TR 83-007

DOT: A DISTRIBUTED OPERATING SYSTEM MODEL OF A TREE-STRUCTURED MULTIPROCESSOR

by

Scott Harrison Danforth

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
1983

Approved by:

Gyula Mayó
Adviser

Reader

Richard Snodgrass
Reader

SCOTT HARRISON DANFORTH. DOT: A Distributed Operating System Model of a Tree-structured Multiprocessor (Under the direction of GYULA A. MAGO.)

Abstract

This dissertation presents DOT, a process-oriented design and simulation model for a highly parallel multiprocessor, and describes a complete associated programming system. The design methodology includes the use of layered design, abstract data types, and a process-oriented view of concurrency. Our results demonstrate that these software engineering structuring principles can be successfully applied to the design of highly parallel multiprocessors.

DOT is represented using an executable high-level language that provides support for discrete-event simulation. This allows verification and accurate simulation of the complete programming system, which is composed of three logical levels.

The top, or user level of the programming system is that of FFP (Formal Functional Programming) languages. The middle, or system support level is that of LPL, a low-level concurrent programming language used to define and implement FFP operators on the DOT architecture. The DOT design represents the lowest level of the programming system, a highly parallel tree-structured multiprocessor that directly supports the LPL and FFP languages.

During execution, user programs consisting of FFP language symbols are entered into a linear array of processing cells (the leaves of the binary tree of processors represented in the DOT design), and segments of this array that contain innermost FFP applications execute LPL programs in order to perform the required reductions. The LPL programs for a useful set of FFP primitives are given.

In addition to DOT and the overall programming system, this dissertation presents an analytic model which may be used to derive upper and lower bounds for program execution time. Predictions of the analytic model are compared with simulation results, and various design alternatives and possible extensions are examined.

ACKNOWLEDGEMENTS

I dedicate this dissertation to my mother, who always wanted a doctor in the family, to my father, who stood by me when I needed it most, and to Kasey.

I want to give special thanks to my advisor, Dr. Gyula Mago, who, in addition to guiding this work with a steady hand, will always guide my understanding of what it is to be a teacher, in the best sense of the word.

For their substantial assistance in improving the quality of this dissertation, I owe a special debt of thanks to the other members of my doctoral committee: Dr. Fred Brooks, who made me aware of the wider concerns of Computer Science education; Dr. Don Stanat, who taught my first Functional Programming course with such enthusiasm; Dr. Rick Snodgrass, who so cogently forced me to rewrite, rewrite, rewrite; and Dr. Barat Jayaraman, whose gentle but firm attitude made late revisions so painless.

I also wish to express my indebtedness to George Entenmann and David Middleton for their active, creative, and invaluable participation in our meetings with Gyula Mago.

Lastly, to the computer itself, my appreciation for the UNIX® editing, formating, graphics, and typesetting tools which supported the development and printing of this dissertation with such ease. This project was supported by the National Science Foundation under grant MCS80-04206, and by a grant from Harris Corporation.

CONTENTS

1. Introduction 1
    Historical Overview 1
    Reduction Machines 3
        GMD Machine (Berkling) 5
        AMPS (Keller) 6
        MM1 (Mago) 8
        Ring-Coupled Reduction Machines (Treleaven, Mole) 11
        Syntax Tree Machine (Tolle) 13
        Cooperating Reduction Machines (Kluge) 15
    Other Related Work 17
        Time and Storage Analysis (Koster) 17
        Message Routing (Kehs, Pargas, Presnell) 18
        Virtual Memory (Frank, Siddall) 19
    Dissertation Overview 19
        Dissertation Organization 21
        Selective Reading 22
2. Topmost Architecture Levels 24
    Introduction 24
        Architecture, Implementation, and Realization 25
        Multi-level Systems 26
        Architectural Concurrency 28
        Implementing Architectural Concurrency 29
    User Architecture -- The FFP Language 29
        Backus' Language Hierarchy 31
            Programming Languages 32
            Complete Languages 32
            Applicative Languages 33
            Closed Applicative Languages 34
            λ-Red Languages 35
            FFP Languages 36
        Observations and Examples 38
    Implementation Architecture -- The LPL Language 41
        LPL Architecture 42
            FFP-level Text Representation 43
            LPL-level Representation of FFP RA Symbols 47
            Message Subsystem 51
            Replicating LPL Contexts 51
        LPL Syntax and Semantics 55

program/endprogram 57
destination/endsegment 58
Lcell Data Movement and Arithmetic 58
Logical Comparisons and Program Control 59
fork/forkc 60
nselect/cselect 62
send/receive/endfilter 64
endsend/smanage 66
Synchronization of Program Segments 66
Synchronization of Forks and Messages 67
Synchronization of Completion 68
Remarks and FFP Operator Definitions 69
FFP Functions 72
Identity 72
Atom 74
Equals 76
Double 78
Length 81
Tail 82
Rotr 83
Distl 85
Matrix Transpose 88
N-ary Add 90
Sort 91
Matrix Multiply 92
FFP Functional Forms 95
Constant 97
Select 98
Composition 100
Construction 102
Conditional 104
Conditional -- phase 2 107
Apply-to-all 109
Element-by-element 111
Meta-composition 113
3. Implementation -- The DOT Model 115
Introduction 115
What DOT is (and what it isn't) 115
Overall DOT Structure 117
A Language for Representing DOT 117
A Process-Oriented Design 120
Communication Between Processes in Different Cells 121
Overall DOT Operation 124
The Basic Machine Cycle 124

Partitioning Phase 126
Area Nodes 128
Directory Creation 129
Loading LPL Programs 130
Execution Phase 130
Lcell Message Support 131
Fork Support 132
Storage Management Phase 132
The Specification for Storage Management 134
Overflow and Program Entry 134
Process Structuring of DOT Cells 135
Tcell Structure 135
Lcell Structure 141
IO Subsystem 148
VM Subsystem 148
Important Algorithms 150
Partitioning 151
Partitioning Upsweep -- Locating RAs 152
Merging Segment Descriptors 154
Switching to Area Channels 158
Partitioning Downsweep -- Pruning 162
Message Support 168
Message Packets 169
Message Handling 170
Stopping Messages 171
Directory Creation 174
Computation of the Symbol Index 174
Upsweep 175
Downsweep 176
Computation of the Directory Tuple 177
Upsweep 180
Downsweep 187
Calculating the Specification for Storage Management 191
Upsweep 192
Downsweep 193
Summary 199
4. Simulating DOT 201
Introduction 201
The Place of Simulation Within the Design Cycle 202
The Cost of Simulation 204
Introducing Global Time into DOT Operation 206
Discovery of a Critical Path 206
Event-Scheduled Modeling 207
Process-Interaction Modeling 207

Process-Interaction with Implicit Events 208
Implementing Implicit Events 209
Simulation Output 211
Tracing Machine Operation 212
Lcell Array Snapshots 215
Simulation Results 217
5. Analytic Performance Model 220
The Execution Cycle 221
Notation 225
Area Heights 226
Phases of RA Progress 229
Storage Management Phase -- RA Creation 229
Partitioning Phase -- RA Detection 229
Execution Phase -- RA Execution 229
Formulas for the Duration of RA Phases 230
RA Storage Management -- (SM) 231
RA Partitioning 232
Partitioning Old RA -- (PO) 233
Partitioning New RA -- (PN) 233
RA Execution -- (EX) 234
Predicted Execution Time for Single RA 237
Analysis and Simulation of ID 239
Analysis and Simulation of N-ary Add 240
Analysis and Simulation of SORT 241
Analysis and Simulation of ROTR 242
Analysis and Simulation of EE1 243
Complete Programs 244
Restrictions 247
Summary 249
6. Design Alternatives and Extensions 252
Design Alternatives 253
FFP-level Text Representation 253
LPL Message Routing 254
Non-blocking Fork 255
Completion Synchronization 257
Duration of Execution Phase 259
Shifting vs. Reloading LPL Code 264
Storage Management Transfer Function 264
Design Extensions 265
Job Control Language 266
Pushdown Storage for Lcells 268
Visual Tracing 272
LPL Storage Management Event Indicator 274
Storage Management with Variable Context Sizes 274

Increasing Phase Independence 275
Multiple Input Ports 276
7. Conclusion 277
References 281

FIGURES

FIGURE 2.1 -- Vertically Recursive Computer Architecture 26
FIGURE 2.2 -- DOT Implements LPL; DOT+LPL Implement FFP 27
FIGURE 2.3 -- DOT Compiler and Operating System Aspects 27
FIGURE 2.4 -- Inner Product of < 1 2 3 > with < 4 5 6 > 30
FIGURE 2.5 -- A Hierarchy of Programming Languages 31
FIGURE 2.6 -- Semantics of FFP Languages 37
FIGURE 2.7 -- LPL Architectural View 43
FIGURE 2.8 -- Derivation Tree for e = (< CONST 5 > 8) 44
FIGURE 2.9 -- Parse Tree for e = (< CONST 5 > 8) 45
FIGURE 2.10 -- A Linear Representation using ALNs 45
FIGURE 2.11 -- Evaluation of (+ < (ID 1) 2 >) 46
FIGURE 2.12 -- Directory Tuples for (IP << 1 2 3 >< 4 5 6 >>) 48
FIGURE 2.13 -- Illustration of Directory Tuple Definition 50
FIGURE 2.14 -- LPL Environment 52
FIGURE 2.13 -- Forking Must Wait for Storage Management 54
FIGURE 3.1 -- Overall DOT Structure 117
FIGURE 3.2 -- The DOT Machine 118
FIGURE 3.3 -- The DOT Tree of Processing Cells 119
FIGURE 3.4 -- A DOT Full Duplex Communication Channel 122
FIGURE 3.5 -- Connecting and Disconnecting Cqueues 123
FIGURE 3.6 -- The DOT Cqueue Head and Tail -- public entries 124
FIGURE 3.7 -- A Partitioned DOT Machine 128
FIGURE 3.8 -- DOT Tcell Processes 135
FIGURE 3.9 -- Tcell Output Process 136
FIGURE 3.10 -- The Tcell Input Process 137
FIGURE 3.11 -- Area Node Downwards Message Handler Process 138
FIGURE 3.12 -- Tcell Manager Process 139
FIGURE 3.13 -- Tcell Node-Manager Process 140
FIGURE 3.14 -- DOT Lcell Processes (and User Context) 141
FIGURE 3.15 -- Lcell Input Process 141
FIGURE 3.16 -- Lcell Message-up Process 143
FIGURE 3.17 -- Lcell Message-Down Process 145
FIGURE 3.18 -- Lcell LPL Interpreter Process 146
FIGURE 3.19 -- The Lcell Manager Process 147
FIGURE 3.20 -- The Main IO Process 148
FIGURE 3.21 -- The VM Overflow and Program Entry Process 148
FIGURE 3.22 -- Finding a RA 152
FIGURE 3.23 -- Sweeping Lcell Contents Upwards Globally 153

FIGURE 3.24 -- The Segment Descriptor Format 155
FIGURE 3.25 -- An Example of a Segment Descriptor 156
FIGURE 3.26 -- The Four Segment Descriptor Formats 157
FIGURE 3.27 -- Analysis of the Initial Partitioning by Tcell Managers 159
FIGURE 3.28 -- The Start of Partitioning in the Lcells 162
FIGURE 3.29 -- An Example of Initial Partitioning 163
FIGURE 3.30 -- After Pruning the Initial Partitioning Example 165
FIGURE 3.31 -- The Area Node Algorithm to Start Pruning 167
FIGURE 3.32 -- Termination of Pruning in the Lcells 168
FIGURE 3.33 -- The Four Message Packet Types 169
FIGURE 3.34 -- Upwards Messages in a Tcell Node 173
FIGURE 3.35 -- Diagram for Symbol Index Calculation 175
FIGURE 3.36 -- Algorithms for Symbol Index 177
FIGURE 3.37 -- Diagram for Directory Tuple Calculation 178
FIGURE 3.38 -- An Example RA Within an Active Area 179
FIGURE 3.39 -- Example Parse Tree and Directory Tuples 180
FIGURE 3.40 -- Before Merging Partial Parse Trees 182
FIGURE 3.41 -- After Merging Partial Parse Trees 183
FIGURE 3.42 -- Directory Upsweep Computation 185
FIGURE 3.43 -- An Example Upsweep 186
FIGURE 3.44 -- Directory Downsweep Computation 188
FIGURE 3.45 -- An Example Downsweep 189
FIGURE 3.46 -- Algorithms for Directory Tuple 190
FIGURE 3.47 -- Calculating a Specification for Storage Management 191
FIGURE 3.48 -- Transfer Value Solution for Storage Management 194
FIGURE 3.49 -- Calculation of the Transfer Function 196
FIGURE 3.50 -- Starting the Downsweep of Preparation for SM 197
FIGURE 3.51 -- Lcell Preparation and Storage Management 198
FIGURE 3.52 -- Tcell Preparation for Storage Management 199
FIGURE 4.1 -- Objectives of Simulation 202
FIGURE 4.2 -- Properties of Simulation 202
FIGURE 4.3 -- Subsidiary Benefits of Simulation 202
FIGURE 4.4 -- The DOT Event Scheduler 210
FIGURE 4.5 -- Qtail.put Implements Implicit Events 211
FIGURE 4.6 -- Filtered High-Level Trace Output 213
FIGURE 4.7 -- Unfiltered High-Level Trace Output 215
FIGURE 4.8 -- FFP Expression for Reduction 215
FIGURE 4.9 -- Sort Time v.s. Number of Messages per Cycle 218
FIGURE 5.1 -- RA Progress Through a Single Machine Cycle 225
FIGURE 5.2 -- Trees for Lower and Upper Area Height Extremes 227
FIGURE 5.3 -- Bounds on Area Height (using H=21) 227
FIGURE 5.4 -- Uniform Loading in the Presence of Empty Lcells 228
FIGURE 5.5 -- RA Progress Through Successive Machine Cycles 231
FIGURE 5.6 -- Duration of Storage Management for RA 232

FIGURE 5.7 -- Duration for Partitioning Old RA 233
FIGURE 5.8 -- Duration for Partitioning New RA 234
FIGURE 5.9 -- Duration of Time to SM_Grant, t_s 235
FIGURE 5.10 -- Duration of the Execution Phase 237
FIGURE 5.11 -- Analytic/Simulation Model Result Format 238
FIGURE 5.12 -- Analysis and Simulation of ID 240
FIGURE 5.13 -- Analysis and Simulation of N-ary Add 241
FIGURE 5.14 -- Analysis and Simulation of SORT 242
FIGURE 5.15 -- Analysis and Simulation of ROTR 243
FIGURE 5.16 -- Analysis and Simulation of EE1 244
FIGURE 5.17 -- A Program for Inner Product of <1 2 3> and <4 5 6> 245
FIGURE 5.18 -- Execution Trace for Inner Product 245
FIGURE 5.19 -- Analysis and Simulation of Inner Product 247
FIGURE 5.20 -- Restrictions on Parallel RAs for Analyzability 248
FIGURE 6.1 -- Possible Design Modifications 253
FIGURE 6.2 -- Efficiency of Fair Operators 261
FIGURE 6.3 -- Possible Variable Cycle Time Heuristics 262
FIGURE 6.4 -- User Interface Facilities 266
FIGURE 6.5 -- A User-defined Operators and its LPL Program 268
FIGURE 6.6 -- COND Copies its Argument 269
FIGURE 6.7 -- COND with Push and Pop 271
FIGURE 6.8 -- Steps to Provide Visual Tracing 272

DEFINITIONS

FP 1
FFP 1
control-flow 3
reduction 3
data-driven 4
demand-driven 4
RAs 9
microprograms 10
DOT 20
LPL 21
architecture 25
implementation 25
realization 25
representation 25
vertically recursive 26
architecturally concurrent 28
programming language 32
complete language 32
semantic function 33
applicative language 33
representation function 34
closed applicative language 34
primitive functions 37
meta-composition 37
innermost reduction semantics 38
evaluation mechanism 38
controlling operator 39
lcells 42
aln 44
directory tuple 48
symbol_index 48
rln 48
LPL environment 51
user context 52
code segments 55
message waves 64
functional form 95
tcells 117

ClassC 117
process-oriented design 120
partitioned 125
partitioning phase 125
execution phase 125
storage management phase 125
specification for storage management 126
preparation for storage management 126
active areas 126
initial partitioning 126
area channels 126
area nodes 126
cell manager channels 126
pruning downsweep 127
top of area 127
toa 127
io channels 130
end-of-wave 131
eow 131
sm_grant messages 133
stop message 133
segment descriptor 154
ground state 156
executing state 156
SPF 157
prefix packet 169
data packet 169
eow packet 170
stop packet 170
discrete event simulation 206
event-scheduled 207
process-interaction 207
PC Representation 254
fair operators 260

CHAPTER 1

Introduction

In this dissertation, we present a highly parallel multiprocessor implementation for a general-purpose functional language suggested by Backus [Bac78]. We describe the implementation and its simulation, provide an analytic model for program execution time, and finally consider design alternatives and extensions.

The first three sections of this introductory chapter provide an overview of related work from the perspectives of language and implementation design. The last section of this chapter describes the objective and overall structure of the dissertation.

1.1. Historical Overview

Historically, programming language design and implementation are often intertwined, with the implementation of new language concepts closely following their inception. We are therefore concerned with presenting two sides of the same coin in this introduction, and the twin aspects of language development and implementation relevant to this dissertation are discussed accordingly.

Backus' first publications concerning functional programming appeared in the early 1970's [Bac72, Bac73]. Then, in the 1977 ACM Turing Award lecture, he presented a new and powerful formalization for functional programming: the *FP* (Functional Programming) and *FFP* (Formal Functional Programming) languages [Bac78]. These are general-purpose high-level programming languages, capable of expressing parallel computations in a natural fashion. In

addition, associated with the FP language class is an algebra of programs that can be used to reason about and transform programs while preserving their semantics. Such a facility can be extremely useful for the verification of programs.

The FFP language class follows FP in spirit, but employs a more restricted syntax -- making it suitable for direct machine execution. Unfortunately, despite the potential of Backus' suggested approach, early implementations of FFP on traditionally structured computer architectures ran slowly. Then, in 1979, using a language-based approach, Mago gave an initial description of a tree-structured cellular network of processors oriented towards efficient parallel execution of FFP language programs [Mag79]. Many problems were raised and left unsolved in this initial description, but the feasibility of a design was demonstrated.

The language-based approach to processor design is not new; over the last decade a number of machines oriented toward direct support for high level languages have been suggested. The approach has been used for implementing sequential languages such as APL [Abr70], Symbol [Ric71], Basic [Bur78], Lisp [Ste81], Jovial [Chu81], and Pascal [Car81]. Nevertheless, although many parallel computer architectures have been recently proposed [Dav75, Des78, Den79, Sha82, Sto83] as a result of the increasing potential of VLSI and other realization technologies, few of these designs have been based on a general-purpose high-level programming language. This is due to the low-level sequential transformation of state and reliance on global memory embodied in most languages, which make it difficult to express a direct mapping between a language and its implementation on a parallel architecture.

In functional programming languages (which arose from the search for an algebra of programs [Bac81, Bac82]), sequential transformation of state and global memory are absent because of their unsatisfactory properties with respect to program semantics. As a serendipitous result, such languages are promising candidates for language-based parallel support.

An important aspect of Mago's language-based proposal is the use of *fine grain parallelism*. This approach removes assumptions of global memory and overall processor state from the language implementation as well, and completely realizes the parallelism allowed by FFP programs. The highly-parallel nature of the multiprocessor suggested by Mago is especially intriguing in view of the recognized difficulty of designing a general purpose machine that can fully exploit the potential of VLSI.

1.2. Reduction Machines

We now review various proposals for the support of functional languages. Among machines for directly executing functional languages are those suggested by Berkling [Ber75], Keller et al. [Kel79], Mago [Mag79], Treleaven [Tre80], Tolle [Tol81], and Kluge [Klu82]. These machines may be characterized as reduction machines -- i.e., they support reduction-style program execution.

In contrast with *control-flow* programs, which are built out of and executed as linear sequences of simple operations called instructions, *reduction* programs are built from nested expressions. In reduction language programs, the nearest thing to an instruction is the application of a function to an argument, but both functions and arguments may be expressions containing further function applications nested within themselves. A reduction language program is an expression -- equivalent to the result of its execution in the same way as the expression *3*(2+2)* is equivalent to the expression *12*.

Nevertheless, the machine language for Berkling's computer is a high-level functional language. This was an important contribution, and opened the way for further work.*

1.2.2. AMPS (Keller)

AMPS stands for "Applicative Multiprocessing System," and this design [Kel79] features a loosely-coupled tree structure to be composed of around 1000 nodes. The language supported is a compiled dialect of LISP called FGL (Functional Graph Language). Streams, or infinite data structures, are supported through the use of a demand-driven evaluation mechanism.

FGL represents a program as a function graph whose nodes are data forming functions (possibly user-defined with an inner sub-graph structure) and whose arcs represent access to data formed and made available by other nodes. The basic data forming operations of Lisp are primitive, and cons is the *lenient cons* suggested by Friedman and Wise [Fri76]. Independent sub-graphs called *productions* are supplied for user-defined functions, and these may be recursive in nature. When a function node requires the data produced by multiple "subordinate" nodes, it may send parallel demands to each of these nodes. Thus the FGL language can express parallelism, and the implementation supports it. AMPS is thus an excellent example of the language-based approach to multiprocessor design; creation of parallel tasks required to utilize the power of the hardware is implicit in the language. Clever compile-time analysis of program text is not required to detect opportunities for parallelism, nor is

---

* Turner's S-K reduction machine [Tur79], and the Cambridge SKIM reduction machine [Cla80] also use high-level functional languages as their machine languages. Although their machine languages are interesting (expressions are built with combinators), these designs are not discussed further because, like Berkling's design, they are single-processor implementations that cannot directly support the parallelism inherent in their machine languages.

static pre-allocation of tasks to processors necessary.

Each leaf node in the physical tree structure of AMPS contains a fairly powerful processor (on the order of a micro-computer) and an associated local memory unit with around 64K words. The interior tree nodes are used for communication and distribution of processing tasks. There is a single unified address space, and, due to the graph reduction mechanism, sharing of data structures is prevalent. Local caching is used to help alleviate contention for the primary copy of a data structure, while the applicative nature of the language guarantees read-only access and the validity of cached data.

The execution of each graph or sub-graph is bound to a single processor, so computation involves relatively large-grain processing tasks. Processing tasks are created "top-down" in response to encountering a demand for the data object produced by a function node. Task creation is therefore dynamic and unpredictable. Run-time binding of tasks to leaf processors often results in the need to "farm out" processing to some other leaf node, and although a unified address space simplifies this procedure logically, the penalty of communication over the shared tree-structure is incurred.

Circuit-switching of communication lines is infeasible due to the possibility of tying up long paths through the tree structure, so the interior nodes support a packet-switching communication protocol. The cost of this communication cannot be known ahead of time, making it impossible to predict the execution time of a program on the machine. AMPS therefore completely supports the logical parallelism inherent in its machine language (by dynamically creating processing tasks as they are required) but at unknown cost and with little additional lower level implementation parallelism aside from that implied by the language.

The issue of predictable performance is one that plagues multiprocessor designs, and has its roots in the degree to which separate processes are allowed to interfere with each other. This interference normally takes the form of contention for a shared resource.

In the case of AMPS, processes must compete for their share of processor time, cache storage, local memory, and communication bandwidth. The difficulties of analytically modeling the results arise from a lack of control over the process interference while at the same time allowing dynamic and unpredictable creation of processes. To the degree that process interference can be carefully limited, controlled or predicted, an analytic model should be able to successfully predict performance.

1.2.3. MM1 (Mago)

This section introduces the design proposed by Mago [Mag79], upon which this dissertation is based. Although various changes in orientation from his original concept have been made, high-level and common aspects will be stressed here. As in the other reviews, overall structure and implications for language support and process interference are of primary interest.

The essential problem with string reduction is how to support it efficiently. Berkling saw this, and envisioned processing in memory as a possible solution, but has not suggested a suitable design. Mago has successfully done so.

The Mago Machine, or MM1, as we shall call it, is essentially a binary tree of small-grain processors, including linear connections between adjacent leaf cells. FFP text is stored, symbol by symbol, in the leaf cells or *lcells*. The interior tree cells, called *tcells*, are used for a variety of functions, and during reduction of innermost FFP applications they support communication routing between the lcells. Thus the lcells act as a linear memory array, and the tcells are used when

global context must be accumulated and used. The design is expandable to any size, and tree machines composed of a million cells (height around 20) are envisioned.

A great deal of ingenuity is required to efficiently accumulate and use (within the overall tree structure) global contexts discovered from individual lcell contents. Initially, innermost applications (called *RAs* for ***reducible applications)*** must be discovered. This is performed in a single upsweep and downsweep of information through the overall tree structure. In this process, the machine is ***partitioned*** so that individual dedicated binary trees for communication routing are associated with each RA. These small trees are embedded in the overall tree-structured network using a circuit switching approach. Also during partitioning, the syntactic structure of each RA is determined, and the containing lcells are given information of limited but useful precision concerning their locations in the corresponding parse tree.

Following partitioning, each innermost application has its own dedicated multiprocessor and communication network embedded in the overlying tree structure for support of its reduction. This approach may be contrasted with AMPS, in which single leaf cells are tasked with the reduction of complete graphs and may interact with tasks and data in other leaf cells. In Mago's approach, more than one processor is entrusted with a single reduction, and communication is performed within dedicated (circuit switched) channels as opposed to the shared packet-switched channels in AMPS. Each active partition of MM1 is therefore able to operate on its own reduction independently of others, and lower level parallelism (beyond that implied by the FFP language) is available to further increase the efficiency of reduction. Message routing is a simple broadcast mechanism within dedicated channels.

After partitioning, the lcells must be told how to behave in order to perform the reduction. This is indicated by the operator of the reduction. The range and utility of FFP operators is great, and the potential number of useful operators for a programming system is correspondingly large. In the interest of flexibility in the operator set, and in recognition of the limited storage space that will be available within lcells, operator definitions are not stored in the lcells, but are brought in on demand at run-time. Mago called these definitions *microprograms* and they consist of a short series of instructions to the lcells of an RA on how to proceed in order to achieve the desired reduction.

The machine operates in a major cycle composed of partitioning, execution, and storage management. Storage management is required to allow FFP text to expand when the result of reducing an expression is larger than the original expression, and is performed by shifting information within the lcell array. During this process, space made available by RAs that reduce to smaller expressions can be used to make room for expressions that are growing. The shifting is performed by sending FFP text symbols along the lateral shift register connections between the leaf cells. Microprograms are interrupted for storage management asynchronously, without need for special preparation on their part, and are then automatically continued after storage management and re-partitioning.

As mentioned earlier, process interference appears to be a primary source of difficulties when predicting program execution time. In Mago's design, process interference is confined to the storage management phase of the execution cycle. This interference may be characterized as contention for the shared memory space in the lcell array. Surprisingly (in view of the other designs reviewed here) the exact character of the interference is predictable

since it is determined by the string reductions implicit in the source FFP text.

Mago's design represents a revolutionary approach to direct support for string reduction of reduction languages. His original paper describing this approach [Mag79] provides a detailed discussion of how an implementation might support each of the three phases of the machine cycle, and presents a "strawman" microprogramming language.*

1.2.4. Ring-Coupled Reduction Machines (Treleaven, Mole)

Treleaven and Mole have proposed an implementation for a multiprocessor reduction machine based on string reduction [Tre80] which also incorporates parallel support for functional languages.

The FFP reduction languages of Backus exhibit linear ordering and are well suited to string reduction. If we imagine an FFP program as a linear tape containing the successive symbols of program text, innermost applications then appear as individual, separate, and independent areas of the tape. The semantics of FFP languages guarantee that we can reduce these in any order (or even in parallel) without affecting the final answer.

Continuing the tape analogy, imagine a special "tape machine" able to move the tape back and forth, collect a portion containing an innermost application, and then splice an application result back in place of the original innermost application text. Consider two such machines, or even more, spread apart but all working on the same tape. To avoid boundary problems the tape could be connected at its ends to form a large circle. This is the essence of Treleaven's approach.

---

* Two designs of a more complete nature have been inspired by this early work: Tolle's design, discussed in a following section; and my own.

Treleaven's tape is implemented by connecting individual reduction machines similar to the above "tape machines" in a ring through the use of hardware and secondary storage deque structures. The reduction machines use the deque structures to shift FFP text through themselves, and replace innermost applications with their results whenever possible. Because of the necessity for storage and reduction of complete RAs, the reduction machines are relatively large-grain processors. This simple and conceptually pleasing design clearly shows the value of string reduction for multiprocessor systems.

Note, however, that the complete parallelism of FFP languages is not supported. If there are n reduction machines, then only n innermost applications can be performed concurrently. And this is the best case; when a large number of consecutive applications are created, they could be caught between and executed by only two reduction machines. Thus, there is limited parallelism at the language level, and (as in AMPS) no additional lower-level parallelism. Another problem is that RAs may be created that are too large to be contained within an individual reduction machine.

How about process interference? At first glance, this seems confined to "tug-of-war" on the "tape", which is easily handled with a priority mechanism or a preferred direction. Once a reduction machine has an innermost application, it will be executed without interference. There is no contention for shared communication paths, no global memory, and no caches.

Unfortunately, as hinted above, performance cannot be predicted for this design either, and for even more serious reasons than for AMPS. Here we don't know how much shifting will be necessary for an innermost application to find a reduction machine, and any number of innermost applications could be trapped between two processors. This last can be viewed as process interference, and it