on the use of channels for effecting a computation. In addition, the OCCAM approach to parallel programming is based on the use of concurrent processes that communicate via channels. For these reasons, it is important that the message processor protocols correctly implement the desired channel semantics.

The message handler protocols are embodied in the queue machine multiprocessor simulation program. Various test programs have been successfully executed on the simulated system. The success of these tests is encouraging. However, a simulation can only confirm the presence of faults in the message handler protocols — it cannot prove the absence of errors in the protocols. For this reason, the protocols were tested in a different manner.

A channel provides an unbuffered, simplex communications path between exactly two contexts. The communication is accomplished using two blocking primitives, send and recv. The message handler protocols must correctly implement these semantics. That is, a channel between exactly two contexts (that execute either on the same or two distinct processing elements) must operate correctly for all possible sequences of send and recv operations. Correct operation means that for every successful send and recv pair, exactly one message is transferred, no messages are lost, and no spurious messages are created. Furthermore, the protocols must detect all illegal sequences of send and recv operations. Note that if in some program more than two contexts attempt to use the same channel, it is a program error. The protocols are not required to detect this situation. The concern is that for all semantically correct programs (i.e., ones in which exactly two contexts communicate on each channel), the message processor protocols must operate correctly.

As described in Chapter 5, channel transfers are accomplished using special dedicated message processing hardware for each processing element in the system. This hardware contains a message cache which has an entry for each channel in the system. This cache entry can be in one of five states: 1) idle, 2) send pending, 3) recv pending, 4) send ack pending, and 5) recv ack pending. Either one or two message processors will be involved in a channel transfer depending
on whether the communicating contexts execute on the same or two distinct processing elements. For this reason, there are at most $5 \times 5 = 25$ states in which a channel transfer can be at any point in the course of a channel transfer. (The state of a channel transfer involving two processors $A$ and $B$ is represented by the pair $(s_A, s_B)$, where $s_A$ is the state of the message cache of processor $A$, and $s_B$ is the state of the message cache of processor $B$). For each of these states, either processor may execute either a send or a recv operation.

A computer program was written to test the message processor protocols. This program constructs a state transition table for all accessible states. A state is accessible if it can be reached through zero or more non-error state transitions from the $(1,1)$ state. (The rules for the state transitions are given in Chapter 5). For each accessible state, the state transition table lists the next state for the following four cases: i) processor $A$ does a send, ii) processor $A$ does a recv, iii) processor $B$ does a send, and iv) processor $B$ does a recv. Each transition is checked to ensure that valid messages are not destroyed and spurious ones are not created.

<table>
<thead>
<tr>
<th>state $(s_A, s_B)$</th>
<th>A:send</th>
<th>A:recv</th>
<th>B:send</th>
<th>B:recv</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1)</td>
<td>(2,1)</td>
<td>(3,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>(1,2)</td>
<td>—</td>
<td>(1,4)</td>
<td>(1,2)</td>
<td>(1,4)</td>
</tr>
<tr>
<td>(1,3)</td>
<td>(1,5)</td>
<td>—</td>
<td>(1,5)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>(1,4)</td>
<td>—</td>
<td>(3,4)</td>
<td>(1,1)</td>
<td>(1,4)</td>
</tr>
<tr>
<td>(1,5)</td>
<td>(2,5)</td>
<td>—</td>
<td>(1,5)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(2,1)</td>
<td>(2,1)</td>
<td>(4,1)</td>
<td>—</td>
<td>(4,1)</td>
</tr>
<tr>
<td>(2,5)</td>
<td>(2,5)</td>
<td>(4,5)</td>
<td>(2,5)</td>
<td>(2,1)</td>
</tr>
<tr>
<td>(3,1)</td>
<td>(5,1)</td>
<td>(3,1)</td>
<td>(5,1)</td>
<td>—</td>
</tr>
<tr>
<td>(3,4)</td>
<td>(5,4)</td>
<td>(3,4)</td>
<td>(3,1)</td>
<td>(3,4)</td>
</tr>
<tr>
<td>(4,1)</td>
<td>(1,1)</td>
<td>(4,1)</td>
<td>—</td>
<td>(4,3)</td>
</tr>
<tr>
<td>(4,3)</td>
<td>(1,3)</td>
<td>(4,3)</td>
<td>(4,5)</td>
<td>(4,3)</td>
</tr>
<tr>
<td>(4,5)</td>
<td>(1,5)</td>
<td>(4,5)</td>
<td>(4,5)</td>
<td>(4,1)</td>
</tr>
<tr>
<td>(5,1)</td>
<td>(5,1)</td>
<td>(1,1)</td>
<td>(5,2)</td>
<td>—</td>
</tr>
<tr>
<td>(5,2)</td>
<td>(5,2)</td>
<td>(1,2)</td>
<td>(5,2)</td>
<td>(5,4)</td>
</tr>
<tr>
<td>(5,4)</td>
<td>(5,4)</td>
<td>(1,4)</td>
<td>(5,1)</td>
<td>(5,4)</td>
</tr>
</tbody>
</table>

Table 6.7.

State transition table for accessible message processor states.

The state transition table generated by the message protocols test program is shown in Table 6.7. The state transition diagram corresponding to this table is shown in Figure 6.13. Of the 25 possible system states, exactly 15 are accessible. The eight empty transition table entries correspond to erroneous operations detected by the protocols. An examination of these states reveals that they belong to two distinct classes: The first class of error occurs when both processors attempt to perform the same operation on the same channel. Since channels are simplex, this is a violation of the channel semantics. The second class of error occurs after a channel transfer has completed and one of the processors attempts to use the channel in the opposite direction. Since channels are unidirectional, this too is a violation of the channel semantics. These errors are detected by the message processor protocols. When the message processor detects these errors, it can interrupt the processing element to abort the computation. All other transitions were found to be correct. Thus, the message handler protocols implement the desired channel
semantics.

6.38. Summary
The simulation study of the queue machine multiprocessor was described in this chapter. The simulations demonstrate the feasibility of the new ideas described in the preceding chapters.

Figure 6.13.
State transition diagram for accessible message processor states.
The organization of the simulation program was described. It was shown how its organization mirrors the proposed architecture. Furthermore, it was argued that the program constitutes a formal specification for the data paths and control unit of the queue machine processing element.

The organization of the queue machine multiprocessing kernel was described in this chapter. It was shown how this kernel provides an environment in which OCCAM programs that have been compiled with the OCCAM compiler described in Chapter 4 may be executed.

The performance of a queue machine multiprocessor was characterized by simulating the execution of a number of OCCAM test programs. The principal performance metric was the system throughput ratio for a number of processors in the range one to eight. It was shown that the super-linear results are not inconsistent with a theoretical analysis based on a modified derivation of Amdahl’s law.

Finally, the method used to confirm the message processor protocols was described. This confirmation was based on an analysis of the protocols themselves rather than on the simulation results. It was shown that the message processor protocols correctly implement the required channel semantics.

Chapter 7
Conclusions

7.39. Summary

The work in this thesis consists of four parts. First, a novel execution model that uses a queue as the underlying mechanism for the manipulation of operands was developed. The motivation for the development of a queue-based execution mechanism arose from the observation that conventional data-flow architectures are all based on a circular pipeline with operand and/or instruction queueing. Two queue machine execution models were developed — simple and indexed queue machines. It was shown that a simple queue machine is as powerful as a conventional stack machine in terms of evaluating arbitrary expressions. Furthermore, it was shown that instruction sequences for a simple queue machine can be efficiently generated from an expression parse tree. An important observation is that a queue machine is better than a stack machine at utilizing a pipelined arithmetic/logic unit.

In order to evaluate arbitrary acyclic data-flow graphs, the indexed queue machine execution model was developed. It was shown that instruction sequences on an indexed queue machine are the natural execution model for a computation task expressed by an acyclic data-flow graph.

The second part of the work involved the issues associated with using the queue machine paradigm in a pseudo-static data-flow machine. The main issues are the method of partitioning a program into a number of acyclic data-flow graphs and the mechanisms for coordinating the execution of these data-flow graphs. The data-flow graph splicing mechanism was developed as a means to coordinate the execution of separate acyclic data-flow graphs. This mechanism is based on the use of special operators for intercontext communication and for context creation.

A prototype compiler for the OCCAM programming language was written. This compiler automatically partitions the source program into a collection of acyclic data-flow graphs. Various compiler algorithms were developed. These include a method for partitioning the source program, a method for generating the acyclic data-flow graphs, a method for sequencing the inputs to the data-flow graphs, and a method for generating the queue machine instruction sequence from the data-flow graph.

The third part of this thesis involved the proposal of a queue machine processing element and multiprocessor system architecture suitable for practical implementation. The data-path architecture, programmer’s model, and instruction set for a queue machine processing element
were specified. This processing element architecture was specifically designed to support the queue-based execution model, as well as the conventional, Von Neumann execution model. An interesting feature of the instruction set of this processing element is its use of virtual register numbers, rather than physical register numbers. Also, the registers have presence bits associated with them which allows the queue machine processing element to automatically execute in a register-to-register mode. A multiprocessor architecture based on the use of a segmented system bus connected in a ring topology was described. In addition, a distributed approach to providing intercontext communications based on the use of special message processors and message caches was proposed. This scheme allows two contexts to communicate on a given channel without actually knowing to which processor or processors the contexts belong. In addition, this scheme minimizes the system bus traffic needed to effect communications.

Finally, the fourth part of the work involved the implementation of a simulation of the proposed queue machine multiprocessor system. This is a detailed simulation that allows the simulated execution of queue machine programs generated by the OCCAM compiler. A queue machine multiprocessing kernel was developed in order to provide an environment in which actual OCCAM programs can be executed. This kernel provides the necessary process and resource management routines. The simulation program served a number of purposes. The most important benefit of the simulation is that it allowed the characterization of the performance characteristics of the proposed system for actual test programs. The secondary benefits of the simulation program are that it provided a means to test and debug the various program decomposition, graph generation, and optimization algorithms of the OCCAM compiler.

7.40. Recommendations for Further Study

The queue-based execution model has been shown to be an efficient method for using a pipelined arithmetic logic unit. A possible application of the queue machine model is in the design of digital signal processors. An interesting area of research would be to determine whether the queue-based execution model has any advantages over conventional general-purpose digital signal processing architectures.

Another possible application of the queue-based execution model is in a conventional, Von Neumann processor. A queue machine has the advantage of automatically using CPU registers without the need for explicitly coding their use. Conventional architectures do need a stack for subroutine nesting and parameter passing. Work needs to be done to see if the queue machine and stack machine concepts can be integrated — perhaps as a stack of queues.

A prototype compiler for a subset of the OCCAM language has been developed. However, there still remain many unresolved compiler implementation issues. The current compiler completely decomposes all loop bodies and conditionals for execution in separate contexts. In many cases, this decomposition is excessive and actually decreases computational throughput. Work needs to be done to develop a more intelligent partitioning of the source program (e.g., do not partition loop bodies that, because of some sequencing constraint, cannot be unravelled). In addition, the current compiler handles the static allocation of shared data very poorly. Work needs to be done to develop a better shared data allocation policy.

A queue machine processing element architecture has been proposed and specified in some detail. Work still remains to specify the control logic for the processing element and possibly to implement a prototype in silicon.

A multiprocessor system with a novel message-handling scheme has been proposed. This scheme provides an unbuffered communications channel between two processes that is independent of the processing element or elements upon which each of the communicating processes are running. Work needs to be done to determine the feasibility of providing buffered communications in the same way. The development of a prototype message processor would also be an
interesting project.

Finally, a number of operating-systems-related questions remain unresolved. In particular, the issues associated with disk operations and other input/output functions have not been resolved. Other issues include the support for virtual memory management and secure support for multiple users and multiple tasks.

Appendix

A.41. OCCAM Program for Matrix Multiplication

```
proc MatrixMultiply (value a [], b [], var c [], value n) =
  -- Matrix multiplication routine
  -- computes c := a * b

  par i = [0 for n]
  par j = [0 for n]
  var sum:
    seq
      sum := 0
      seq k = [0 for n]
        sum := sum + (a [(i * n) + k] * b [(k * n) + j])
      c [(i * n) + j] := sum:

  -- Main program
  def N = 3:
  var A [N * N], B [N * N], C [N * N]:
  MatrixMultiply (A, B, C, N)
```

A.42. OCCAM Program for Fast Fourier Transform

```occam
def omega.real = table [0, 0, 0, 0];
def omega.imag = table [0, 0, 0, 0]:

proc FFT (var A.real [], A.imag [], value n, logn) =
    -- FFT procedure
    proc Compute =
        -- Procedure to compute FFT by calling Butterfly procedure
        proc Butterfly (value i, j, w.real, w.imag) =
            -- Procedure to compute a single butterfly of the FFT
            -- body of Butterfly
            var x.real, x.imag, y.real, y.imag, t.real, t.imag:
            seq
                x.real := A.real [i]
                x.imag := A.imag [i]
                y.real := A.real [j]
                y.imag := A.imag [j]
                t.real := x.real - y.real
                t.imag := x.imag - y.imag
                A.real [i] := x.real + y.real
                A.imag [i] := x.imag + y.imag
                A.real [j] := (t.real * w.real) - (t.imag * w.imag)
                A.imag [j] := (t.real * w.imag) + (t.imag * w.real):
    -- body of Compute
    seq
        i = [0 for logn]
        par j = [0 for (1 << i)]
        var base, n2:
        seq
            base := j * (n >> i)
            n2 := n >> (i + 1)
            par k = [0 for n2]
            var m:
            seq
                m := k * i
                Butterfly (base + k, base + n2 + k, omega.real[m], omega.imag[m]):
    proc BitReverse (value x, var result) =
        -- Procedure to reverse the least significant logn bits of x
        -- body of BitReverse
        seq
            result := 0
            seq i = [0 for logn]
                result := result \ ((x >> i) \& 1) \& (< ((logn - 1) - i)):
```

-- body of FFT procedure
seq
Compute
-- now order the results
par i = [0 for n]
var j:
seq
BitReverse (i, j)
if
  i < j
  var t.real, t.imag:
  seq
    t.real := A.real [i]
    t.imag := A.imag [i]
    A.real [i] := A.real [j]
    A.imag [i] := A.imag [j]
    A.real [j] := t.real
    A.imag [j] := t.imag:

-- main program
def N = 16:
def logN = 4:
var A.real [N], A.imag [N]:
FFT (A.real, A.imag, N, logN)
A.43. OCCAM Program for Cholesky Decomposition

def max.n = 3: -- maximum matrix size

proc Cholesky (var A [], value n) =
   -- This procedure factors
   -- a symmetric positive definite n × n matrix A
   -- into the product of a lower triangular matrix L
   -- and its transpose L^T.
   -- The matrix L is written back into the lower half of A and
   -- the matrix L^T is written back into the upper half of A.

proc sqrt (value x, var y) =
   -- A square root procedure goes here.
   y := x * x: -- dummy procedure

-- array of horizontal and vertical communication channels
chan h [max.n * max.n], v [max.n * max.n]:

-- body of Cholesky procedure
par
   par i = [0 for n]
   par j = [0 for n]
   var k, done, Aij:
   var index, north, south, east, west:
   seq
      index := (i * n) + j: -- index of array element A_i,j
      north := index - n: -- index of vertical channel from north
      south := index: -- index of vertical channel to south
      east := index: -- index of horizontal channel to east
      west := index - 1: -- index of horizontal channel from west
      Aij := A [index]
      k := 0
      done := false
      while not done
         seq
            if
               (k = i) and (k = j)
               seq
                  sqrt (Aij, Aij)
                  done := true
               k = j
               var an:
               seq
                  v [north] ? an: -- receive from north
                  Aij := Aij / an
                  done := true
            k = i
            var aw:
            seq
               h [west] ? aw: -- receive from west

   -- clxxiv --
Aij := Aij / aw
true
var an, aw:
seq
v [north] ? an -- receive from north
h [west] ? aw -- receive from west
Aij := Aij - (an * aw)
k := k + 1
v [south] ! Aij -- send south
h [east] ! Aij -- send east
A [index] := Aij

-- eastern sinks
par i = [0 for n]
var west:
seq
west := (i * n) + (n - 1)
seq k = [0 for i + 1]
h [west] ? any -- receive from west and discard

-- southern sinks
par j = [0 for n]
var north:
seq
north := ((n - 1) * n) + j
seq k = [0 for j + 1]
v [north] ? any: -- receive from north and discard

-- main program
var A [max.n * max.n]:
Cholesky (A, max.n)
A.44. OCCAM Program for Congruence Transformation

```occam
def max.n = 3: -- maximum matrix size

proc Cong (value Q[], A [], var C [], value n) =
    -- This procedure computes a congruence transformation of matrix A.
    -- That is, given two \( n \times n \) matrices \( A \) and \( Q \),
    -- it computes \( C = QAQ^T \).
    -- The computation is done in two steps: \( B = QA^T \)
    -- and \( C = QB^T \).

chan east [max.n * max.n]:
chan north [max.n * max.n]:
par i = [0 for n]
par j = [0 for n]

var index, from.s, from.w, right:

proc QAT (value q, a, var b) =
    -- This procedure is used to compute the product \( B = QA^T \).
    -- Specifically, this procedure computes the element \( b = B_{i,j} \),
    -- given \( q = Q_{i,j} \) and \( a = A_{i,j} \).

var aa:
seq
    aa := a
seq kk = [1 for n]
var k, s:
seq
    k := (kk + i) \ n
    if
        k = j
        s := q * aa
        true
        seq
            east [from.w] ? s
            s := s + (q * aa)
    if
        k = right
        b := s
        true
        east [index] ! s
    if
        (i \ 1) = 0
        seq
            -- even rows send first to prevent deadlock
            north [index] ! aa
            north [from.s] ? aa
        true
    var tmp:
```

- clxxvi -
seq
-- odd rows receive first to prevent deadlock
north [from.s] ? tmp
north [index] ! aa
aa := tmp:

var row, Aij, Bij, Cij, Qij:

-- body of Cong procedure
seq
row := (i * n)
index := row + j
from.w := row + ((j + (n - 1)) \ n)
from.s := (((i + 1) \ n) * n) + j
right := (j + 1) \ n
Aij := A [index]
Qij := Q [index]
QAT (Qij, Aij, Bij)
QAT (Qij, Bij, Cij)
C [index] := Cij:

-- main program
var Q [max.n * max.n]:
var A [max.n * max.n]:
var C [max.n * max.n]:
Cong (Q, A, C, max.n)

[Arvind 1978]

[Arvind 1980a]

[Arvind 1980b]


[O’Leary 1985]

[Pier 1983]

[Plas 1976]

[Preiss 1984]

[Preiss 1985]

[Reghbati 1979]

[Riganati 1984]

[Rose 1985]

[Sanguinetti 1986]

[Solomon 1980]

[Srini 1986]

[Tanenbaum 1985]

[Todd 1982]

[Treleaven 1982]

[Watson 1979]

[Watson 1982]
[Whitby-Strevens 1985]