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 Magó
Adviser

[illegible]
Reader

[illegible]
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

FIGURES

DEFINITIONS

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.

Treleaven has identified two basic approaches toward executing reduction programs: string reduction and graph reduction [Tre82]. The basis of string reduction is that a program is manipulated in-place; each application of an operator to an argument is textually replaced by an equivalent expression, and expressions are not shared. Commonly used expressions must be duplicated throughout the program where necessary. In graph reduction, implicitly shared references to expressions are manipulated.

Thus, for example, evaluation of $a+a$ with string reduction would require the definition of a to be loaded twice, replacing both of its references. Evaluation of the same expression with graph reduction doesn't require such replacement; instead, the expression referred to by a is itself evaluated (in a similar fashion) without modification of the addition expression, and the result is made available by reference to support the addition.

The distinction between *data-driven* and *demand-driven* computation for this example concerns whether the expression represented by a is evaluated before it is required by $a+a$ (data-driven), or after it is required (demand-driven).

As mentioned, the design to be presented in this dissertation is largely derived from the work of Mago. A description of this early work is included here for historical continuity, and to emphasize its place relative to the other attempts that have been made. Not included in this review are architecture implementations that are based on a physical tree structure but which are designed to execute other than functional languages. These include the designs suggested by Despain [Des78], Davis [Dav78], Stolfo [Sto83], and Shaw [Sha83].

For each proposed design, we will be concerned with the particular functional language that the architecture supports, and the computational model used (string vs. graph reduction, demand vs. data-driven, etc.). Program decomposition for multiprocessor architectures will be examined. This includes distribution of data processing tasks to their respective processors, as well as re-integration of results. The number of processing units and their individual capabilities are also important aspects of a design, as well as the power of language primitives. Limits to parallelism and efficiency will be scrutinized. Note that parallelism and efficiency are not the same thing, since the cost of creating or maintaining parallelism can be greater than the gain in performance.

1.2.1. GMD Machine (Berkling)

Central to the concept of reduction-style execution is the replacement of operator applications with their results. As described, the two basic computational approaches are actual textual substitution (i.e., string reduction) and the use of pointers and a global memory (i.e., graph reduction). Berkling was among the first to recognize the need for research into systems based on textual substitution for directly supporting high level reduction languages. In response to Backus' early work on reduction languages, Berkling designed a computer system implementation [Ber75] based on the concept of string reduction which supported direct execution of a variant of Backus' λ-Red (for Lambda-Reduction) languages [Bac73].

Berkling recognized the potential of the substitution approach, and envisioned the feasibility of parallel processing "in memory" without the use of a central processing unit. In his implementation, however, he chose a traditional organization, and used a central active processing component to operate on data stored in passive memory (three hardware stacks). His implementation neither supports the parallelism inherent in λ-Red languages, nor makes use of lower level parallelism within the implementation in the interest of efficiency.

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 makes it impossible to predict the interval between the implicit creation of a processing task and its actual execution. Nevertheless, Treleaven's approach is very useful conceptually; it clearly shows both the important benefits and problems associated with string reduction.

Mago's suggested design predates Treleaven's effort, and its fruition in the implementation we will soon describe can be viewed as an attempt to circumvent the above problems of string reduction, while maintaining all of the benefits. The extent to which we have been successful will be examined later, but our approach (as suggested by Mago) can be viewed in the following way with respect to Treleaven's: instead of moving the text symbols on shift register deque structures between intelligent reduction machines, make the individual shift register components intelligent enough to perform their own reductions and their own splicing. Instead of requiring movement of the "tape" though a single device in order to sequentially accumulate the global context required to identify innermost applications, use an overlying tree-structure to perform this process in parallel. The ultimate result is that innermost applications never have to wait for processing power, which solves the scheduling problem experienced by Treleaven's design.

1.2.5. Syntax Tree Machine (Tolle)

Tolle has proposed a design [Tol81] inspired by and in some ways similar to the original proposal of Mago. Tolle also uses a binary tree of processing elements to accomplish string reduction of FFP text stored within the leaves of the tree. Where Mago proposes strict limitations on the capabilities of the interior tree cells, however, Tolle investigates the potential of giving them a greatly increased and programmable flexibility.

In response to FFP text within the leaf cells, seven different logical types of nodes are embedded in the physical cells of the overall machine tree structure. Of these nodes, those that are associated with an innermost FFP application further differentiate and split themselves into six different node types to form an *SN-CP network* that is based on the syntactic structure of the underlying FFP text. This network is quite similar in structure to the derivation tree for its underlying FFP text expression, and is derived by effectively parsing the text to discover the topmost structure levels. For this reason, the SN-CP network is also referred to as a syntax tree.

Reduction of an innermost application is then guided and performed entirely within its dedicated syntax tree in response to STL (Syntax Tree Language). STL works by driving the syntax network top-down, to dynamically create processing tasks and data pipes within the SN-CP network. Processes and pipes are freely created, and in multiplicity, resulting in the ability to move data out of the leaf nodes holding the FFP text into the overlying syntax tree, and to move this data in many directions and in support of many processing tasks concurrently (all in the service of a single reduction).

This flexibility has both advantages and problems when compared with Mago's approach. The expanded capabilities of the interior tree cells can increase the efficiency of some reductions because tree cells may be used quite effectively to hold and combine data. In Mago's design, during execution of a reduction, the tree cells are primarily used to support communication between the leaf cells where all non-message data is constrained to reside. Unfortunately, the complexity of implementing the dynamically created processes and pipes of Tolle's design is not immediately clear; he has left this for further investigation. Contention for physical communication channels and tree cell processing time is also an open issue.

There is a very loose coupling between any SN-CP network and the rest of the tree. This is locally good for processing a given reduction, but when more lcells are needed to hold additional FFP text in order to complete a reduction, neighboring SN-CP networks must cooperate. Executive routines running in a more global context of the machine (called the *TA-Mediator network)* monitor execution and detect *molten zones* whose contents may be shifted to create room when needed, but it is not possible to interrupt STL execution asynchronously as in Mago's design. This is because there is no way (and no place) to store the execution context of STL programs, and then restore and reload them following storage management and creation of newly-formed and different SN-CP networks.

FFP text movement is therefore accomplished only in multiple disjoint (molten) areas which allow it. A need for increased space to complete a reduction is automatically satisfied by the machine only if there is enough space in the local molten zone. Reductions may therefore be delayed even though the machine as a whole has ample space. This potential for process interference, in a way similar to that encountered in Treleaven's design, makes it very difficult to predict program execution time for this design.

1.2.6. Cooperating Reduction Machines (Kluge)

Kluge [Klu82] has proposed a multiprocessor network to be composed of Berkling's sequential reduction machines. As in AMPS, a number of large-grain processors (i.e. large enough to perform reductions on their own) are used as a processing pool to execute tasks as they are dynamically created through demands for data values. In AMPS, tasks are created by passing demands top-down through a function graph; Kluge's system sees tasks in the unfolding creation of independent executable reductions in much the same manner.

Each executable reduction may be viewed as the program and initial state of a "virtual" reduction machine, which must then be mapped onto a physical processor for execution. Processors are not time-sliced. Instead, they support LIFO execution scheduling in an efficient and pleasing manner. A property of Berkling's reduction mechanism is that the contents of the three system stacks completely specify the state of a reduction. When a new task is mapped to a processor, the required context switch is performed simply by pushing separation symbols onto the three stacks, and then loading the new virtual machine. This virtual machine will execute to completion (unless interrupted by additional tasks), and then continue the execution of the interrupted context.

The task scheduling mechanism and a means of controlling the migration of tasks between processors are important aspects of Kluge's design. Migration and the resulting creation of parallelism is controlled through the use of local ticketing operations that are independent of network topology. Because of the local character of this load balancing, and other restrictions placed on process migration, it is possible that overall multiprocessor utilization might be poor. The price of increased flexibility in processor scheduling would be increased potential for contention within shared communication resources. Because Kluge leaves network topology an open question, little more can be said concerning this tradeoff.

The use of an arbitrary number of processors in this design helps support the architectural concurrency exhibited by reduction languages, but with parallelism limited by the number of available processors. This is analogous to the situation for AMPS, but communication costs seem potentially worse here. This is because complete copies of executable reductions must be passed between machines, and arbitrarily large results returned to their enclosing reductions. AMPS experiences a corresponding problem related to graph reduction and contention for shared data structures between processors. However, the local cache mechanism used to alleviate this problem in AMPS has no counterpart in the string reduction design suggested by Kluge. As in the case of AMPS, there is no clear way of deriving good estimates of program execution time.

1.3. Other Related Work

We now briefly review work performed here at UNC that is closely related to the MM1 proposal of Mago, upon which this dissertation is based.

1.3.1. Time and Storage Analysis (Koster)

Alexis Koster developed a methodology for analyzing time and space requirements of FFP programs on a machine organized around the principles suggested by Mago for MM1. In Koster's dissertation [Kos77], generic performance characteristics loosely representative of MM1 are assumed, and the times required to execute primitive operations are expressed in terms of a clock cycle time (essentially, the time required to pass information from one cell to a neighbor). Execution times and storage requirements for general expressions (in FFP languages, even programs are expressions) are then derived and applied to a variety of program segments.

In order to simplify the analysis, Koster assumes that the storage management phase of the machine cycle takes no time. As mentioned, this is the phase of the machine cycle most sensitive to process interference, so Koster's results give lower-bounds. With his approach, useful and interesting results are made available with a minimum of difficulty, and he was able to successfully analyze a variety of matrix multiplication programs (showing clear tradeoffs between time and space requirements), and a tree traversal program.

The work described by Koster in his dissertation has subsequently been extended in a joint effort [Sta81, Mag82], to include upper bounds analysis as well.* The analytic model of execution time to be presented in this dissertation is based on this work.

1.3.2. Message Routing (Kehs, Pargas, Presnell)

David Kehs has investigated the idea of using connections between horizontally adjacent tcells in a tree of processors similar to MM1 to route data between and among leaf cells of the tree [Keh78]. Theoretical results are presented to indicate the potential for increased efficiency of data movement.

Roy Pargas has investigated the use of a tree machine similar to MM1 for the solution of partial differential equations [Par82]. He presents an interesting and powerful high-level mechanism for communication routing within the tree called *GDCA* (Generalized Distributed Communication Algorithm). GDCA requires programmable implementation support within the tree cells on a per-message basis, but Pargas does not suggest a means of implementing this facility; his analysis of algorithms for the solution of partial differential equations assumes that the tcells have already been programmed to behave as necessary in support of a message routing. While this absence of concern for an implementation for GDCA is unfortunate, his results show the power and value of generalized routing within a tree structure.

* Also of interest in this regard is the analysis by Williams [Wil81] of algorithms for parallel associative searching algorithms on tree machines.

1.3.3. Virtual Memory (Frank, Siddall)

Geoffrey Frank's dissertation [Fra79] takes a formal mathematical approach in exploring the idea of a virtual memory for a machine organized around the principles of MM1. Data is to be kept in a second level store until needed by an innermost application, at which time room is made for it in the leaf array and it is brought in for reduction. Advantages of such a scheme include fewer symbols to be shifted about during storage management, and the ability to execute programs that are larger than the capacity of the leaf cell array. The two-level memory hierarchy is hidden from FFP user programs by a *virtual memory interpreter*.

Frank's dissertation considers aspects relating to correctness and implementation of the interpreter, and investigates the time and space efficiency of programs under such an execution regime. Improved execution time and space requirements are shown for some FFP programs.

William Siddall has continued investigations along this line [Sid83] by examining a variety of different virtual memory schemes for FFP interpreters. He developed a simulator for storage management that allows the performance of these schemes to be evaluated. Our present design incorporates one of his suggested approaches, which allows FFP text movement into and out of the machine through the leftmost lcell. This allows both program entry and overflow.

1.4. Dissertation Overview

The objective of the research described in this dissertation is the design of a computing system for maximally parallel and efficient execution of FFP language programs. The system design we present is an outgrowth of the tree-structured architecture implementation suggested by Mago [Mago79], and is specific and concretely verifiable; it is, in fact, executable. It completely supports all parallelism architecturally implicit in FFP languages, and its efficiency is the result of a very high degree of lower-level implementation parallelism.

The primary constituent of this design, a complete and detailed model for implementation of the architecture, is called *DOT (Distributed Operating system model of a Tree-structured multiprocessor)*. DOT is represented using active and passive abstract data types (tasks and classes) in the C programming language. This allows simulation of the architecture implementation during execution of actual FFP programs, and is invaluable for the verification of what is a highly complex and concurrent system. In addition to the DOT implementation model, this dissertation presents an accurate analytic model of program execution time based on the algorithms and communication protocols used.

The name DOT was chosen to emphasize the fact that the implementation model it represents may be viewed as a distributed operating system embedded in hardware and firmware.* In this dissertation, the term "DOT machine" will often be used to denote a multiprocessor organized and operating as indicated by the DOT design.

Although the overall computing system is designed to support FFP in a "direct execution" sense, the individual processing units from which it is constructed (whose implementations are represented in DOT) do not execute FFP. Instead, these individual processing units cooperate in order to collectively parse FFP text, and then load and execute "microprograms" that implement the required FFP operators through cooperative and highly parallel action.

* An operating system typically performs memory management, process control, input-output operations, and runtime support for interprocess communication. DOT performs all of these functions.

The intermediate level programs that determine this cooperative action are therefore the other half of the story. They are expressed in *LPL*, a low-level concurrent programming "assembly" language with specialized message passing and process creation features. All computation on the architecture is guided and determined by LPL programs, which can implement powerful functions as low-level highly parallel manipulations of FFP program text.

The DOT implementation defines the LPL architecture, whose purpose is to fit between the arbitrarily powerful and high-level FFP view, and a lower level composed of simple and restricted operations made available directly by hardware and firmware. LPL represents a major component of the design. The analytic model of program execution includes parameters based on LPL definitions of the FFP operators as well as parameters determined by DOT. This dissertation therefore presents a complete programming system, including LPL programs for a powerful set of FFP operators.

1.4.1. Dissertation Organization

The programming system we present is composed of three logical levels. The top (user) level is that of FFP languages, and the middle (system support) level is that of LPL, the concurrent programming language used to define and implement arbitrary FFP operators. These two levels are supported by DOT, and are described in Chapter 2.

DOT is both a design and an implementation model for the desired parallel architecture. It is the lowest level of the programming system, as we examine it here, and is described in Chapter 3.

In Chapter 4, we describe the simulation approach taken, and present results of various simulation studies. Chapter 5 then presents an analytic model of program execution time (for a restricted set of programs), and verifies its agreement with the results of simulation studies.

In Chapter 6, design alternatives referred to throughout the presentation of the architecture and programming system are collected and reviewed. Chapter 7 concludes the dissertation with remarks concerning the the DOT model and implications for hardware.

1.4.2. Selective Reading

Because we are concerned with presentation of a highly complex and parallel implementation model, certain sections of this dissertation may not be appropriate for the casual reader. To allow selective reading, an overall guide to levels of detail is provided here.

The first section of Chapter 2 is important; it provides a basis for terminology used throughout the dissertation, and includes a simple example of FFP languages. The formal definition of FFP languages may be skipped over if desired, though this leads naturally into the explanation of LPL that follows, which is central to the dissertation. Details of actual LPL programs may then be skipped, by proceeding directly to Chapter 3.

Chapter 3 is organized into four main sections. The first two provide an overview of DOT, and describe the basic machine cycle. These sections should be read. The third section provides detailed descriptions for the processes and objects of the DOT model, and is not essential for a high-level understanding of the programming system. Finally, the last section provides a detailed analysis of the most important algorithms used by the model processes. This is the most formidable part of the dissertation. Though it may be skipped by turning to Chapter 4, this section contains the essence of many difficult problems that had to be faced in order to efficiently utilize the tree-structured communication topology. Formal verification of algorithms via mathematical induction is performed when possible.

Chapter 4 discusses how the DOT implementation is simulated, and is easy reading. The analytic model of program execution time is then presented in Chapter 5. This chapter contains the implications for performance of the algorithms that were analyzed in the last section of Chapter 3, and concludes with a discussion of the degree to which DOT decouples parallel function evaluations -- an important aspect of the design. The introduction and conclusion of Chapter 5 are therefore recommended, while the details of the performance model may be skipped if desired.

Both Chapter 6 and 7 should be read. Chapter 6 is fairly conversational, and provides a feeling for the type of design decisions that were needed to create DOT. The alternative approaches and extensions are motivated by hindsight, so these provide a helpful review. Chapter 7 reviews the dissertation and discusses hardware considerations for further work.

If the chapters are read in their entirety, a ranking of their difficulty in decreasing order would be as follows: 3, 2, 5, 4, 6, 1, 7. Selective reading in order to avoid low-level details should only be required for Chapters 2, 3, and 5. These Chapters therefore contain appropriate pointers to aid readers in avoiding the more difficult sections, should this be desired.

CHAPTER 2

Topmost Architecture Levels

2.1. Introduction

All communication begins with agreement. In order that ideas, a design, or structure be explicated, a common understanding of the task at hand and the terms used to describe it are necessary. Within the context of this dissertation, the terms architecture, implementation, and realization are of utmost importance, and are best used only after agreement on their meaning has been reached. This is especially true because these terms are used in everyday conversation, with little concern for an exact denotation. The term *computer architecture*, for instance, clearly has something to do with logical structure, and such a vague perception is often good enough for informal communication. But implicit faith in commonly perceived meanings can be a stumbling block when exact ideas of fairly technical nature must be communicated. In particular, the difference between architecture and implementation can be quite confusing in the absence of prior agreement. This has been noted by other authors including Delesalle:

> "*Classification is fundamental to human thinking. It is performed in various fields... In computer science, several classifications have been suggested... Authors have addressed, sometimes in passing, the classification of computer hardware. Their taxonomies, however, only address a few incidental concepts, which are not formally specified.* **Also, the subject matter often mixes architecture with implementation.**" [*Del83*]*

* emphasis added

2.1.1. Architecture, Implementation, and Realization

This dissertation presents a programming system composed of numerous levels, and in order to clearly present a cohesive view of the overall design these levels must be correctly placed and described with respect to each other. For this purpose, the terms *architecture*, *implementation*, and *realization* are invaluable. These terms and their use are described by Blaauw and Brooks [Bla83], whose approach I shall use.

> "*The* **architecture** *of a computer system we define as* **the functional appearance of the system to its immediate user,** *that is, its conceptual structure and functional behavior as seen by anyone who programs in machine language. A computer's architecture is by this definition distinguished from other domains of computer design: the logical organization of its data flow and controls, called the* **implementation**; *and the physical structure embodying the implementation, called the* **realization.**" [*Bla83*]

Given the concepts of virtual machines and micro-code implementations, identification of *the* single machine language of a computer system may be problematic. Although questions of architecture and implementation may be relative, for a particular computer system the question of "what is the realization" has a direct and existentially unambiguous answer. One merely points to the actual hardware as it sits before one. In the absence of an actual machine, the manufacturing specifications serve as a *representation* of the realization.

Moving up from the realization level, we enter the domain of architecture and implementation. While the realization level has a comforting and concrete nature to it, the higher levels do not. They are abstractions to be embodied in a realization.

2.1.2. Multi-level Systems

The relativity of architecture and implementation levels is shown in the system of Figure 2.1, in which one architecture is implemented in another. Blaauw and Brooks call such architectures *vertically recursive* [Bla83]. Each upper level of this system may be called an architecture because of its correspondence to a virtual machine language. In additional, all levels but the topmost are used to implement the next highest level.

FIGURE 2.1 – Vertically Recursive Computer Architecture

$$\begin{tikzcd}
\text{Application Language (Arch)} \arrow[d, leftrightarrow] \\
\text{Compiled Language (Imp/Arch)} \arrow[d, leftrightarrow] \\
\text{Assembly Language (Imp/Arch)} \arrow[d, leftrightarrow] \\
\text{Machine Language (Imp/Arch)} \arrow[d, leftrightarrow] \\
\text{Micro-code Language (Imp/Arch)} \\
\text{Computer Hardware (Realization)} \arrow[u]
\end{tikzcd}$$

Reality is more complex than indicated by Figure 2.1. Omitted (or disguised) is an important aspect of architecture implementations: more than one system component and level may be used to implement additional architecture levels. For instance, it is true that assembly language implements a compiled language, but so do the language compiler (which is more usually thought as the implementation of a compiled language), and the operating system (which implements IO and under whose control compiled programs run).

Within DOT, such compiler and operating system aspects are explicitly merged into a single implementation model for the LPL and FFP architectures, and the resulting system structure is depicted in Figure 2.2. DOT implements LPL, and the combination of DOT and LPL implements FFP. Authors dealing with language-driven architectures sometimes speak of embedding the operating system and compiler in hardware. The compiler and operating system aspects taken over by DOT are shown in Figure 2.3.

FIGURE 2.2 -- DOT Implements LPL; DOT+LPL Implement FFP

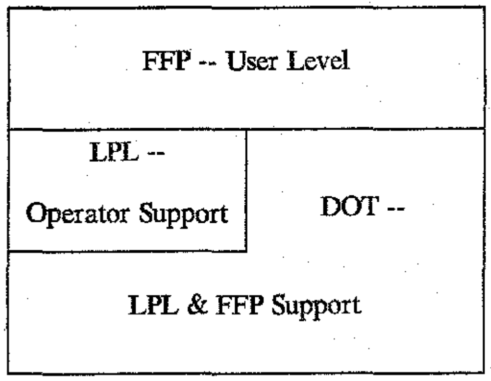


FIGURE 2.3 -- DOT Compiler and Operating System Aspects

(Compiler) o Locate/Parse Innermost Applications

(OS) o Multiprocessor Scheduling

o Virtual Memory

o Input/Output Services

o Storage Management

2.1.3. Architectural Concurrency

Another important point concerning architecture -- one of great importance here -- is that languages can be *architecturally concurrent*. For example, even an assembly language with a no-wait start-io instruction is architecturally concurrent. Architectural concurrency may be seen in terms of the absence of a guarantee (on the part of the language semantics) of strict sequentiality. In the case of the above example, no guarantee is made that an instruction which textually follows a start-io will execute before or after io activities complete.

On the other hand, architectural concurrency may involve more explicit control of multiple processes, as in Concurrent Pascal [Bri77] or Ada [Ich79]. In the absence of a better definition, we will say that a language is *architecturally concurrent* if it admits to parallel interpretation.* FFP and LPL are both architecturally concurrent.

Architectural concurrency can be quite useful. It enables a straightforward expression of many algorithms that are most naturally represented in terms of multiple processes and concurrent behavior. In addition, an implementation is freed from the necessity of strictly sequential support, which may allow valuable gains in run-time efficiency. Per Brinch Hansen has convincingly demonstrated the gains in system throughput that are possible when multiple concurrent processes at the level of Pascal code are used to increase a system's freedom of action (even when the processes are implemented through time-slicing on a single processor) [Bri77].

* A processing unit is a sequential interpreter of its machine language. Parallel interpretation thus involves more than one processor.

2.1.4. Implementing Architectural Concurrency

Given architectural concurrency in a language, an implementer has a variety of options. These include direct parallel support for language concurrency in an implementing architecture, parallel implementation in some way that does not exactly mirror the supported language (perhaps by using limited language-level parallelism as in Treleaven's design in Section 1.2.4, or by using additional lower-level implementation parallelism), and enforcing sequentiality at the implementation level through a process scheduling mechanism such as time-slicing.

In DOT, all concurrent aspects of the FFP and LPL architecture levels have been supported through the use of an even greater degree of parallelism (and at a much finer grain) within their implementation. Moreover, the DOT representation is designed to suggest realization as a highly parallel multiprocessor, in a way that provides direct parallel support within the realization for implementation parallelism.*

We now present FFP, the topmost architecture level of the programming system presented in this dissertation, and LPL, the architecture level that implements FFP operators.

2.2. User Architecture – The FFP Language

Informally, an FFP language program is a linear sequence of symbols, of which four types of symbol are specially distinguished for the purpose of providing syntactic structure: opening and closing application-forming symbols

* The DOT implementation model operates in three modes: it represents a parallel implementation for FFP and LPL; its representation is executable, so it supports simulation of the implementation via writing LPL programs to implement FFP operators, and then actually running FFP programs on it; and lastly, DOT suggests a realization as a tree-structured cellular network of fine-grained processors suited to VLSI fabrication technology.

for *applications*, and similarly balanced list-forming symbols. An application is composed of an operator and exactly one operand. Both operator and operand may be lists and may contain further (i.e., nested) applications. A non-trivial FFP program is an application, and execution proceeds by successively reducing innermost applications according to the semantics of their respective operators until there are no further applications. The ultimate result is a constant (i.e., non-reducible) expression.

The application symbol in our representation is a parenthesis "(", and the list-forming symbol is an angle bracket "<". Within DOT, all program symbols have an associated FFP text nesting level, which removes the need for storage of the balancing symbols ")" and ">". Figure 2.4 gives an execution trace for an FFP program that calculates the inner product of two vectors.

FIGURE 2.4 – Inner Product of < 1 2 3 > with < 4 5 6 >

```
          – The original FFP program is:
( + ( < α * > ( τ < < 1 2 3 > < 4 5 6 > > ) ) )
          – τ (matrix transpose) is innermost, so it is reduced yielding:
( + ( < α * > < < 1 4 > < 2 5 > < 3 6 > > ) )
          – <α * > (apply-to-all multiply) is innermost, and yields:
( + < ( * < 1 4 > ) ( * < 2 5 > ) ( * < 3 6 > ) > )
          – three multiplications are innermost; parallel reduction yields:
( + < 4 10 18 > )
          – + (n-ary add) is innermost, so it is reduced yielding:
32
          – which is the answer (no further applications to be performed)
```

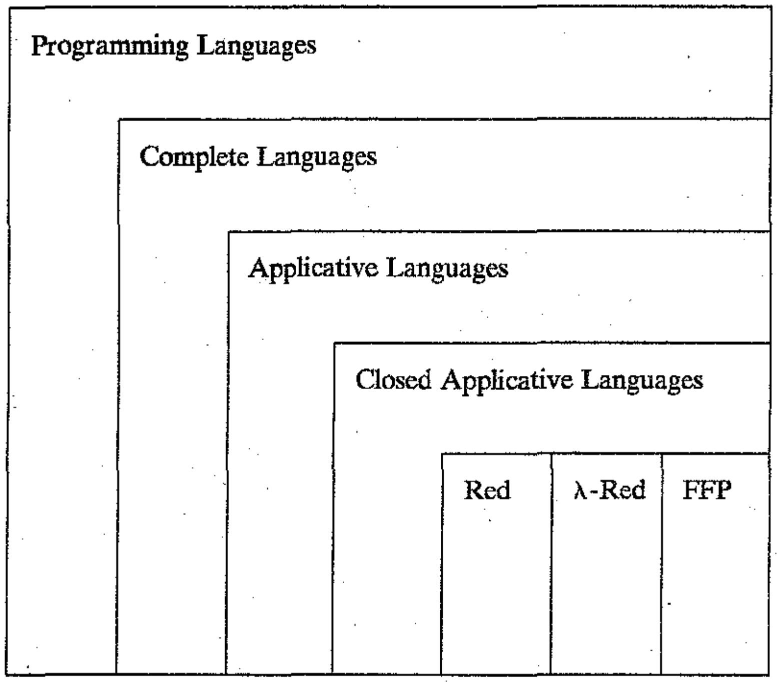
FFP reductions are completely local in nature and are tightly encapsulated with respect to the rest of the program. This fact allows immediate, completely parallel and non-interfering execution of all innermost applications (hereafter referred to as *reducible applications*, or *RAs)*, and it is this property of FFP languages that makes them so attractive for multiprocessor support. User programs are actually written in FP, a human-engineered version of FFP described by Backus that allows programs to be written in a more structured and understandable fashion [Bac78]. A pre-processor based on macro expansion is used to convert FP programs to the equivalent FFP representation. The FP program corresponding to the inner product example in Figure 2.4 is:

$$IP == + \;@\; \alpha^* \;@\; \tau,$$

where @ is used to represent functional composition.

2.2.1. Backus' Language Hierarchy

FIGURE 2.5 – A Hierarchy of Programming Languages



Although FFP is the user-level architecture whose implementation is the object of this dissertation, FFP is part of a larger hierarchy of languages suggested by Backus [Bac73]. A brief review of this hierarchy, based on terminology suggested by Backus [Bac73], is now given. This will enable us to refer to some of the more formal aspects of FFP semantics in following sections.*

2.2.1.1. Programming Languages

A *programming language*, **L**, comprises:

L1) A set of expressions, E
L2) A domain of discourse, D
L3) A semantic relation, $\sigma \subseteq E \times D$

Thus, a programming language is a triple, $L = (E,D,\sigma)$, and when $(e,d) \in \sigma$, we say that $d \in D$ is a *consequent* of the expression $e \in E$.

2.2.1.2. Complete Languages

In a *complete language*, the semantic relation is constrained to be a function, called μ. The domain of discourse, now called C, is constrained to lie within the set of language expressions, and is the set of fixedpoints of μ.** If $\mu(e) = c$, then we say that c is the *meaning* or *value* of e. Thus, for instance, $\mu(2+2) = 4$. The function μ need not be defined for all expressions; $\mu(1/0)$ might be undefined, for example. Formally, then, a language $L = (E,C,\mu)$ is complete iff

CL1) $C \subseteq E$
CL2) μ *is a partial function from* E *onto* C
CL3) C *is the set of fixedpoints of* μ

* Readers uninterested in these details may skip to Section 2.2.2, which concludes the discussion of FFP, and leads into Section 2.3, on the LPL lcell programming language.

** A fixedpoint, x, of a function, f, satisfies the equation f(x)=x. Chapter 5 of Manna [Man74] provides a good introduction to the fixpoint theory of programs.

Elements of C are called *constants* since $\mu(c) = c$, and μ is called the *semantic function* since it determines the meaning of expressions.

2.2.1.3. Applicative Languages

In an *applicative language*, a constructor syntax is employed to create expressions, some of which are called applications, and the semantic function, μ, is then tailored to handle such expressions. The use of a constructor syntax partitions a set of expressions into atomic and non-atomic expressions (a familiar constructor syntax is that used for lists), and is a natural way to specify a simple and regular syntax. Given a set of atoms, A, the pair (A,K) is called a constructor syntax for E iff

CS1) $A \subseteq E$
CS2) Each $k_n \in K$ *is a function:* $E^n \to E$, $n \geq 0$
CS3) if $e \not\subseteq A$, *then there is a unique* k_n *and* $e_1 \ldots e_n$ *such that* $k_n[e_1 \ldots e_n] = e$

Thus, if E has a constructor syntax, every valid expression $e \in E$ is either atomic (in which case, $e \in A$), or has a unique representation $k_n[e_1 \ldots e_n]$ built by a constructor.

We can now define an applicative language as a complete language, $L = (E,C,\mu)$ with an associated constructor syntax (A,K) such that

AL1) $A \subseteq C$
AL2) There is a binary $ap \in K$ *such that* $\mu(ap[e_1,e_2]) = \mu(ap[\mu(e_1), \mu(e_2)])$
AL3) $\forall k_n \in K\text{-}\{ap\}$, $\mu(k_n[e_1 \ldots e_n]) = k_n[\mu e_1 \ldots \mu e_n]$

Clause AL1 indicates that atoms are their own meanings. Clause AL3 indicates that the meaning of an expression that is constructed by constructors other than the ap constructor is simply the construction of the meanings of the expression components. Expressions built using the ap constructor are called *applications*, and clause AL2 indicates that the meaning of an application must be found by first computing the meaning of its components. In addition, clause AL2 shows that the ap constructor is special since it affects the the meaning of the expression it constructs -- thus the outermost μ on the right hand side of AL2.

2.2.1.4. Closed Applicative Languages

The above definition of applicative languages doesn't restrict the way μ, the meaning function, acts on applications. In what Backus has called *closed applicative languages*, μ is restricted by requiring it to operate on applications of the form ap$[e_1,e_2]$ as if e_1 is a function and e_2 is its argument. To take this step, however, there must be a mapping from an expression (in this case, the expression e_1) to the function which it represents. This can be accomplished through the use of a *representation function* called ρ.

In 1973, Backus defined ρ as mapping constant expressions to functions which map constants to expressions. This yields the class of closed applicative languages. Using different constructor syntax and constraining ρ and μ in different ways then yields the Red and λ-Red language classes [Bac73]. (Actually, as Backus later realized [Bac78], ρ and its range of functions can be extended so they are defined for all expressions -- not just constants.)

A *closed applicative language* is therefore an applicative language L = (E,C,U) with constructor syntax (A,K), and an associated representation function, $\rho \in [C \to [C \to E]]$ such that

CAL1) ρ *is total over C*
CAL2) $\forall c \in C,\ \rho(c) = f \in [C \to E]$ *is total over C*
CAL3) $\forall c_1, c_2 \in C,\ \mu(ap[c_1,c_2] = \mu(f(c_2)),\ f = \rho(c_1)$

Clause CAL3 specifies the computation of the meaning of ap$[c_1,c_2]$: the function $\rho(c_1)$ is applied to c_2. If the result is a constant, we are done. Otherwise, we apply to this result the appropriate axiom among AL2, AL3, and CAL3.

The definition of closed applicative languages tells us how to evaluate applications whose components are constants. The differences between FFP, Red, and λ-Red languages (all of which are closed applicative languages) arise from differences in ρ, and the manner in which applications involving non-constant expressions are evaluated. In both of these respects, FFP and Red languages are quite similar. FFP languages amount to a later and conceptually simplified version of Red languages. For this reason, a description of Red languages will be omitted here.

λ-Red languages, on the other hand, are quite different from FFP languages; they resemble the λ-calculus. λ-Red languages differ from λ-calculus in the following aspects: no bound variables need be converted as by I- and α-conversion in the λ-calculus, and innermost applications can be immediately evaluated. These are both important factors which make λ-Red languages easier to implement than λ-calculus based programming languages.*

2.2.1.5. λ-Red Languages

In λ-Red languages, the set of atomic constants, $A \subseteq C$, is the union of two disjoint sets: *variables*, V, and *objects*, O. Five different constructors are used: ***pair***, ***lambda***, ***application***, ***formal application*** (all of which are two-place constructors); and ***bottom*** (a zero-place constructor). A typical set of textual representations for these constructors is:

$$\mathit{pair}(e_1,e_2) \equiv \langle e_1,e_2 \rangle$$
$$\mathit{lambda}(e_1,e_2) \equiv \langle \lambda e_1,e_2 \rangle$$
$$\mathit{application}(e_1,e_2) \equiv (e_1{:}e_2)$$
$$\mathit{formal\ application}(e_1,e_2) \equiv (e_1.e_2)$$

* The following section on λ-Red languages, included for completeness, may be skipped by the casual reader.

$bottom() \equiv \perp$

The difference between application and formal application is that $(e_1{:}e_2)$ is a valid expression iff e_1 and e_2 have no free variables, while $(e_1.e_2)$ is well-formed only otherwise. Formal applications cannot be reduced until one or more λ-substitutions transform them into applications eliminating all free variables.

As closed applicative languages, λ-Red languages satisfy axioms AL1, AL2, and CAL3. Their semantics are as follows.

$$\mu(a) = a,\ a \in A = O \cup V$$
$$\mu(<e_1,e_2>) = <\mu e_1,\mu e_2>$$
$$\mu(<\lambda v,e>) = <\lambda(\mu v),\mu e> = <\lambda v,\mu e>,\ v \in V$$
$$\mu(e_1{:}e_2) = \mu(f(\mu e_2)),\ f = \rho(\mu e_1)$$
$$\mu(e_1.e_2) = (\mu e_1.\mu e_2)$$

Recall that ρ maps elements of C into functions [C→E]. The set C, here, is composed of all expressions containing no applications of the form $(e_1{:}e_2)$. As above, let $v \in V$, and let $c, c_1, c_2 \in C$. Then ρ is defined for λ-Red languages so that

$$\rho<c_1,c_2> = f(\mu(\rho(c_2))),\ f = \rho(c_1)$$
$$\rho<\lambda v,c> = \lambda(v,c)$$

where for every variable $v \in V$, and every constant $c \in C$, $\lambda(v,c)$, an auxiliary function, is a function from C into E defined in such a way as to express lambda-abstraction in the presence of free and bound variables. The first of the above rules for ρ thus expresses regular functional composition, and the second, lambda-abstraction.

2.2.1.6. FFP Languages

In FFP languages (and Red languages), a very simple constructor syntax is used, and there are no variables. Members of the set of atomic constants, $A \subseteq C$, are called ***objects***. Bottom, $\perp$, is a special object used to indicate "undefined".

There are only two constructors: *sequence* (an n-ary constructor for constructing lists), and *application* (a binary constructor). Expressions are thus either atoms, lists of the form $<e_1,...,e_n>$, or applications of the form $(e_1:e_2)$. As with the λ-Red languages, the constants are those expressions containing no applications. The representation function, ρ, maps atoms to functions $[C \rightarrow E]$, i.e., $\rho: A \rightarrow [C \rightarrow E]$, and the set of atoms for which $\rho(a)$ is defined represents the *primitive functions* of the language.

Note that the domain of the representation function does not include non-atomic constant expressions. The semantic function, however, deals with such cases by providing a way of reducing such expressions to the application of a primitive function. This mechanism is called *meta-composition*. The semantic function obeys the following rules.

FIGURE 2.6 – Semantics of FFP Languages

a)	$\mu(a) = a,\ a \in A$
b)	$\mu(<e_1,...,e_n>) = <\mu(e_1),...,\mu(e_n)>$
c)	$\mu(e_1:e_2) =$
c1)	$e_1 = \perp \rightarrow \perp$
c2)	$e_1 \in A \rightarrow \mu(f(\mu(e_2))),\ f = \rho(e_1)$
c3)	$e_1 \in C \text{ and } e_1 = <y_1,...,y_n> \rightarrow \mu(y_1:<e_1,e_2>)$
c4)	$e_1 \notin C \rightarrow \mu(\mu(e_1):e_2)$

Clauses a and b are as expected in closed applicative languages. Clause c represents a further restriction of CAL3. Clause c1 says that the special object bottom represents the function that always returns bottom (i.e., undefined). Clause c2 indicates that e_2 must be evaluated before $\rho(e_1)$ is applied to it. Thus, function arguments are always evaluated before function application -- this shows the data-driven character of FFP. If $\rho(e_1)=\perp$ (i.e., the atom e_1 does not represent a primitive function of the language), then clause c1 shows that meaning of the application is $\perp$ independent of the function arguments. Clause c3 represents the rule for *meta-composition* (to be elaborated on below), and indicates how to reduce an application involving a function represented by a non-atomic constant. Clause c4 says that if e_1 is not a constant expression, innermost applications must be performed first in order to reduce e_1 to a constant. Clause c2 and c4 together show that FFP languages have what is called *innermost reduction semantics*.

Meta-composition is a clever formal device that not only handles evaluation of applications involving non-atomic function expressions (thus allowing user-defined functions to be represented as expressions involving the primitive functions of the language), but also permits definition within the framework of FFP languages of recursive and iterative functions as well as more general functional forms (i.e., functions that are parametrized). Its use will be explained further in the following section, as we clarify some of the issues raised by the above definition of FFP languages.

2.2.2. Observations and Examples

In an FFP language, as explained above, atoms represent the primitive functional operations of the language. The semantics of FFP languages indicate that when such an atom is encountered in the operator position of an application, the application should be replaced by the result of evaluating the represented function on its argument. To do this, we must either know what function is represented (analytically), or have an algorithm for computing the represented function. Mathematicians often take the first approach; a function is defined analytically in terms of other known functions. Thus we might agree on certain functions known to τ, an *evaluation mechanism* corresponding to μ, and then define ρ and its range accordingly.* Backus uses this approach to give an example of the use of meta-composition [Bac78]. He defines the functional form associated with constant functions, which is represented in FFP by the sequence $< CONST\ n >$ when ρ(CONST) is defined appropriately. Backus therefore gives

Def
$\rho(\text{CONST}) \equiv 2 \ @ \ 1$

where τ (an evaluation mechanism corresponding to μ) is assumed to "understand" FP. He could have also given

Def
ρ(CONST) $\equiv$ the second element of the first element of the argument list

where τ is assumed to understand English. In either case, evaluation of the following expression (in which the constant function whose value is always 5 is applied to its argument, 8) would take place as follows:

$$(< CONST\ 5 > : 8\ ) \rightarrow (CONST : < < CONST\ 5 > 8 >) \rightarrow 5\ .$$

The first step in the above evaluation is indicated by the FFP meta-composition rule, c3; the second, by rule c2 and the definition of ρ(CONST). Note that following use of the rule for meta-composition, the first element of the function expression, called the ***controlling operator*** of the functional form (CONST is the controlling operator here), always has as its operand a pair whose first element is the original function expression. This is what allows recursive and iterative operators to be defined so easily within the simple framework of FFP languages.

* The distinction between the evaluation mechanism represented here by τ, and the semantic function represented by μ is primarily one of implementation. The above definition for FFP languages gives equivalences between the meanings of expressions; an evaluation mechanism makes use of these equivalences to produce an actual result. Backus has formalized this distinction using the concept of ***strict language realization*** [Bac73].

An example of ρ without the use of meta-composition is the following:

Def
ρ(IP) ≡ the inner product of the two argument vectors .

Evaluation of the following expression would then take place as follows:

(IP : < < 1 2 3 > < 4 5 6 > >) → 32 .

The semantics of FFP languages lets us use ρ to define whatever primitive functions are appropriate to our needs. The question to be asked, then, if we are to produce a realistic implementation for FFP, is "how much should τ know"? Or, equivalently, "how do we define the functions of ρ"? Our answer is the combination of LPL, through which the functions of ρ are defined algorithmically, and DOT, which incorporates an evaluation mechanism τ that understands (among other things required by the semantics of FFP) how to evaluate applications of primitive operators defined in LPL.

Backus presented FFP languages in his 1977 Turing Award Lecture, which was primarily devoted to describing FP (Functional Programming) languages and their associated algebra of programs [Bac78]. FP languages are an important step in programming language evolution. They are higher-level and less constricted in their syntax than FFP languages, and are designed for ease of use by human programmers. They are capable of expressing parallelism in a natural and functionally powerful fashion, and are amenable to manipulation, transformation, and verification through use of their associated algebra.

In his original presentation, Backus motivated FFP by its similarity to the FP languages, and noted the feasibility of simple and direct translation from FP to FFP. As shown above, the FFP language class can be given a very succinct definition; both the syntax and semantics of FFP languages are extremely simple. This doesn't mean that FFP languages are in any way limited in their expressive power, though. The representation function ρ allows use of arbitrarily powerful primitive operators, and meta-composition provides an high degree of extensibility.

FFP languages can be easily augmented with a definitional facility: if an atom in the operator position of an application represents a user-defined function, then it is replaced by an expression representing its FFP definition, yielding a new application whose meaning is the same as that of the original application.*

Backus has pointed out the importance of distinguishing between the framework and the changeable parts of a programming language. In conventional programming languages, the framework tends to be elaborate and complex, while the changeable parts lack expressive power. In an FFP language, the framework is small and simple, yet able to accommodate a powerful and wide variety of changeable parts.

2.3. Implementation Architecture – The LPL Language

LPL allows the definition of powerful FFP primitive operators. Inner product, matrix transposition, and matrix and multiplication are feasible, as are all of the primitives suggested by Backus in his Turing Award Lecture. The rest of this

* Backus used a mechanism to support such user-defined operators that was based on a special backing store to hold the definitions. This is actually unnecessary since ρ can be extended to handle such definitions. The function represented by $\rho(u)$, for a user-defined operator, u, can simply be the function that produces a new expression that is the desired application. The difference between the two approaches is that in the first approach, no reduction is performed -- the operator is replaced by its definition (and the argument is left alone) -- while in the second approach, the application is actually reduced, producing the same result: an application of the desired user-defined operator expression to the original argument. In DOT, we use the second approach because of its regularity within the implementation. The difference between these two approaches is entirely transparent to the user, therefore this decision does not effect the FFP architecture.

chapter is devoted to presenting LPL and its architecture model.*

2.3.1. LPL Architecture

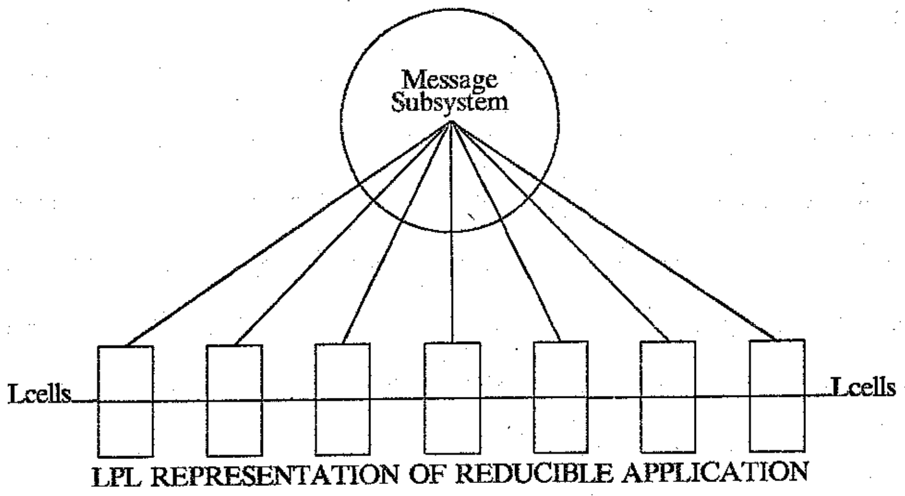
LPL defines an FFP primitive as a program specifying low-level and architecturally concurrent manipulations on the representation of an FFP RA. The LPL-level representation for an RA (not to be confused with the representation function, ρ) is an important aspect of the LPL architecture, and is designed to assist locality of execution within the implementation.

As shown in Figure 2.7, the LPL architecture corresponds to a single RA contained in a linear array of cells, each one of which is a small grain processor used to hold a symbol of the application plus other local state information. These cells, called *lcells*, operate independently and are able to communicate through a globally shared message subsystem.** The linear connections between lcells are not available for communication in LPL, but allow creation of new symbols in the lcell array through an operation called forking. Horizontal communication is not allowed because, at the DOT implementation level, any number of empty lcells may actually be distributed among those holding the text of an RA visible to LPL.

* For the casual reader, Section 2.3.1 provides sufficient overview of the LPL architecture. Section 2.3.2 gives LPL syntax and semantics, and Section 2.3.3 discusses low-level synchronization issues of use to the LPL programmer. Both these sections may be skipped if desired. Finally, Section 2.4 presents actual LPL programs. Although of primary interest to the LPL programmer, these programs are annotated to clarify important aspects of LPL, and may be glanced over to provide a feeling for the size of LPL programs required to implement FFP operators.

** As elements of an architecture, lcells are logical abstractions. In fact, they correspond to what are also called lcells in the DOT implementation, and it is projected that the DOT lcells will be realized as small grain processors. The message subsystem, on the other hand, is implemented within DOT using circuit switched communication channels between entities called tcells (for tree cells). The LPL architecture we present does not require knowledge of the tree structure that implements it. A different LPL architecture might incorporate messages that specify routing through this tree structure, in which case such knowledge would be necessary at the LPL level.

FIGURE 2.7 – LPL Architectural View



As can be seen, LPL corresponds to a powerful multiprocessor architecture. Hence, we can expect fairly efficient definitions for a wide variety of FFP primitive operators. Before the actual capabilities of LPL are discussed, however, let us investigate and motivate this architecture by giving informal definitions for some FFP primitives in English. To do this, a representation for applications must be agreed upon. As a first step towards a representation for the symbols of an RA seen by LPL, we will try to use information appropriate at the FFP architectural level.

2.3.1.1. FFP-level Text Representation

So far, we have used the notation $(e_1 : e_2)$ to represent use of the application constructor. We will now make two changes in this representation. First, we make "(" and ")" special reserved symbols for denoting application. With this change, the colon denoting application is redundant, so we can now write $(e_1 \ e_2)$ to represent an application. This corresponds nicely with

$< e_1 \ldots e_n >$, which represents a list.

A second change in representation is also made in the interest of architectural economy of representation. Corresponding to the constructor syntax of FFP is a unique derivation tree for every valid FFP expression. Figure 2.8 shows an example. The derivation tree shown in Figure 2.8, in which constructors are represented by labeled arcs between expressions, can also be expressed as a parse tree in which constructors appear as non-leaf nodes. Figure 2.9 shows the parse tree corresponding to Figure 2.8. The parse tree can in turn be represented in an ordered linear notation appropriate to the lcell array through the use of level numbers, called *alns* (for absolute level number) as shown in Figure 2.10.

FIGURE 2.8 -- Derivation Tree for e = (< CONST 5 > 8)

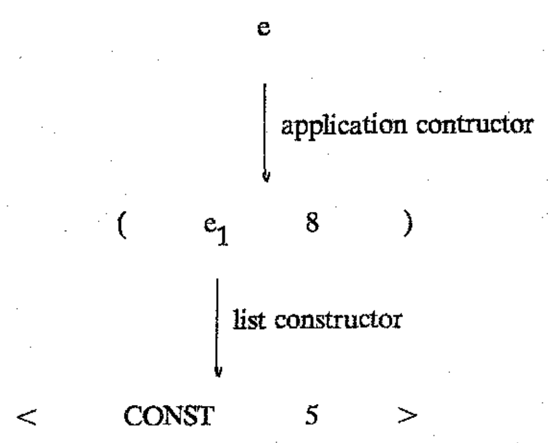


FIGURE 2.9 -- Parse Tree for e = (< CONST 5 > 8)

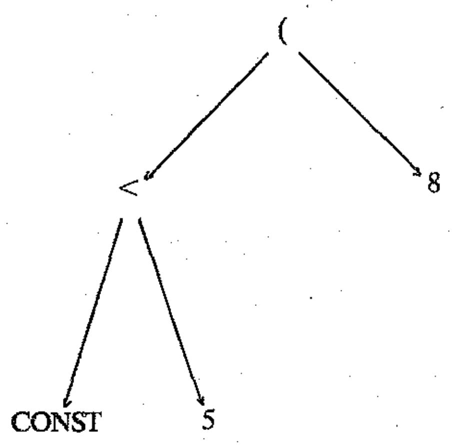
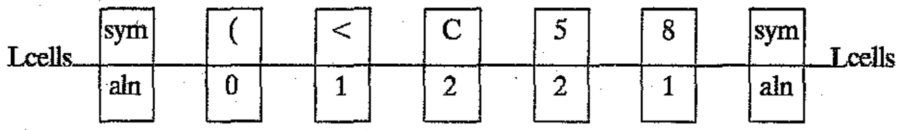


FIGURE 2.10 -- A Linear Representation using ALNs



LPL REPRESENTATION OF REDUCIBLE APPLICATION

The representation of Figure 2.10 amounts to a record of a pre-order traversal of the parse tree of Figure 2.9, and can be easily generated from the FFP text in O(n) time and constant space by a pre-processor. This is the representation we choose to use in the FFP architecture. Its advantages include the use of fewer lcells than would be required by the more straightforward approach of using balanced constructor symbols.

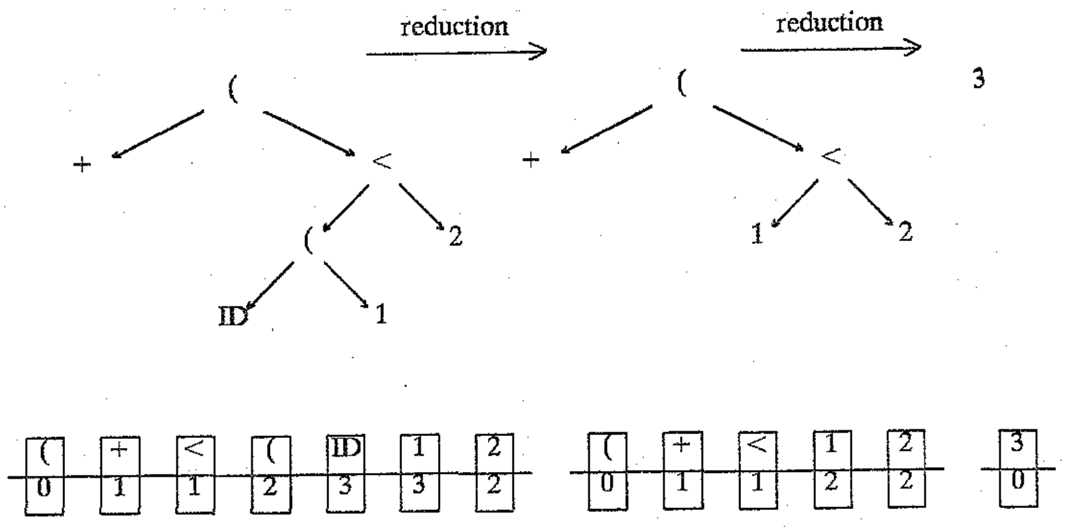
We now define a simple FFP operator, the identity function.

Def
ρ(ID) ≡

a) The application symbol is deleted by its containing lcell.
b) The operator, ID, is deleted by its containing lcell.
c) Symbols of the argument are kept by their containing lcells, but their aln values are decremented by 1.

The level numbers of the argument symbols are decremented so that replacing the application with its result (which is the objective of executing an LPL program) will not change the structure of the containing FFP expression. Assuming for the moment a definition for $\rho(+)$, evaluation of an expression involving ID (which shows how level numbers are modified) is illustrated in Figure 2.11.

FIGURE 2.11 – Evaluation of (+ < (ID 1) 2 >)



In multiprocessor architectures non-local interactions should be kept to a minimum in order to realize the potential of separate processors. The above definition of ρ(ID) requires different behavior by the various lcells based on whether they contain application, operator, or argument symbols. Thus, to avoid non-local interactions, each lcell involved in the reduction of an application of the ID operator should be able to locally discover what part of the RA it contains. Unfortunately, the FFP representation used so far contains no clue as to this information. How then, is an lcell to determine what part of the it contains? Messages between the lcells might be used, but this would require non-local interactions. If we want to efficiently utilize the power of the LPL architecture, the LPL-level representation for symbols of an FFP RA must be augmented with additional information besides that available at the FFP level.

2.3.1.2. LPL-level Representation of FFP RA Symbols

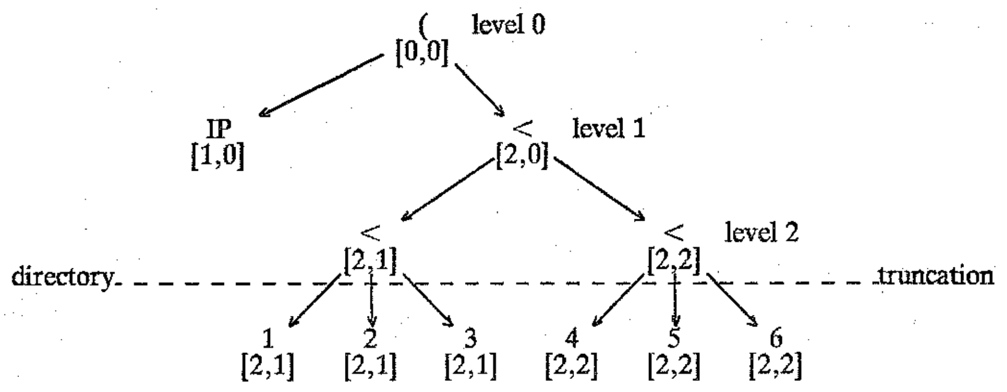
In order to give each lcell information concerning its place in the RA, information descriptive of the location of its contained symbol in the parse tree of the RA is included in the LPL-level representation of the symbol. One approach might be to store within each lcell the parse tree for the entire application, and indicate which node of the parse tree represents the locally held symbol. This would give each lcell complete information concerning its place in the application, but such an approach is unfeasible because of storage limitations. Applications can be arbitrarily large, and we assume the lcell will have a fixed amount of storage.

Instead, in the belief that the topmost structure levels are the most important, we represent the symbol location based on its pre-order position in a "truncated" version of the associated parse tree.* This limited precision

* While this does not provide an lcell with *total* information concerning the place of its symbol in the RA (e.g., the last symbol of an RA cannot be locally identified), it represents a useful compromise. Questions of implementation are left for Chapter 3, but a primary reason for choosing this representation is that it can be efficiently calculated using the overlying tree structure.

representation of a symbol's position within the parse tree of its RA is called a *directory tuple.* Figure 2.12 gives a parse tree for an RA and includes directory tuples resulting from truncation at level 2. As shown by this diagram, a symbol below the level of truncation has the same directory as its ancestor at the truncation level.

FIGURE 2.12 -- Directory Tuples for (IP << 1 2 3 >< 4 5 6 >>)



In addition to the directory tuple (called *d1..d4* in LPL)* the LPL-level representation for each symbol of an RA includes a symbol index (called *symbol_index* in LPL) to guarantee a unique representation for each symbol of an RA.** Also included is a relative nesting level (called *rln* in LPL) which is the nesting level of a symbol relative to that of the application symbol for the RA. The *rln* represents the depth of a symbol within the parse tree of its RA, and is used to calculate the directory tuple. While the *rln* is only defined for a symbol when it is contained within an RA, the *aln* is always defined, and represents the depth of a symbol within the parse tree of the complete program. The

* At present, truncation for the LPL directory is performed at level 4.

** The first symbol of an RA (from left to right) has *symbol_index* = 0, the second symbol of an RA has *symbol_index* = 1, etc.

application symbol for an RA always has *rln* = 0, and *symbol_index* = 0.

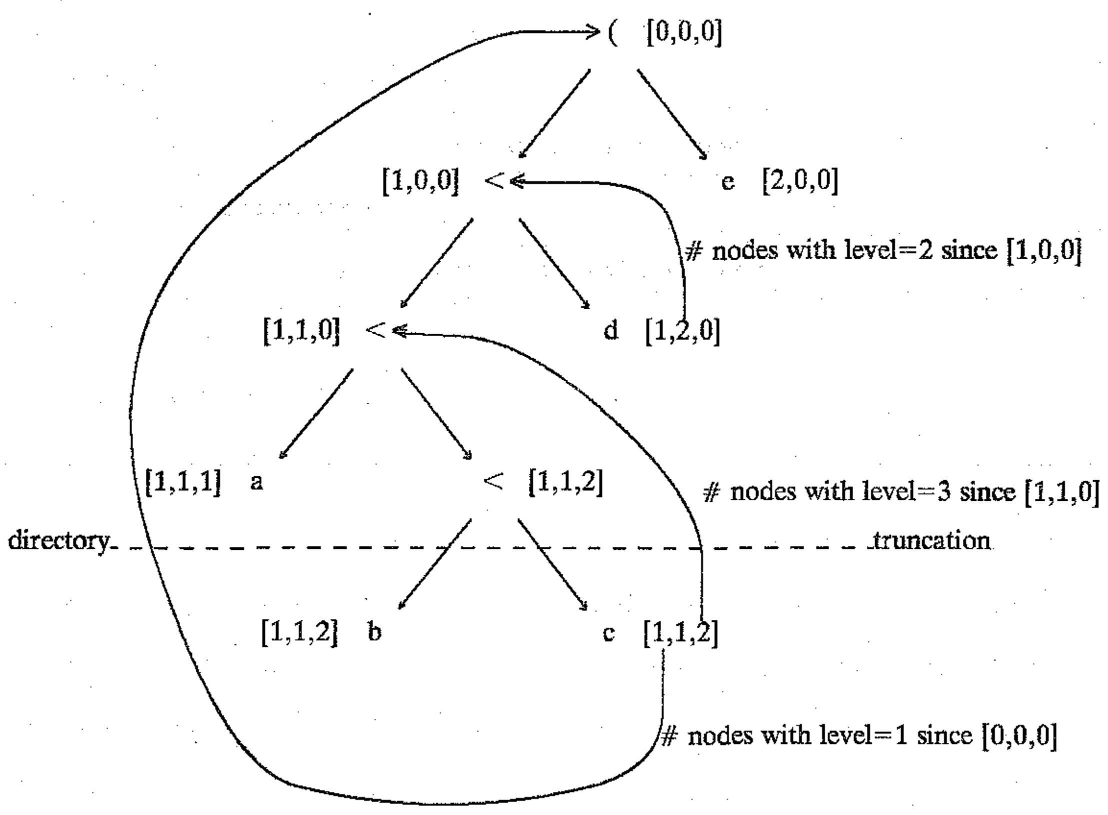
A definition for directory tuples with truncation at an arbitrary level n is now given.

Def
Directory-Tuple ≡

For a symbol of an RA with directory tuple $D = [d_1,\ldots,d_j,\ldots,d_n]$, d_j *is the number of symbols with rln=j (including the symbol of interest) that are encountered in a pre-order traversal from the node with directory tuple* $D' = [d_1,\ldots,d_{j-1},0,0,\ldots,0]$. *The root of the tree has directory* $D_r = [0,\ldots,0]$.

Figure 2.13 illustrates this definition using truncation at level three. As shown by Figure 2.13, the value of a general d_j in a directory tuple indicates the left-to-right count of level j symbols within the RA (up to the one of interest) that are within the scope of the last symbol with *rln*=j-1. For purposes of illustration, the arrows in the diagram point from a level j directory entry to the last symbol with nesting level j-1. The directory tuple for "d" in this diagram is [1,2,0] -- d_1 is 1 because it counts the operator sequence symbol (the only level 1 symbol that occurs before "d" and is within the scope of the application symbol at level 0); d_2 is 2 because it counts the level 2 symbols (including "d") within the scope of the operator sequence symbol (which is the last symbol with nesting level 1, and is pointed to by an arrow); and d_3 is 0 because there are no level 3 symbols within the scope of the last level 2 symbol ("d" itself) to be counted. As another example, the directory tuple for "b" in this diagram is [1,1,2] -- d_1 is 1 because it counts the operator sequence symbol (the only level 1 symbol that occurs before "b" and is within the scope of the application symbol at level 0); d_2 is 1 because it counts the second sequence symbol of the RA (the only level 2 symbol that occurs before "b" and is within the scope of the operator sequence symbol at level 1); and d_3 is 2 because it counts "a" and the third sequence symbol of the RA (the only level 3 symbols that occur before "b" and are within the scope of the second sequence symbol of the RA at level 2).

FIGURE 2.13 – Illustration of Directory Tuple Definition


([0,0,0]
[1,0,0] <
e [2,0,0]
# nodes with level=2 since [1,0,0]
[1,1,0] <
d [1,2,0]
[1,1,1] a
< [1,1,2]
# nodes with level=3 since [1,1,0]
directory
truncation
[1,1,2] b
c [1,1,2]
# nodes with level=1 since [0,0,0]

To summarize, the LPL architecture specifies a representation for symbols of innermost applications that comprises the following:

- *FFP symbol*
- *FFP level – aln*
- *Application level – rln*
- *Application Directory Index -- symbol_index*
- *Application Directory Tuple -- d1,d2,d3, and d4.*

Another definition for ρ(ID) can now be made that clearly involves only actions local to each lcell:

Def
ρ(ID) $\equiv$ if $d1$=2 then keep *symbol* and decrement *aln*.*

2.3.1.3. Message Subsystem

The message model employed in the LPL architecture provides powerful communication primitives that are efficiently and simply implemented, and are easily realized, as projected, in a tree-structured cellular network. In the interest of simplicity, the architecture uses a central message server and a broadcast protocol. In the interest of power, the central message server has added capabilities: it can sort or select messages according to keys, and it can combine the data portion of messages according to associative arithmetic operations such as addition and multiplication. These additional capabilities are reasonable since they have efficient implementations in a tree-structure. Further details concerning messages are given with the LPL statements that use them.

2.3.1.4. Replicating LPL Contexts

Other information besides the above-described symbol representation is available to LPL statements executing in an lcell. The totality of this information is referred to as the *LPL environment* (i.e., that data available to an LPL program). The ClassC representation of the LPL environment is given in Figure 2.14.

* In LPL, the FFP symbols to result from reduction must be explicitly placed or kept (as in the example) within lcells. Thus, there is no need to delete symbols of an RA that will not appear in the reduced result.

FIGURE 2.14 – LPL Environment

```
/*** Symbol Representation ***/
char    symbol,                 /* FFP-level symbol */
        aln,                    /* FFP-level aln */
        rln,                    /* relative level within RA */
/*** Directory ***/
        symbol_index,
        directory[DLEVELS],
/*** Next Symbol Representation ***/
        nsymbol_cnt,            /* validity flag for next symbol */
        nsymbol,                /* next FFP-level symbol */
        naln,                   /* next FFP-level aln */
/*** Message Support ***/
        margs[MARGSIZE], /* hold message args for transmission */
        mtmp[MTMPSIZE],  /* holds received messages */
        temps[TMPSIZE];  /* temporary registers for general use */
/*** Fork Support ***/
        fork_id,                /* environment variable set by fork */
/*** Condition Code ***/
        cc;                     /* condition code set by cmp */
```

The portions of the LPL environment that describe the RA symbol have been explained. The contents and LPL names for the other portions are explained with the statements that use them. The LPL environment is part of a larger context, called the execution context, or *user context* of the LPL program.* The user context includes (in addition to the LPL environment) LPL code in a compiled form appropriate for execution, and other information such as program and message counters.

We have presented some essential aspects of LPL execution in the lcells by investigating ρ(ID). One more element of the LPL architecture remains to be mentioned before giving actual LPL statements. To this end, we give a definition for the FFP primitive, DBL.

* The term "user" is perhaps a poor choice, but refers to the LPL program's "use" of lcell processing power. At the LPL architecture level, LPL programs are user programs.

Def
ρ(DBL) ≡ the pair whose two elements equal the original argument

Thus, for example, the following reductions are indicated:

$(DBL\ x) \rightarrow < x\ x >$, *and*
$(DBL < x_1 \ldots x_n >) \rightarrow < < x_1 \ldots x_n > < x_1 \ldots x_n > >$.

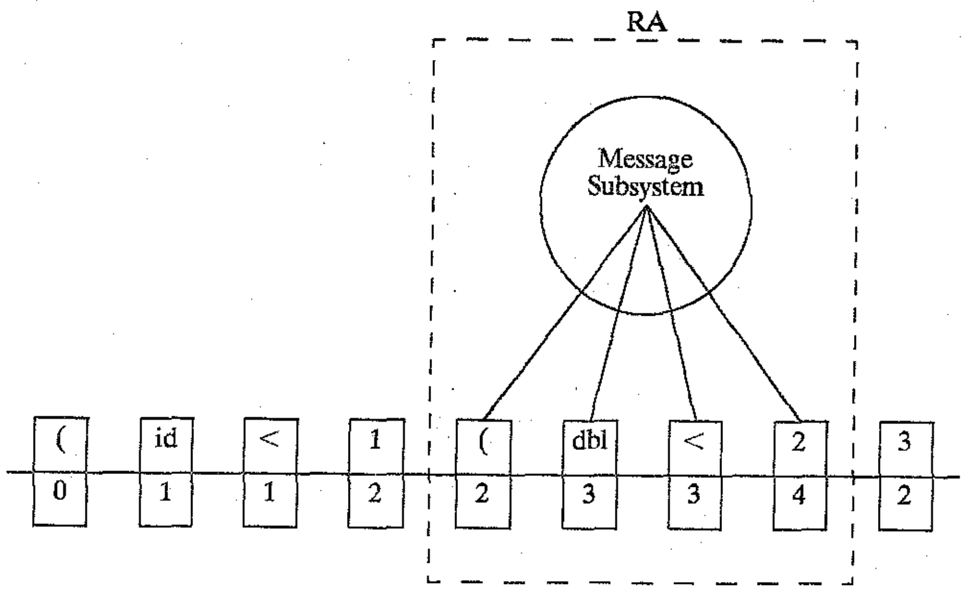
As can be seen, DBL is different in its operation from ID in that new FFP-level symbols must be created, and the expression can grow in size. How does the LPL architecture handle such activity? Copying is required, and the LPL message subsystem can support this. Before this is done, however, LPL environments must be created to act as recipients of these messages.

To accomplish this, we let an LPL environment replicate itself "sideways" in a manner conceptually similar to a "fork" operation, in which a single process is split into two or more parallel execution paths. Within DOT, a process performs execution of LPL code in an lcell of the LPL architecture. Forking this process places copies (called *children)* of the *parent* process's user context into adjacent lcells, shifting the contexts associated with the other symbols of the RA to make room. This is the reason for the horizontal connections between lcells of the LPL architecture. With this ability, much more flexibility in the creation of FFP-level symbols resulting from a reduction is possible. The replicated contexts differ from each other in a single respect: the *fork_id* environment variable is set to 1 for the parent (which is placed leftmost) and increased by one for each child, in left-to-right order. This allows forked processes to condition their behavior in order to perform differently.

The idea used in the LPL program for DBL is to count the number of symbols in the argument using messages. Since the message subsystem supports broadcast routing, this number can be received by an lcell located where the duplicate argument copy is desired. This lcell forks off enough LPL user environments to receive the symbols of the argument, which are sent using additional messages. Each newly-created receptive environment can select the appropriate symbol from the many that are received by matching its *fork_id* with the order of arrival of the symbols, in order to reproduce the argument symbols in the correct order.

Up to now, all aspects of the LPL architecture have been localized to a single RA. The fork operation requires us to admit that there may be FFP-level symbols within the machine other than those seen by a single reduction. To show why this is so, Figure 2.13 depicts a situation involving the use of DBL, and leads to the question of when the fork operation should be allowed to proceed.

FIGURE 2.13 – Forking Must Wait for Storage Management



DOT supports the fork operation during a period of time called storage management, when all LPL programs are held in a quiescent state and execution contexts can be shifted about on the horizontal connections between lcells. A variety of mechanisms for the scheduling of this period are possible. Chapter 6, on design alternatives, will review some of these. The approach we take allows LPL programs themselves to exert a measure of control over this scheduling.

The initiation of the storage management phase is performed in a manner similar to an interrupt on conventional machines. The executing context is saved and the appropriate service routine is initiated. As is the case for conventional assembly languages, we give the LPL architecture the ability to mask out interrupts. LPL programs always begin an execution period with the storage management interrupt masked out. Execution of a fork statement (or other statements to be described below) then removes the mask locally. When all executing LPL contexts have "allowed" the storage management interrupt in this way, storage management and fork operations may take place.

2.3.2. LPL Syntax and Semantics

We now present the syntax and semantics of LPL statements.* LPL is essentially an assembly language appropriate for execution by the fine-grained processors that are expected to realize the lcells. The language is designed to provide simple yet powerful low-level control of the lcells of an RA.

An LPL program defines an FFP primitive by specifying appropriate actions for each lcell of an application. LPL is therefore designed to manipulate local lcell registers containing the LPL environment, and possibly invoke global message operations with which LPL statements in other lcells of the same RA may interact. Various groups of lcells within the RA are given the same instructions (e.g., all elements of a sequence), so an LPL program consists of *code segments* -- one for each such group. The advantage of this approach is

* Readers uninterested in details of LPL, found within the remainder of Section 2.3, may still wish to skim over Section 2.4, which provides examples of useful FFP primitives. Their names, which appear as subsection titles, are generally descriptive of their functions. The introduction to FFP functional forms, in Section 2.4.2, may also be of interest. For the reader primarily interested in implementation details, Chapter 3 may be begun.

that less conditional execution need be specified within LPL segments. This will be made clear in the program examples that follow presentation of LPL.

The most interesting aspects of LPL are the message interactions between the lcells of an RA (controlled with the **send, receive,** and **endfilter** statements) and the way LPL contexts may spawn copies of themselves (controlled with the **fork** and **forkc** statements) in order to create additional FFP text symbols within the lcell array. With these capabilities, LPL programs can implement powerful FFP operators, and parallelism within the tree structure can be used very effectively.

There are no stack-based variables in LPL as in procedure-oriented languages. Instead, the LPL environment variables within local lcell registers are referred to. Some of these environment variables are set up by DOT before LPL statements are allowed to execute. These are *symbol*, *aln*, *rln*, and the directory, composed of *symbol_index*, and *d1,d2,d3,d4* (the directory tuple). In addition to its use by LPL statements, the directory 4-tuple is also used by DOT to choose which code segment of an LPL program should be executed within an individual lcell. This will be explained in conjunction with the LPL destination statement.

Upon completion, the reduction is "stepped forward" to its result. This is done by DOT with the aid of the environment variables *nsymbol_cnt*, *nsymbol*, and *naln*. The "n" prefix stands for "next," and these variables are set up in each lcell of an RA by the LPL program. If nsymbol_cnt is zero when the RA is stepped forward (this is the default), the containing lcell becomes empty (i.e., there is no FFP-level symbol in the lcell following completion of the reduction). If *nsymbol_cnt* is 1 (or non-zero) nsymbol is moved to symbol, and naln is moved to aln.

Thus, the LPL programmer is primarily concerned with creating code which (for each lcell of the RA) will load nsymbol and naln with the symbol and aln values which should next appear within the lcells of the RA in order to implement the required reduction. All code segments of an LPL program must complete (by executing an **endsegment** statement) in the same machine cycle.*

We now give an informal presentation of the LPL statements, explaining their use and purpose. Statements that are closely related are given together in the same section.

2.3.2.1. program/endprogram A program statement is the first statement of an LPL program. Its form is

```
program x
```

where x is the (integer op-code) identifier of the FFP operator the LPL program implements. The LPL assembler creates a library object file for subsequent use whose name is based on this identifier. The end of an LPL program is signaled with an endprogram statement. Its form is

```
endprogram
```

* The machine cycle will be discussed in the following chapter. It arises from the necessity for successive storage management operations during the on-going operation of the machine. Symbols of an RA must be replaced by their reduced result in a single atomic operation between cycles, and DOT presently assumes that if an lcell has completed execution of its code segment, then all lcells of the RA have done so, and the reduction may therefore by stepped forward within the lcell as indicated by the local values of *nsymbol_cnt*, *nsymbol*, and *naln*. It is possible for DOT to guarantee that reductions are correctly stepped forward in the absence of this restriction on LPL segment completion, but at the cost of execution efficiency. The tradeoffs involved are discussed in Chapter 6.

2.3.2.2. destination/endsegment The same sequence of LPL statements is not executed in each lcell of an RA. Instead, an LPL program consists of a collection code segments, each of which begins with a **destination** statement that indicates its the lcells in which it should be executed. The first segment of an LPL program whose destination matches an lcell's directory 4-tuple is the segment that the lcell will execute, and all following segments are ignored.* The form of the destination statement is

```
destination d1 d2 d3 d4
```

where each of ***d1*** through ***d4*** is either an integer, or an integer followed by "*". A match, as referred to above, occurs if each of the lcell 4-tuple directory entries is either equal-to (no "*" used) or equal-to-or-greater-than ("*" used) the respective destination value. The LPL program for ID given in Section 2.4.1.1 illustrates the importance of the textual ordering of **destination** statements.

The end of a program segment is signaled with an **endsegment** statement of the form

```
endsegment
```

Execution of this statement allows storage management for its lcell.

2.3.2.3. Lcell Data Movement and Arithmetic Presently, FFP symbols and other data are bytes. Real, complex, and even vector data of limited size would also be supported in a more realistic implementation. Data movement within the lcell is accomplished with the **mov** statement. It has the form

* LPL program code segments are loaded in order of their textual definition within the LPL program. The word "first" therefore corresponds to textual appearance within the program, as well as temporal appearance of the object code as it is received by an lcell.

```
mov source destination
```

where ***destination*** is one of the named environment variables, and ***source*** can be either an environment variable, or an immediate value. There are two types of immediate values: numeric, in which a numeric string is prefaced with #; and character, in which a single character is prefaced with ". The usual arithmetic operations are also supported. These statements are named **add, sub, mul, div,** and their forms are the same as for the **mov** statement. They behave as usual for arithmetic statements in two-address assembly language architectures.

Another data movement statement is **keep.** It has the form

```
keep
```

It is not primitive since **mov** could be used to achieve the same results, but its use saves space in the LPL object code. Its effect is to move *symbol* to *nsymbol*, *aln* to *naln*, and 1 to *nsymbol_cnt*. The dual of **keep** is **erase,** whose effect is to move 0 to *nsymbol_cnt*. This statement has the form:

```
erase
```

2.3.2.4. Logical Comparisons and Program Control One of the environment variables is called *cc*, and its purpose is to act as a memory to hold the boolean result of past comparisons. Conditional branches refer to it, and it may be manipulated by name as a variable. The **cmp** statement implicitly manipulates it. The form of this statement is

cmp $value_1$ $value_2$ *test cc-op* where ***value**$_1$* and ***value**$_2$* are either immediate values or named variables, and ***test*** is one of the logical comparison operations: "<", "<=", "=", ">=", ">", and "<>". The ***cc-op*** argument is one of ".", "+", and "*", which mean respectively that cc should be loaded, logically "or"-ed, or logically "and"-ed with the result of the comparison. The LPL program for EQUALS given in Section 2.4.1.3 uses two successive **cmp** statements to check for equality of respective *symbol* and *aln* values.

There are no structured program control statements such as "while" or "if-then-else". Conditional branching is provided by the **br** statement. The form of this statement is

```
br cc-test s-label
```

where ***cc-test*** is one of ".", "+", and "-", which mean respectively that the branch should be executed always, if cc is true, or if cc is false. ***S-label*** is the label of the statement that should be next executed if the branch is taken. A **label** statement is used in conjunction with **br** to indicate that an identifier should be associated with the statement that follows the identifier. Its form is

```
label id
```

where ***id*** (a positive integer) is the ***s-label*** to be used in a **br** statement.

2.3.2.5. fork/forkc Forking is the means by which additional lcells are allocated to hold expanding FFP text. The word "fork" is used because each lcell may be thought of as a single process that executes a sequential LPL program segment. A **fork** spawns copies of its program segment and its execution context to create new processes in the requested number of adjacent lcells. Execution continues after allocation and loading of these lcells by DOT (during storage management). A **forkc** spawns completed results in the form of FFP symbols and alns in the requested number of adjacent lcells. Execution does not continue in this case, since the RA is assumed to have completed. The form of the **fork** statement is

```
fork forksize
```

where ***forksize*** is the (non negative) number of lcells desired. The *fork_id* environment variable is set by DOT during support for this operation. The parent of the fork operation is always given *fork_id* = 1, while the children are given *fork_id* = 2 through **forksize** in left-to-right ordering. This fact can be used in subsequent LPL statements to condition execution.

Fork is often followed by **nselect** (explained below), which can use *fork_id* as a selector for the next FFP symbol to be placed in the lcell. Copying or moving groups of FFP symbols into new locations is done by forking LPL environments into the required number of lcells, and then using **receive** (explained below) to selectively accept the desired symbols based on order of receipt and the local *fork_id*. The forked lcell with *fork_id* = 1 accepts the first symbol to arrive, the forked lcell with *fork_id* = 2 accepts the second symbol to arrive, and so on. Temporary registers, *t1* .. *t9* are available for use as message counters and other purposes.

The statement **fork** #1 can be considered a no-op that delays execution until after the next storage management is performed.* Its use can help code segments maintain synchronization over multiple machine cycles, so they all complete during the same cycle. The statement **fork #0** allows an lcell to "drop

* Recall # signals an immediate value as opposed to a variable name.

out" of an RA during the middle of a multi-cycle reduction.* This is a way of freeing up lcells within a reduction as soon as possible, and can allow more efficient storage management. The LPL program for DBL given in Section 2.4.1.4 uses **fork #**1 for synchronization, and also uses the temporary variable *t1* to fork an LPL environment into a variable number of lcells with the statement **fork t1.**

Forkc is similar to **fork.** Its form is

```
forkc forksize
```

Forkc should be preceded by **cselect** (explained below) in order to fill a temporary register array with the FFP symbols and aln values that will be shifted out during the next storage management as a result of its execution. Use of **forkc** can enable improvements in the execution efficiency of LPL programs that are able to complete by forking off FFP-level symbols requiring no further execution. The LPL program for EE1 given in Section 2.4.1.8 uses **forkc** in this way. Both **fork** and **forkc** allow storage management for the lcell in which they are executed.

2.3.2.6. nselect/cselect The **nselect** statement is used to select and load one element of a literal string from an LPL program into *nsymbol* and *naln* while also setting *nsymbol_cnt* = 1. A list of *nsymbol/aln*-offset pairs is given and the effect of the statement is to load the appropriate pair based on a selector. Nselect is thus analogous to a case statement in which the objective is always

* Following a **fork #0**, LPL execution halts in the containing lcell (as usual for fork operations), and during the following storage management phase no descendants (not even a parent process) for the executing LPL environment are created. Thus the executing LPL environment literally disappears between cycles. Care must be taken that repartitioning will correctly detect and connect the RA in the absence of symbols that disappear in this way (thus, an application symbol should never execute this statement). The LPL directory is only created during the first partitioning of an RA, so disappearing symbols don't change the directory.

the loading of *nsymbol* and *naln*. Its form is

nselect *selector* $nsym_1$ $oset_1$ $nsym_2$ $oset_2$ ... $nsym_n$ $oset_n$.

Once a particular *nsymbol/aln*-offset pair has been chosen (based on the value of the selector), *nsymbol* is loaded appropriately, and *naln* is loaded with the present *aln* plus the chosen offset (which may be negative). **Nselect** is often used with #1 as a selector, which allows a single symbol and aln-offset to be selected, and *nsymbol_cnt* to be set, in a single statement.

Nselect can be useful after a fork operation. It allows setting up the next FFP symbols to appear within a group of forked lcells by using *fork_id* as a selector. **Cselect** is designed for use *before* a fork operation when the LPL program requires no further execution and can complete by performing an appropriate storage management. Its form is as follows.

cselect $nsym_1$ $oset_1$ $nsym_2$ $oset_2$ ... $nsym_n$ $oset_n$.

The **cselect** statement is thus identical to the nselect statement, with the exception that a selector is not used. Its function is to load a temporary register array with the resulting values, so that the desired FFP symbols and alns will be shifted out following **forkc**.* The LPL program for EE1 given in Section 2.4.1.8 shows the use of **cselect**.

* The size of the temporary register array within the lcell will be ultimately determined by space considerations related to the lcell realizations. **Forkc** and **cselect** are both important because of the efficiency they allow when compared to **fork** and **nselect**, so the area set aside to support **cselect** should be as large as possible. Presently, this array can hold 20 symbols.

2.3.2.7. send/receive/endfilter

These statements are used for global communication within an RA. Messages are sent and received during globally sequenced activities called *message waves*, and all the lcells of an RA have the option of participating in any of them. A limited amount of processing can take place within the message subsystem of an RA during transmission of a message wave, and appropriate instructions for this purpose are automatically sent up by the lcells to introduce each new message wave. The information necessary for this is supplied in the **send** statement.

The LPL messages within a message wave travel from the lcells into the LPL message subsystem. Here, messages are sorted, combined or passed selectively, and are then broadcast to all lcells in the RA. Those lcells doing either a **send** or a **receive** for that particular message wave then "see" all returning messages for the wave. **Send** and **receive** have a filter portion that describes the actions to be taken for each incoming message, and a DOT lcell message process invokes this filter for each message arrival after first moving the message into a reception area within the LPL environment. The difference between **send** and **receive** is that the former sends a message then filters incoming messages, including its own, while the latter merely filters incoming messages. Their forms are as follows:

```
send mwave order combine-op key1 key2 msize
        filter-statements
        endfilter
```

```
receive mwave
        filter-statements
        endfilter
```

Mwave is the (positive integer) index of the message wave desired, and ***order*** indicates the order in which two messages of differing key values should be returned when broadcast to the lcells of the RA.[*] The possible values for ***order*** are "+", and "-", which indicate respectively larger first, and smaller first. When two messages arriving at a tcell have identical key values, the respective messages are combined according to ***combine-op***. The possible combine operations are addition, multiplication, selection of the message with the largest data value, or selection of the message with the smallest data value. Additionally, a null combine operation is included to prevent combination even if the key values for two messages are the same. These possibilities are respectively indicated by "+", "*", ">", "<", and ".". ***Msize*** is the number of message arguments (in addition to the key values) that are to be sent. Additional message arguments as required by a positive **msize** are taken from lcell registers referred to in LPL as *m_arg1* ... *m_arg5*. When messages are combined arithmetically, it is *m_arg1* that is actually combined. The lcell registers referred to in LPL as *r_key1*, *r_key2*, *r_arg1* ... *r_arg5* are the ones into which the arguments of a message are placed by DOT prior to executing a filter. The LPL program for ATOM given in Section 2.4.1.2 uses messages to send all argument symbols to the application symbol, where they are counted.

Restrictions must be placed on the statements within a filter: nested message requests (i.e., **send** or **receive** statements) are not allowed, and forks are not allowed. Branches may be executed, but only if the branch destination is

[*] Key1 is given precedence over key2.

within the same filter or another filter for the same message wave.*

2.3.2.8. endsend/smanage **Endsend** tells the message subsystem that no more sends will be performed (receives are still allowed) by the signaling lcell. Its use will be discussed in the following section where synchronization between program segments is treated. Its form is:

```
endsend
```

Smanage indicates that the containing lcell is willing to be interrupted for the purpose of storage management. Its form is:

```
smanage
```

Smanage is different from the other three statements that allow storage management within the executing lcell (the **fork, forkc,** and **endsegment** statements) since execution continues following its use. All lcells of an RA must allow storage management before an execution cycle can come to an end, thus one of these four statements must be executed by each lcell of each RA during every cycle. This is analogous to enabling interrupts on a conventional machine. Failure to execute such a statement in one RA lcell will ultimately deadlock the entire machine.

2.3.3. Synchronization of Program Segments

The issue of synchronization for program segments arises in two ways. First, there is the overall synchronization of completion required of all segments in an LPL program. Second, there is the synchronization required for transfer of

* Branching into another filter can be done to reduce code size in a case where a code segment performs either a **send** or a **receive,** both of which require the same message filter. The LPL program for MM (matrix multiply) given in Section 2.4.1.12 does this.

information to forked lcells during a message wave (e.g., when copying FFP-level symbols from one place to another). The LPL programmer must explicitly provide completion synchronization, while synchronization of forks with messages is essentially automatic.

In this section, we show how the **fork, endsend** and **smanage** statements allow control of both types of synchronization. Synchronization of forks with messages is discussed first, since information on message handling is useful for the discussion of segment completion. The LPL program for DBL given in Section 2.4.1.4 provides examples of both types of synchronization.

2.3.3.1. Synchronization of Forks and Messages

Copying FFP text from one location to another requires the coordinated use of the **fork, send,** and **receive** statements in the following way. Destination lcells are prepared by forking the required number of LPL processes, each of which subsequently executes a **receive** statement on (say) wave n. The source lcells are required to execute a **send** on message wave n.

But how can we guarantee that the sent information will not be delivered before the **fork** completes? After all, a **fork** requires storage management, and this is invisible to LPL program segments.

The answer is that the message subsystem requires, for each message wave, at least implicit acknowledgement from each lcell process of its opportunity for participation in the message wave. When a process executes a **send** or a **receive** for message wave n, it is interpreted by the message subsystem as acknowledgement and rejection of participation in all lower-numbered message waves. Only after all processes have either completed or requested message service for the present (or possibly future) message wave will messages for the present wave be delivered. If an lcell forks to receive messages sent on a particular wave, then, only after the fork completes and receive statements are executed by the destination lcells will the message wave containing the information to be copied actually be delivered.

Thus, as long as the same message wave is used by receiving and sending lcells, all necessary synchronization is automatically provided by the message subsystem, even in the presence of forking. The use of **endsend** can now be clarified. It turns off the automatic message synchronization for the executing lcell by telling the message subsystem that the lcell will send no more messages. This allows the lcell process to fork without holding up message waves. Although its use does not preclude subsequent execution of a **receive** statement within the lcell or its descendants, it does remove the above synchronization of sends and receives in the presence of forks (but only for the executing lcell and its descendants).

2.3.3.2. Synchronization of Completion

All LPL program segments must complete by executing **endsegment** during the same machine cycle. For single-cycle LPL programs, this is no problem. For multiple-cycle programs, there are two ways of using LPL statements to synchronize completion.

When the number of cycles is a small constant value (this is the usual situation -- no LPL program given here requires more than 2 cycles) using **fork** #1 to allow storage management and delay completion in segments that would otherwise complete too early is often the simplest approach. This approach must be used carefully if messages are also involved, however. If messages are being sent during the same cycle in which a **fork** is executed, they will not be delivered (for the reasons explained above) unless an **endsend** is executed before the **fork** statement.

Another way to synchronize completion of programs that use messages during their last cycle is to add a **receive** statement (for the last message wave) to segments that would otherwise complete too early. In this case, messages will be handled as otherwise desired (i.e., adding a final **receive** statement doesn't effect message synchronization in the same way as **fork** does), but now the LPL programmer must be concerned with the progress of the machine cycle, and must allow storage management when appropriate. This may be done by using the **smanage** statement.

As will be seen in the following LPL program examples, either of the two alternatives described above is usually possible. The decision as to which approach is best in a given situation is generally a question of style, although questions of code size tip the balance in favor of using **fork** #1 when possible.*

2.4. Remarks and FFP Operator Definitions

We now present LPL definitions for a variety of FFP primitive operators. FFP functions are given first, followed by FFP functional forms. For each operator, we provide a description of behavior, and point out interesting aspects of the LPL code. Where appropriate, the definition of the corresponding FP operator suggested by Backus [Bac78] is also given.** The programs have all been tested, and run correctly on the simulation described in Chapter 4. They provide the basis for many parameters employed in the analytic model of Chapter 5.

* A **fork** statement uses 2 bytes of object code. A **receive** statement (including the **endfilter** statement, and an **smanage** statement) requires 4 bytes.

** Backus' definitions include concern for undefined results, and produce bottom when appropriate. The LPL programs we give assume that the restrictions stated in their header are satisfied. Operators could easily check their arguments for appropriate form, but error handling in FFP languages is a current area of research by Don Stanat and others here at UNC. We have therefore left open the question of implementation support for error reporting.

Each LPL program is prefaced by a header of the following form:

```
FFP OPERATOR -- description of application result
Restrictions:

          Summary of Analytic Model Parameters:
                    program size: x
                    cycles required: x
                    cycle1: messages: x (wave=x; msize=x)
                              forks: x  (completed/executing)
---------------------------------------------------------------
sym:      ( op arg
aln:      0 1  1
---------------------------------------------------------------
dir:      0 1  2
          0 0  0
          0 0  0
          0 0  0
---------------------------------------------------------------
nsym:
naln:
```

FFP OPERATOR is the name of the operator the following LPL program implements. Program size is the total size (in bytes) of the object code which must be loaded in through IO subsystem when the compiled operator definition is required. Also included are the number of machine cycles required, and a breakdown of the communication and fork requirements for each cycle. The communication breakdown includes the number of returning messages for each message wave. Also included is the message size.* The fork breakdown includes the number of new cells required, and whether the symbols forked are completed or executing. The distinction is important to the analytic model because of the difference in context sizes.

* The message size given in the header is the **msize** value coded in the corresponding **send** statement. The number of returning messages and the corresponding message sizes are used in the analytic model.

Also within the header is an example reduction, including the directory for the example. The *nsym* and *naln* values show the result of the reduction. Although a single example may not completely describe the desired behavior of an operator, it is often convenient to refer to the example directory when reading destination statements in the following LPL code. When the symbols of an RA are shifted by forking to make room for information that is to be copied or moved, blanks are used in the header description of the original RA to show where this additional space is made available. The header of the LPL program for DBL given in Section 2.4.1.4 provides the first example of this.

Recall that a destination statement describes, in terms of the four-level lcell directory, the destination(s) that should execute the following segment -- provided that no earlier segment is accepted. An asterisk "*" is used to encode a *wild card* directory match for the level on which is appears; it matches all directory entries (on it's level) that are equal to or greater than the given value. Thus, for instance, destination 2 0* 0* 0* addresses all symbols of the argument, and destination 0* 0* 0* 0* addresses all symbols of the RA. Comments are supplied with a destination statement to make it clear which symbols of the RA are being addressed. These comments often use abbreviations to save space. The application symbol for an RA is referred to as "app sym", the sequence symbol that encloses the elements of an argument list is referred to as "arg seq", and storage management is referred to as "sm".

2.4.1. FFP Functions

2.4.1.1. Identity

Using colon to denote application of an FP function to its argument, Backus defines the result of applying the FP *id* operator to an argument, x, as shown. The LPL definition of the corresponding FFP operator then follows.

Def
$id : x \equiv x$

```
ID -- result is the argument
Restrictions: none

        Summary of Analytic Model Parameters:
                program size: 29
                cycles required: 1
                cycle1: 0 messages, 0 forks
```

```
sym:     ( 23 x
aln:     0 1  1
```

```
dir:     0 1  2
         0 0  0
         0 0  0
         0 0  0
```

```
nsym:         x
naln:         0
```

```
Method: All symbols of the argument remain, but with adjusted nesting.
        The application symbol and the operator erase themselves.

program 023
        destination 2 0* 0* 0*               /* The argument symbols
                nselect #1 symbol #-1 .      /* adjust their nesting.
                endsegment
        destination 0* 0* 0* 0*              /* Everybody else
                endsegment                   /* goes away.
        endprogram
```

This simple LPL program illustrates the value of a clever textual ordering of destination statements. Symbols of the argument receive their LPL program first, after which the destination 0* 0* 0* 0* is used to address all of the remaining symbols of the RA. In addition to placing themselves in the result of the reduction, the argument symbols must adjust their level numbers, and **nselect** allows this to be done with a single statement. Note that symbols other than those of the argument simply execute an **endsegment** statement without placing successors in *nsymbol* and *naln*. These symbols therefore do not appear in the reduced result.

2.4.1.2. Atom

Backus defines the result of applying the FP *atom* operator to an argument, x, as shown. The LPL program for the corresponding FFP operator follows.

Def
atom $: x \equiv x$ *is an atom* $\rightarrow T; x \neq \perp \rightarrow F; \perp$

```
ATOM   -- true (=1) if the arg is an atom, else false (=0)
Restrictions: none

      Summary of Analytic Model Parameters:
              program size: 68
              cycles required: 1
              cycle1: messages: 1 (wave=1; msize=0)
                      forks:  none
```

```
sym:     ( 18 x
aln:     0 1  1
```

```
dir:     0 1  2
         0 0  0
         0 0  0
         0 0  0
```

```
nsym:    1
naln:    0
```

```
Method: The argument symbols send themselves. If the argument is
        an atom, then only one message is received. The
        application symbol checks this and places the result.
        All other symbols go away.

program 018
        destination 0 0 0 0            /* The app symbol
                mov #0 t1              /* counts messages
                receive #1
                        add #1 t1
                        endfilter
                cmp #1 t1 = .          /* one argument symbol?
                nselect #1 cc #0 .     /* place the result
                endsegment
        destination 1 0 0 0            /* The operator
                endsegment             /* goes away
        destination 0* 0* 0* 0*        /* Symbols of the argument
                send #1 + . symbol #0 0 /* send themselves
```

```
                    endfilter
              endsegment                    /* then go away
        endprogram
```

In this LPL program, all argument symbols send themselves. The application symbol receives these, and counts them. If there is but a single argument symbol then the result is true. The result is found in cc after comparing the number of messages received with 1, and the application symbol uses **nselect** to place the answer in the reduction result.

2.4.1.3. Equals

Backus defines the result of applying the FP *equals* operator to an argument, x, as shown. The LPL program follows.

Def
equals : $x \equiv (x=<y,z> \ \&\ y=z) \rightarrow T;\ (x=<y,z> \ \&\ y\neq z) \rightarrow F; \perp$

```
EQUALS  -- result is true iff the argument elements are equal
Restrictions: the argument is a pair

        Summary of Analytic Model Parameters:
        n = # symbols in second argument element
                program size: 134
                cycles required: 1
                cycle1: messages: 1 (wave=1; msize=0)
                                  n (wave=2; msize=1)
                                  1 (wave=3; msize=1)
                        forks: none
```

```
sym:    ( 19 < < a < a
aln:    0  1 1 2 3 2 3
```

```
dir:    0 1  2 2 2 2 2
        0 0  0 1 1 2 2
        0 0  0 0 1 0 1
        0 0  0 0 0 0 0
```

```
nsym:   1
naln:   0
```

```
Method: The number of symbols of the first argument element are
        determined.  Symbols of this element check themselves against
        the corresponding values of the second arg element, and also
        check equal element sizes. The results are combined using
        logical multiplication.
```

```
program 019
        destination 0 0 0 0        /* The app symbol receives the result
                receive #3
                        nselect #1 r_arg1 #0 .
                endsegment
        destination 0* 0 0 0       /* Operator and arg seq go away
                endsegment
        destination 2 1 0* 0*      /* First element
                mov #1 m_arg1      /* Counts itself
```

```
          send #1 + + #0 #0 1
                  mov r_arg1 t1  /* t1 is symbol count
                  endfilter
          mov #2 t2        /* msg counter (offset for cmp index)
          receive #2       /* receive symbols of second elem
                  add #1 t2
                  cmp t2 symbol_index = .  /* cmp this symbol?
                  br - 1
                  cmp symbol r_key2 = .    /* yes
                  cmp aln r_arg1 = *       /* aln must also match
                  mov cc t3                /* save result for later
                  label 1
                  endfilter
          add #2 t1        /* counteract msg cnt offset
          mov t3 cc        /* get back cmp result
          cmp t1 t2 = *    /* symbol cnt must also match
          mov cc m_arg1
          send #3 + * #0 #0 1  /* and all results for app sym
                  endfilter
          endsegment
  destination 0* 0* 0* 0*      /* The second arg element
          mov aln m_arg1       /* sends its symbols to the first
          send #2 - . symbols_index symbol 1
          endsegment
  endprogram
```

The LPL program for EQUALS works in the following fashion. The symbols of first argument element (hereafter referred to as A1) count themselves in the first message wave by using an add combine-op. During the second message wave, the second argument element (hereafter referred to as A2) sends its symbols (ordered by symbol index) so the symbols of A1 can compare themselves with the corresponding symbols of A2. After this, each symbol of A1 uses multiplication for the combine-op of a third message wave to send a boolean value representing whether the symbol is matched by A2 (and A1 and A2 contain the same number of symbols). The application symbol receives this last result, and uses **nselect** to place the correct reduced result.

2.4.1.4. Double

The result of applying the FP ***dbl*** operator to an argument, x, may be defined as shown. The LPL program follows.

Def
$dbl : x \equiv x \neq \perp \rightarrow <x\ x>; \perp$

```
DBL -- result is a pair whose elements equal the original argument
Restrictions: none

        Summary of Analytic Model Parameters:
        n = # symbols in the argument
                 program size: 121
                 cycles required: 2
                 cycle1: messages: 1 (wave=1; msize=1)
                          forks:  1 sym forks n contexts (executing)
                 cycle2: messages: n (wave=2; msize=1)
                          forks: none
```

```
sym:    ( 6     < a b
aln:    0 1     1 2 2

dir:    0 1     2 2 2
        0 0     0 1 2
        0 0     0 0 0
        0 0     0 0 0

nsym:   < < a b < a b
naln:   0 1 2 2 1 2 2
```

Method: Count argument, fork the operator, and receive the argument symbols.

```
program 006
        destination 0 0 0 0     /* The app sym becomes a seq sym
                keep
                mov "< nsymbol
                endsend         /* allow wave 1 to complete
                fork #1         /* allow storage management, and
                endsegment      /* synchronize completion
        destination 1 0 0 0     /* The operator forks
                keep
                receive #1      /* First, get forksize
                        mov r_arg1 t1
                        endfilter
                fork t1         /* Then fork
```

```
            mov #1 t1         /* t1 counts symbols as they arrive
            receive #2
                    cmp t1 fork_id = .  /* Is this symbol for me?
                    br - 1
                    mov r_key2 nsymbol  /* if so, then load it
                    mov r_arg1 naln
                    label 1
                    add #1 t1           /* bump the counter
                    endfilter
            endsegment
    destination 0* 0* 0* 0*             /* The argument
            keep                        /* remains,
            mov #1 m_arg1               /* and counts itself
            send #1 + + #0 #0 1
                    endfilter
            smanage                     /* allow operator to fork
            mov aln m_arg1
            send #2 - . symbol_index symbol 1 /* and send copy
                    endfilter
            endsegment
    endprogram
```

This is the first LPL program we show that requires forking. Since the argument to be copied is not restricted in size, we use the general **fork** statement, as opposed to **forkc**. The approach taken is to count the symbols of the argument during the first message wave, and then fork the operator symbol to receive symbols of the argument sent on a second message wave. The application symbol is replaced with a sequence symbol of the same nesting level in order to encapsulate the resulting duplicate elements.

Note the use of **endsend** and **fork** #1 by the application symbol, and of **smanage** by the argument -- these statements allow storage management to proceed so that the fork executed by the operator symbol can complete. Without these statements, this program will deadlock, effectively halting the entire machine by preventing storage management. The **smanage** statement in the argument segment could be replaced with a **fork #1** statement without changing the behavior of the program. If the application symbol were to use

smanage to allow storage management, however, its execution would continue to the next statement, and would then complete in the first machine cycle. This is not allowed. All lcells must complete their LPL programs in the same cycle. The application symbol could prevent this from happening by performing a receive on the second message wave (which occurs in the second cycle) after executing the **smanage,** but the use of **fork #1** is simpler and requires less code space.

2.4.1.5. Length

Backus defines the result of applying the FP *length* operator to an argument, x, as shown. The LPL program follows.

Def
length $: x \equiv (x=<x_1, \ldots, x_n>) \rightarrow n; x=\varphi \rightarrow 0; \perp$

```
LENGTH -- result is the number of elements of the argument
Restrictions: argument is a sequence

      Summary of Analytic Model Parameters:
             program size: 42
             cycles required: 1
             cycle1: messages: 1 (wave=1; msize=0)
                    forks: 0
```

sym:	(1 < 1 2 < 3 4
aln:	0 1 1 2 2 2 3 3
dir:	0 1 2 2 2 2 2 2
	0 0 0 1 2 3 3 3
	0 0 0 0 0 0 1 2
	0 0 0 0 0 0 0 0
sym:	3
aln	0

```
Method: Argument symbols send d2 using select max as the combine-op.
        The winning message holds the length of the argument, and
        this result is placed by the application symbol.

program 001
        destination 0 0 0 0             /* Hold the result
               keep
               receive #1
                      mov r_arg1 nsymbol
                      endfilter
               endsegment
        destination 0* 0* 0* 0*         /* find max column index
               mov d2 m_arg1
               send #1 + > #0 #0 1      /* as the maximum d2 value
                      endfilter
               endsegment
        endprogram
```

2.4.1.6. Tail

Backus defines the result of applying the FP *tail* operator to an argument, x, as shown. The LPL program follows.

Def
tail $: x \equiv (x=<x_1>) \rightarrow \varphi;\ (x=<x_1, \ldots, x_n> \ \&\ n \geq 2) \rightarrow <x_2, \ldots, x_n>;\ \bot$

```
TAIL -- result is the tail of the argument list
Restrictions:  the argument is a non-empty list

       Summary of Analytic Model Parameters:
               total prog size: 39
               cycles required: 1
               cycle1: messages: none
                       forks: none
```

```
sym:     ( 24 < a b c
aln:     0 1  1 2 2 2
```

```
dir:     0 1  2 2 2 2
         0 0  0 1 2 3
         0 0  0 0 0 0
         0 0  0 0 0 0
```

```
sym:          <   b c
aln           0   1 1
```

```
Method: The tail and sequence symbol of the argument lift themselves
        one level. All other symbols go away.

program 024
        destination 2* 0* 0* 0*          /* The argument list
                cmp #1 d2 = .
                br + 1
                nselect #1 symbol #-1 .  /* remain if not first element
                label 1
                endsegment
        destination 0* 0* 0* 0*          /* Everybody else goes away.
                endsegment
        endprogram
```

2.4.1.7. Rotr

Backus defines the result of applying the FP *rotr* operator to an argument, x, as shown. The LPL program follows.

Def
$rotr : x \equiv x=\varphi \rightarrow \varphi;\ x=<x_1> \rightarrow <x_1>;$
$\qquad (x=<x_1, \ldots, x_n> \;\&\; n \geq 2) \rightarrow <x_n, x_1, \ldots, x_{n-1}>; \perp$

```
ROTR -- move rightmost argument element to leftmost position
Restrictions: argument is a list

       Summary of Analytic Model Parameters:
       n = # list elements
       m = size of rightmost element (to be moved)
               total prog size: 143
               cycles required: 2
               cycle1: messages: n (wave=1; msize=1)
                       forks: one symbol forks m+1 (executing)
               cycle2: messages: m (wave=2; msize=1)
                       forks: none
```

```
sym:     ( 26 <     a b c < d
aln:     0 1  1     2 2 2 2 3

dir:     0 1  2     2 2 2 2 2
         0 0  0     1 2 3 4 4
         0 0  0     0 0 0 0 1
         0 0  0     0 0 0 0 0

nsym:         < < d a b c
naln          0 1 2 1 1 1
```

```
Method: wave 1: find rightmost argument element, and its size. Then
        the argument sequence symbol forks to receive it. wave 2:
        the rightmost argument element sends itself and erases itself.

program 026 /* rotate right
        destination 2 1* 0* 0*              /* each arg list element
                nselect #1 symbol #-1 .     /* assume not rightmost for now
                mov #1 m_arg1               /* each element counts itself
                send #1 - + d2 #0 1         /* and rightmost arrives last
                        mov r_key1 t1
                        endfilter
                smanage
                cmp d2 t1 = .               /* am I on the right?
```

```
          br + 1                         /* if so, need to erase and send
          receive #2                     /* otherwise, sync and complete
                  endfilter
          br . 2
          label 1 erase                  /* come here iff rightmost
          mov naln m_arg1
          send #2 - . symbol_index nsymbol 1
                  endfilter
          label 2 endsegment
destination 2 0 0 0                      /* the arg seq sym
          nselect #1 symbol #-1 .        /* finds out size of rightmost
          receive #1                     /* list element
                  mov r_arg1 t1
                  endfilter
          add #1 t1                      /* must include self as well
          fork t1                        /* and then forks to receive it
          cmp #1 fork_id  = .
          br + 1
          mov #2 t1
          receive #2                     /* on wave 2
                  cmp fork_id t1 = .
                  br - 3
                  mov r_key2 nsymbol
                  mov r_arg1 naln
                  label 3 add #1 t1
                  endfilter
          label 1 endsegment
destination 0* 0* 0* 0*                  /* everybody else
          endsend
          fork #1                        /* sync and go away
          endsegment
endprogram
```

To do ROTR, we first need to locate the rightmost argument element and count it, so that it may be sent over to the left of the sequence, where the argument sequence symbol will fork and receive it. Although the LPL symbol representation does not indicate when a symbol is the rightmost element of a list, the argument symbols make effective use of their first **send** statement to both discover the rightmost element, and count it. Using *d2* as a sort key, message wave 1 returns messages sent by the rightmost element last, and this information is the desired count since addition is used for a combine operation. After this, execution is similar to that for DBL.

2.4.1.8. Distl

Backus defines the result of applying the FP *distl* operator to an argument, x, as shown. The LPL program follows.

Def
$distl : x \equiv x{=}{<}x_1,\varphi{>} \rightarrow \varphi;\ x{=}{<}x,\ {<}y_1,\ \ldots,\ y_n{>}{>} \rightarrow {<}{<}x,y_1{>},\ \ldots,\ {<}x,y_n{>}{>};\ \bot$

```
DISTL -- distribute left element to all right elements
Restrictions: argument is a pair
              whose second element is a sequence

     Summary of Analytic Model Parameters:
     n = # elements of inner list
     m = size of element to be distributed
           total prog size: 190
           cycles required: 2
           cycle1: messages: 1 (wave=1; msize=1)
                 forks nm + 1 (executing)
           cycle2: messages: m (wave=2; msize=1)
                 forks: 0
```

```
sym:     ( 25 <   a < b c
aln:     0 1  1   2 2 3 3

dir:     0 1  2   2 2 2 2
         0 0  0   1 2 2 2
         0 0  0   0 0 1 2
         0 0  0   0 0 0 0

nsym:         < < a   b < a c
naln:         0 1 2   2 1 2 2
```

```
Method: The arg seq forks to "< <", the inner list seq goes away. The
        leftmost arg element counts itself, and all but leftmost inner
        elements fork and receive it.

program 025 /* distribute left
        destination 2 0 0 0      /* the arg seq forks to "< <"
                endsend          /* allow message wave 1 to complete
                fork #2          /* maintain sync, and allow sm
                nselect fork_id "< #-1 "< #0 .
                endsegment
        destination 2 1 0* 0*  /* leftmost arg counts and sends itself
                keep      /* keep to go with leftmost elem of inner list
                mov #1 m_arg1
        send #1 + + #0 #0 1        /* count self
                endfilter
        smanage                    /* allow sm
        mov naln m_arg1
        send #2 - . symbol_index nsymbol 1 /* send self
                endfilter
        endsegment
destination 2 2 2* 0      /* all but leftmost elem of inner list
        keep
        receive #1          /* get count for new elem
                mov r_arg1 t1
                endfilter
        add #2 t1           /* must also include self and seq sym
        fork t1             /* create necessary space
        cmp fork_id t1 < .  /* am I old or new symbol?
        br + 1              /* if new, go get loaded
        nselect #1 symbol #-1 .  /* otherwise keep old
        br . 2
        label 1 cmp #1 fork_id < .  /* am I right of seq?
        br + 3                      /* if so, go get loaded
        nselect #1 "< #-2 .         /* otherwise become seq
        br . 4
        label 3 mov #2 t1   /* msg counter (offset for cmp forkid)
        receive #2
                cmp fork_id t1 = .      /* should I receive this?
                br - 5
                mov r_key2 nsymbol      /* if so, load it
                mov r_arg1 naln
                label 5 add #1 t1       /* count msg
                endfilter
        label 2 label 4 endsegment
destination 2 2 0 0       /* the separator seq goes away
        endsend             /* allow wave 1 to complete
        fork #1             /* maintain sync, allow sm
        endsegment
destination  2 0* 0* 0*  /* the rest of the arg symbols
        endsend             /* allow wave 1 to complete
        nselect #1 symbol #-1 .
        fork #1             /* maintain sync, allow sm
        endsegment
destination 0* 0* 0* 0*  /* everybody else goes away
        endsend             /* allow wave 1
        fork #1             /* maintain sync, allow sm
        endsegment
endprogram
```

In this program, the first cycle is used to determine the size of the leftmost argument element, and to fork the leftmost symbol of all but the first element of the second element of the argument. Many of the segments use **fork #1** to maintain synchronization and complete in the second cycle. These segments must also use **endsend** to allow the first message wave to complete. The first segment could have used a **cselect** and **forkc #2** in the second cycle.

2.4.1.9. Matrix Transpose

Backus defines the result of applying the FP τ operator (i.e., *transpose)* to an argument, x, as shown. The LPL program follows.

Def

$\tau : x \equiv x=\langle\varphi, \ldots, \varphi\rangle\rightarrow\varphi; x=\langle x_1, \ldots, x_n\rangle\rightarrow\langle y_1, \ldots, y_m\rangle; \perp$

where $x_i=\langle x_{i1}, \ldots, x_{im}\rangle$, *and* $y_j=\langle x_{1j}, \ldots, x_{nj}\rangle$,
with $1 \leq i \leq n$, *and* $1 \leq j \leq m$.

```
TRANS -- Transpose 2-D Rectangular Matrix
Restrictions: matrix elements are atomic

        Summary of Analytic Model Parameters:
        m = #rows
        n = #columns
        h = log(m(n+1)+3)
                  program size: 195
                  cycles required: 2
                  cycle1: messages: 1 (wave=1; msize=1)
                                forks: n(m+1)
                  cycle2: messages: m(n-1) (wave=2; msize=1)
                                forks: 0

sym:     ( 11 < < 1    2    3    < 5 6 7
aln:     0 1  1 2 3    3    3    2 3 3 3

dir:     0 1  2 2 2    2    2    2 2 2 2
         0 0  0 1 1    1    1    2 2 2 2
         0 0  0 0 1    2    3    0 1 2 3
         0 0  0 0 0    0    0    0 0 0 0

nsym:         <   < 1 5 < 2 6 < 3 7
naln:         0   1 2 2 1 2 2 1 2 2
```

```
Method: each element of the first row forms a new row with the required
        number of columns (by forking the required number of symbols).

program 011
        destination 0 0 0 0        /* the app symbol goes away
                endsend            /* allow wave 1 to complete
                fork #1            /* allow sm and synch
                endsegment
```

```
destination 1 0 0 0       /* the operator can free space at
        fork #0           /* the end of the first cycle
        endsegment
destination 2 0 0 0       /* the matrix arg seq symbol
        keep
        add #-1 naln      /* adjust nesting
        endsend           /* let wave 1 go
        fork #1           /* allow sm and synch
        endsegment
destination 2 1* 0 0      /* the seq sym for each row
        mov #1 m_arg1     /* assists in a row count
        send #1 + + #0 #0 1
                endfilter
        fork #0           /* and then vanishes
        endsegment
destination 2 1 1* 0      /* each element of first row
        keep
        mov #1 t1         /* forks a sequence symbol
        receive #1        /* plus the received # cols
                add r_arg1 t1
                endfilter
        fork t1           /* fork out required space
        cmp #1 fork_id = .        /* for complete row
        br - 1                    /* of the result.
        nselect fork_id "< #-2 . /* the seq that starts
        br . 5                    /* a row is now done
        label 1 add #-1 naln/* other cells modify their nesting
        cmp #2 fork_id = .  /* first element of row is
        br + 6              /* is the original symbol
        mov #1 t3 /* result row counter
        mov #3 t5 /* result column counter (offset)
        receive #2
                cmp d3 t3 = .          /* is this row for me?
                br - 2
                cmp  fork_id t5 = .  /* is this column for me?
                br - 3
                mov r_arg1 nsymbol   /* yes, so place it
                label 2 label 3 add #1 t5  /* increment column
                cmp t5 t1 > .     /* time to start new row?
                br - 4
                mov #3 t5
                add #1 t3
                label 4 endfilter
        label 5 label 6 endsegment
destination 0* 0* 0* 0*          /* elements of all other rows
        mov symbol m_arg1
        smanage                   /* allow sm
        send #2 - . d3 d2 1       /* send themselves
                endfilter
        endsegment
endprogram
```

2.4.1.10. N-ary Add

```
N-ARY ADD  -- result is sum of the argument elements
Restrictions: argument is a sequence whose elements are atomic

      Summary of Analytic Model Parameters:
             program size: 53
             cycles required: 1
             cycle1: messages: 1 (wave=1; msize=1)
                    forks:  0
```

sym:	(4 < a b c
aln:	0 1 1 2 2 2
dir:	0 1 2 2 2 2
	0 0 0 1 2 3
	0 0 0 0 0 0
	0 0 0 0 0 0
nsym:	a+b+c
naln:	0

Method: Each element of the list sends itself, and is added on the way up the tree. The sum is returned to the lcells, and is placed by the application symbol.

```
program 004
       destination 0 0 0 0                /* app symbol holds result
              keep
              receive #1
                    mov r_arg1 nsymbol
                    endfilter
              endsegment
       destination 1* 0 0 0               /* Op and arg seq go away
              endsegment
       destination 0* 0* 0* 0*            /* The arg elements
              mov  symbol m_arg1
              send #1 + + #0 #0 1         /* Send themselves with add op
                    endfilter
              endsegment
       endprogram
```

2.4.1.11. Sort

```
SORT -- elements of argument are sorted in ascending order
Restrictions: argument is a sequence whose elements are atomic

      Summary of Analytic Model Parameters:
              n = #elements to be sorted
              program size:   59
              cycles required: 1
              cycle1: messages: n (wave=1; msize=0)
                      forks: 0
```

```
sym:    ( 5 < 2 4 1 5 3
aln:    0 1 1 2 2 2 2 2
```

```
dir:    0 1 2 2 2 2 2 2
        0 0 0 1 2 3 4 5
        0 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0
```

```
nsym:       < 1 2 3 4 5
naln:       0 1 1 1 1 1
```

Method: The argument sequence symbol is kept in place. The application symbol and the operator erase themselves. The argument elements send themselves, with ordering so that the smallest values are passed through first.

```
program 005
        destination 2 0 0 0                 /* "<" stays in front
                keep
                endsegment
        destination 0* 0 0 0                /* The app and op syms
                endsegment                  /* go away
        destination 0* 0* 0* 0*             /* The argument sorts itself
                keep                        /* in place.
                mov #3 t1    /* msg counter (with offset for cmp index)
                send #1 - . symbol #0 0             /* symbol is the key
                        cmp t1 symbol_index = .     /* so they are received
                        br - 1
                        nselect #1 r_key1 #-1 .     /* in order
                        label 1
                        add #1 t1
                        endfilter
                endsegment
        endprogram
```

2.4.1.12. Matrix Multiply

```
MM -- In-place multiplication of matrices A x B → C.
Restrictions: argument is a pair of square matrices

      Summary of Analytic Model Parameters:
      n = # of rows and columns of A,B, and C.
             total prog size: 246
             cycles required: 1
             cycle1: messages: 1   (wave=1; msize=1)
                               n^2 (wave=2; msize=2)
                                      ...
                               n^2 (wave=j+1; msize=1)
                        forks: none
```

```
sym:   ( 28 < < < 1 1 < 2 2 < < 1 2 < 1 2
aln:   0 1  1 2 3 4 4 3 4 4 2 3 4 4 3 4 4

dir:   0 1  2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
       0 0  0 1 1 1 1 1 1 1 2 2 2 2 2 2 2
       0 0  0 0 1 1 1 2 2 2 0 1 1 1 2 2 2
       0 0  0 0 0 1 2 0 1 2 0 0 1 2 0 1 2

sym:          < < 2 4 < 4 8
aln:          0 1 2 2 1 2 2
```

Method: The application symbol and the operator go away. The result is held in A, the first argument matrix. The second argument matrix, B, goes away after assisting the multiplication. The outer-product algorithm is used, which operates as follows: $C(i,j) = \Sigma_k(A(i,k) * B(k,j))$.

Matrix B finds max k in wave 1,
sends row k in wave 1 + k.
Matrix A gets max k in wave 1,
sends column k in wave 1 + k.
During each wave, A(i,j) multiplies and accumulates a result based on a msg from A (key1=d3; key2=0) and a msg from B (key1=d4; key2=1).

```
program 028
        destination 2 2 1* 1*          /* -- elements of B --
                mov d3 m_arg1          /* mov row index to message
                send #1 + > #0 #0 1    /* send with select max
                        mov r_arg1 t9  /* last msg is the row count
                        endfilter
                mov #1 t8              /* init current row
                mov #2 t7              /* init current wave
        label 1 cmp d3 t8 = .      /* am I right row to send?
        br - 2                     /* if not, go receive
        mov symbol m_arg1          /* move symbol to message
        send t7 + . d4 #1 1        /* include col index and B flag
                endfilter
        br . 3
        label 2 receive t7         /* if not send, keep step
                endfilter
        label 3 add #1 t7          /* new wave
        add #1 t8                  /* new row
        cmp t8 t9 <= .             /* more sends?
        br + 1                     /* yes if more rows
        endsegment                 /* go away when finished
destination 2 1 1* 1*              /*  -- elements of A->C --
        receive #1                 /* get
                mov r_arg1 t9      /* max k
                endfilter
        mov #1 t8                  /* init column counter
        mov #2 t7                  /* init wave counter
        mov #0 t6                  /* init accumulator
        label 1 mov #0 t4          /* init t4 and t5 to
        mov #0 t5                  /* hold values to multiply
        cmp d4 t8 = .              /* am I right column to send?
        br - 2                     /* if not go receive
        mov symbol m_arg1          /* move symbol to message
        send t7 + . d3 #0 1        /* include row index and A flag
                label 9 cmp #0 r_key2 = . /* Which matrix?
                br + 4             /* A -- go to handle
                cmp r_key1 d4 = . /* B -- check for key1=d4
                br - 5             /* no
                mov r_arg1 t5      /* hold B value
                br . 6
                label 4 cmp r_key1 d3 = . /* A -- check for key1=d3
                br - 7             /* no
                mov r_arg1 t4      /* hold A value
                label 5 label 6 label 7 endfilter
        br . 8
        label 2 receive t7         /* either send or receive
                br . 9             /* both are handled above
                endfilter
        label 8 mul t4 t5          /* wave t7 is now over
        add t5 t6                  /* add B contribution to accum
        add #1 t7                  /* count message wave
        add #1 t8                  /* and increment column
        cmp t8 t9 <= .             /* is there more work?
        br + 1                     /* if so, get next wave
        nselect #1 t6 #-2 .        /* otherwise place accumulator
        endsegment
destination 0* 0* 0* 0*            /* everybody else -- includes
        cmp #2 d1 = .              /* enclosing seq syms of A, B
        cmp #1 d2 = *              /* do I enclose A?
        br - 1                     /* if not, go away
        nselect #1 symbol #-2 .    /* otherwise modify nesting
```

```
          label 1 endsegment
endprogram
```

2.4.2. FFP Functional Forms

A *functional form* is a parametrized function. For example, <CONST n> is the functional form used in FFP to represent the function whose application always reduces to n, regardless of its argument. As the example definitions for ρ(CONST) given in Section 2.2.2 showed, evaluation of the application of a functional form within FFP uses meta-composition.

Backus's definition of FFP in this way was motivated by the desire for a concise, uniform representation of self-referential functions. Within the operational context of the DOT implementation, however, there is no need for meta-composition (whose purpose is to provide an operator with access to itself as well as the original argument). This is because LPL definitions for FPP operators *always* have access to an entire RA, and this includes the operator expression. Functional forms are therefore implemented directly within LPL without the intermediate step (and extra reduction cycle) implied by meta-composition. This is done for all the usual functional forms which occur in the form "< ff ... >".

Within Backus' formal semantics for FFP an operator can meaningfully occur in the form "<< ff ... > ... >" or with even deeper "leftmost" nesting. Such operator expressions can be created within FFP, and a complete semantics must provide a definition of their meaning. The result of application of such an operator could be defined as bottom without sacrifice of computational power, but the meta-composition rule instead handles such an application by unraveling the operator as usual. For this reason, the DOT implementation knows about meta-composition, and, upon encountering an operator with at least two leading sequence symbols, brings in an LPL program that implements Backus' meta-composition rule. The LPL program for meta-composition is therefore given in this section for completeness.

One of the main differences between FP and FFP is the treatment of functional forms. In FFP, each functional form is uniformly represented as a sequence with a controlling operator. In FP, on the other hand, functional forms represent operations of an associated algebra of programs, and their representation varies in the interest of clarity and notational convenience.

2.4.2.1. Constant

Using $\overline{x}$ to represent the FP function whose value is always the object, x, Backus gives

Def
$\overline{x} : y \equiv y \neq \bot \rightarrow x; \bot$

```
< CONST n > -- result is the object, n
Restrictions: none

        Summary of Analytic Model Parameters:
                program size: 27
                cycles required: 1
                cycle1: messages: 0
                        forks: 0
```

```
sym:    ( < 21 n x
aln:    0 1 2  2 1
```

```
dir:    0 1 1  1 2
        0 0 1  2 0
        0 0 0  0 0
        0 0 0  0 0
```

```
nsym:          n
naln:          0
```

```
Method: Erase everything but the object parameter of CONST

program 021
        destination  1 2 0 0
                nselect #1 symbol #-2 .
                endsegment
        destination 0* 0* 0* 0*
                endsegment
        endprogram
```

2.4.2.2. Select

Backus considers selectors to be special FP functions [Bac78], but we prefer to treat *select* as a functional form. Using $\hat{s}$ to represent the FP function that selects the sth element of an argument sequence, we give

Def
$\hat{s} : x \equiv (x=<x_1, \ldots, x_n> \ \& \ 1 \leq s \leq n) \rightarrow x_s ; \perp$

```
<SELECT n> -- result is the nth element of the argument sequence
Restrictions: argument is a sequence

      Summary of Analytic Model Parameters:
              total prog size: 61
              cycles required: 1
              cycle1: messages: 1 (wave=1; size=0)
                      forks: none
```

```
sym:     ( < 27 2 < a < b c
aln:     0 1 2  2 1 2 2 3 2
```

```
dir:     0 1 1  1 2 2 2 2 2
         0 0 1  2 0 1 2 2 3
         0 0 0  0 0 0 0 1 0
         0 0 0  0 0 0 0 0 0
```

```
nsym:              < b
naln:              0 1
```

```
Method: The selector value is sent in a message, and each argument
        symbol compares its d2 directory value with the selector.
        All symbols that are not part of the argument, or whose
        d2 value does not equal the selector value are erased. Those
        symbols that remain adjust their nesting by raising
        themselves two levels.

program 027 /* Select
        destination 1 2 0 0                 /* the selector value
                send #1 + . symbol #0 0 /* sends itself.
                        endfilter
                endsegment
        destination 2 0* 0* 0*              /* The argument receives
                receive #1                  /* the selector value
                        cmp r_key1 d2 = .
                        endfilter
```

```
          br - 1
          nselect #1 symbol #-2 .  /* and keeps itself if selected
          label 1 endsegment
destination 0* 0* 0* 0*            /* Everybody else goes away.
          endsegment
endprogram
```

2.4.2.3. Composition

Within FP, functional composition is represented using @ as an infix operator. Thus Backus gives

Def
$f@g : x \equiv f : (g : x).$

We give an n-ary FFP functional form for composition, for which an appropriate FP form might be defined as follows.

$$@(f_1, \ldots, f_n) : x \equiv f_1 : (f_2 : (\ldots (f_n : x) \ldots)).$$

<COMP f_1 f_2 ... f_n> -- result is the desired composition of f's
Restrictions: none

```
        Summary of Analytic Model Parameters:
        n = # of fns to be composed
                program size: 103
                cycles required: 1
                cycle1: messages: 1 (wave=1; msize=1)
                        forks: 2n  (completed)
```

```
sym:      (  <  20 f     <     g  h     x
aln:      0  1  2  2     2     3  2     1

dir:      0  1  1  1     1     1  1     2
          0  0  1  2     3     3  4     0
          0  0  0  0     0     1  0     0
          0  0  0  0     0     0  0     0

nsym:              (  f  (  <  g  (  h  x
naln:              0  1  1  2  3  2  3  3
```

```
Method: Cselect is used to let the first symbol of each function
        create an application. The functions count themselves by
        sending d2 up with select maximum as the combine-op. This
        result counts the COMP controlling operator (which is not
        wanted) so for what follows, cnt=result-1.
                Each arg gets naln=aln+cnt-1
                The app symbols get naln=aln-4+d2
                The fcn symbols get naln=aln-3+d2

program 020
        destination 2* 0* 0* 0*          /* the arg text
        keep                            /* the argument remains
        receive #1
                add r_arg1 naln
                endfilter
        sub #2 naln                     /* but with modified nesting
        endsegment
destination 1 2* 0 0                    /* first part of each fcn
        mov d2 m_arg1
        send #1 + > #0 #0 1             /* determine # of functions
                endfilter
        mov d2 t1
        sub #4 t1                       /* determine nesting for
        mov t1 t2                       /* the new application
        add #1 t2                       /* and this symbol
        cselect "( t1 symbol t2 .  /* cselect
        forkc #2                        /* and forkc to result
        endsegment
destination 1 2* 0* 0*                  /* rest of each fcn
        mov d2 t1                       /* remains
        sub #3 t1                       /* but with modified nesting
        nselect #1 symbol t1 .
        endsegment
destination 0* 0* 0* 0*                 /* ( < comp
        endsegment                      /* these go away
endprogram
```

2.4.2.4. Construction

In FP, *construction* is an important way of using parallelism to create a list. Each element of the list that is constructed by this functional form is created by application of a separate function. Backus gives the following FP definition.

Def
$[f_1, \ldots, f_n] : x \equiv < f_1 : x, f_2 : x, \ldots, f_n : x >$

```
<CONS f1 f2 ... fn> -- result is the list constructed by the f's
Restrictions:  none

      Summary of Analytic Model Parameters:
      n = # functions
      m = argsize
             program size: 175
             cycles required: 2
             cycle1: messages: 1 (wave=1; msize=1)
                    forks: 3n + m(n-1) executing
             cycle2: messages: m (wave=2; msize=1)
                     forks: 0

sym:    ( < 22 <    f g     x
aln:    0 1 2  2    3 2     1

dir:    0 1 1  1    1 1     2
        0 0 1  2    2 3     0
        0 0 0  0    1 0     0
        0 0 0  0    0 0     0

nsym:          < ( < f x ( g x
naln:          0 1 2 3 2 1 2 2
```

```
Method: cycle1: wave 1: find arg size
                each fcn but the first forks to "arg ( f"
                the first fcn forks to "< ( f"
        cycle2: wave 2: the argument sends itself

program 022 /* Construction
        destination 2 0* 0* 0*              /* The argument
                nselect #1 symbol #1 .      /* remains with nesting increased,
                mov #1 m_arg1               /* counts itself
                send #1 + + #0 #0 1         /* during message wave 1,
                        endfilter
                smanage                     /* permits others to fork,
          mov naln m_arg1           /* then sends itself on wave 2
          send #2 - . symbol_index symbol 1
                    endfilter
          endsegment
destination 1 2 0 0/* First sym of first fcn forks to "< (f"
          endsend            /* allow messages to proceed
          fork #3            /* then fork
          nselect fork_id "< #-2 "( #-1 symbol #0 .
          endsegment
destination 1 3* 0 0/* First sym of other fcns fork to "arg (f"
          keep
          mov #2 t1          /* offset to include "(" and f
          receive #1
                    add r_arg1 t1     /* in forksize
                    endfilter
          fork t1                     /* then fork.
          cmp fork_id t1 < .          /* if rightmost, then
          br - 1                      /* go place original f symbol
          sub #1 t1                   /* otherwise
          cmp fork_id t1 < .          /* if need to receive arg copy
          br + 2                      /* go do that.
          nselect #1 "( #-1 .         /* otherwise place "("
          br . 3
          label 2 mov #1 t1           /* init symbol counter
          receive #2
                    cmp fork_id t1 = .  /* is this symbol for me?
                    br - 5
                    mov r_key2 nsymbol  /* if so, place it
                    mov r_arg1 naln
                    label 5 add #1 t1   /* increment counter
                    endfilter
          label 1 label 3 endsegment
destination 1 2* 0* 0*   /* fcn bodies (all but leftmost symbol)
          endsend            /* allow wave 1
          fork #1            /* allow sm and sync
          keep               /* remain
          endsegment
destination 0* 0* 0* 0*  /* everybody else
          endsend            /* allow wave 1
          fork #1            /* allow sm and sync
          endsegment         /* go away
endprogram
```

2.4.1.5. Conditional

Conditional in FP is defined by Backus in the following way.

Def
$(p \rightarrow f;g) : x \equiv ((p{:}x)=T) \rightarrow f{:}x;\ ((p{:}x)=F) \rightarrow g{:}x;\ \perp$

Thus, the result of reducing an application of such a functional form depends on whether the predicate, p, reduces to true or false when applied to the argument, x. If the predicate reduces to true, then the result is an application of the function f to the argument (i.e., f : x), otherwise, if the predicate reduces to false, the result is application of the function g to the argument (i.e., g : x). If the predicate reduces to an undefined result, the result is undefined.

Reducing an RA with string reduction destroys the original expression. Conditional is thus implemented in two steps. In the first step, the argument is copied, and the original expression is restructured so a newly created application of the predicate is innermost to an application of the second phase conditional operator, COND2. Upon reduction of the predicate on its argument copy, the second phase operator checks the result of the predicate evaluation and then creates an application of the appropriate function, f or g.

We take the liberty of representing the FFP functional form with the predicate on the right. This make it easier to apply the predicate.

```
<COND f g p> -- result is COND2 with inner application of p
Restrictions: none

      Summary of Analytic Model Parameters:
      n = argsize
      m = predicate size
             program size: 221
             cycles required: 2
             cycle1: messages: m (wave=1; msize=0)
                               1 (wave=2; msize=1)
                    forks: 1 symbol forks n+1 (executing)
             cycle2: messages: n (wave=3; msize=1)
                    forks: 1 symbol forks 3 (completed)
```

```
sym:    ( < 9   f g p       x
aln:    0 1 2   2 2 2       1

dir:    0 1 1   1 1 1       2
        0 0 1   2 3 4       0
        0 0 0   0 0 0       0
        0 0 0   0 0 0       0

sym:    ( < 10 f g < ( p x x
aln     0 1 2   2 2 1 2 3 3 2
```

Method: COND works in two phases with an intermediate operator, COND2.
In the first phase (< COND2 f g > < (p x) x >) is produced.
COND2 then applies f or g depending on the value of (p x).
To do phase 1, we first count the argument size, then fork and
create the inner application of the predicate during cycle 2.

```
program 009
       destination 2* 0* 0* 0*  /* argument symbols
               keep             /* remain for later use
               mov #1 m_arg1
               send #2 + + #0 #0 1      /* count the argument symbols
               smanage                  /* allow sm for forking
               add #1 naln              /* increase nesting by 1
               mov naln m_arg1          /* all arg symbols
               send #3 - . symbol_index symbol 1 /* send themselves
               endsegment
       destination 0* 0 0 0     /* the app and and arg seq symbols
               keep             /* remain
               endsend          /* allow message waves 1 and 2
               fork #1          /* allow sm
               endsegment
       destination 1 1 0 0      /* the operator symbol (COND)
               keep
               endsend          /* allow message waves 1 and 2
```

```
          fork #1           /* allow sm
          mov #10 nsymbol   /* change operator to COND2
          endsegment
destination 1 4 0* 0*       /* the symbols of the predicate
          send #1 - . symbol_index #0 0 /* find rightmost symbol
                  mov r_key1 t1      /* last r_key1 through is from
                  endfilter          /* rightmost predicate symbol
          receive #2        /* find out number of lcells for rightmost
                  mov r_arg1 t2      /* predicate symbol to fork
                  endfilter          /* to hold the argument
          cmp symbol_index t1 = .  /* am I rightmost?
          br - 4            /* if not, go around argument copying
          keep              /* signals that these symbols are set up
          add #1 t2         /* must hold self as well as argument
          fork t2           /* fork to receive argument
          cmp #1 fork_id = .         /* true for parent
          br - 1            /* if not parent, go receive nsymbols
          add #1 naln       /* parent merely modifies nesting
          br . 3            /* and goes around argument copying
          label 1           /* come here to get nsymbols from wave 3
          mov #2 t1         /* symbol counter (offset for cmp forkid)
          receive #3
                  cmp t1 fork_id = .   /* is this nsymbol for me?
                  br - 2               /* if not, loop
                  mov r_key2 nsymbol   /* otherwise load it
                  mov r_arg1 naln
                  add #1 naln
                  label 2 add #1 t1    /* increment symbol counter
                  endfilter
          br . 5
          label 4 fork #1   /* if not rightmost, then allow sm for fork
          label 3 label 5   /* need to handle left part of predicate
          cmp #0 d3 = .     /* am I leftmost predicate symbol?
          cmp #1 fork_id = *  /* with fork_id=1?
          br - 6            /* if not, go adjust nesting and complete
          cselect "< #-1 "( #0 symbol #1 . /* otherwise, create
          forkc #3          /* innermost predicate application
          label 6 cmp #1 nsymbol_cnt = .  /* is nsymbol set up?
          br + 7                     /* if so, go complete
          nselect #1 symbol #1 .  /* otherwise, adjust nesting for
          label 7 endsegment      /* innermost symbols of predicate
destination 1 2* 0* 0*      /* all symbols of the functions f and g
          keep              /* remain
          endsend           /* allow waves 1 and 2
          fork #1           /* allow sm for forks
          endsegment        /* complete
endprogram
```

2.4.1.6. Conditional – phase 2

```
<COND2 f g > -- result is application of f or g to second element
                of the argument, depending on the first element
Restrictions: argument is a pair
              whose first element is true or false

       Summary of Analytic Model Parameters:
               program size: 121
               cycles required: 1
               cycle1: messages: 1 (wave=1; msize=0)
                       forks: 0
```

```
sym:    ( < 10 f g < t x
aln:    0 1 2  2 2 1 2 2

dir:    0 1 1  1 1 2 2 2
        0 0 1  2 3 0 1 2
        0 0 0  0 0 0 0 0
        0 0 0  0 0 0 0 0

sym:    (      f       x
aln     0      1       1
```

Method: Check t, and apply f or g as appropriate.

```
program 010
        destination 0 0 0 0     /* app sym stays
                keep
                endsegment
        destination 2 2* 0* 0*  /* arg stays
                keep
                add #-1 naln
                endsegment
        destination 2 1 0 0     /* t sends itself and is erased
                erase
                send #1 + . symbol #0 0
                endsegment
        destination 1 2 0* 0*   /* f keeps itself if t is true
                receive #1
                        cmp #1 r_key1 = .
                        br + 1
                        erase
                        br . 2
                        label 1
                        keep
                        add #-1 naln
                        label 2
                endfilter
        endsegment
destination 1 3 0* 0*    /* g keeps itself if t is false
        receive #1
                cmp #1 r_key1 = .
                br - 1
                erase
                br . 2
                label 1
                keep
                add #-1 naln
                label 2
                endfilter
        endsegment
destination 0* 0* 0* 0*
        erase
        endsegment
endprogram
```

2.4.1.7. Apply-to-all

In FP, *apply-to-all* provides a powerful means of creating parallelism. Backus defines it as follows.

Def
$\alpha f : x \equiv x{=}\varphi \rightarrow \varphi;$
$\quad x{=}{<}x_1, \ldots, x_n{>} \rightarrow {<}f_1{:}x, \ldots, f_n{:}x{>}; \perp$

```
<AA f> -- result is  a sequence of applications of the
          function, f, to the argument elements
Restrictions: argument is a sequence

       Summary of Analytic Model Parameters:
       n = # of list elements
       m = size of operator
               program size: 174
               cycles required: 2
               cycle1: messages: 1 (wave=1; msize=1)
                       forks: (n-1) symbols fork (m+1) contexts
               cycle2: messages: m (wave=2; msize=1)
                       forks: none

sym:      ( < 29 op < a b      c
aln:      0 1  2 2  1 2 2      2

dir:      0 1  1 1  2 2 2      2
          0 0  1 2  0 1 2      3
          0 0  0 0  0 0 0      0
          0 0  0 0  0 0 0      0

sym:      < (    op   a ( op b ( op c
aln       0 1    2    2 1 2  2 1 2  2
```

```
Method: In the first cycle, the operator counts itself, and the first
        symbol of each argument element (except the first element, which
        can use the original operator) uses this count to fork off
        enough symbols to hold the operator. To complete in the second
        cycle, the application symbol becomes a sequence symbol, the
        first seq. symbol becomes an app. symbol. The AA symbol erases
        itself, as does the arg seq.  The operator to be applied stays
        where it is, and also sends itself to the argument list. The
        members of the argument list receive the operator, and load
        its symbols in order of reception to create the new applications.

program 029
        destination 0 0 0 0      /* the application symbol
        endsend            /* does not send
        fork #1            /* maintain sync
        nselect #1 "< #0 .          /* become a seq symbol
        endsegment
destination 1 0 0 0        /* the op seq
        endsend            /* does not send
        fork #1            /* maintain sync
        nselect #1 "( #0 .          /* become leftmost app symbol
        endsegment
destination 1 1 0 0        /* the aa opcode can go away
        endsend            /* immediately
        fork #0            /* by vanishing between cycles
        endsegment
destination 1 2 0* 0*      /* the operator to be applied
        keep               /* keeps itself
        mov #1 m_arg1      /* and counts itself for the arg elems
        send #1 + + #0 #0 1
                endfilter
        mov aln m_arg1
        smanage            /* allow arg elems to fork
        send #2 - . symbol_index symbol 1
                endfilter
        endsegment
destination 2 0 0 0        /* The arg seq can vanish
        endsend
        fork #0
        endsegment
destination 2 2* 0 0       /* first symbol of each arg elem
        keep               /* (except the first arg elem)
        receive #1         /* get op size
                mov r_arg1 t1
                endfilter
        add #2 t1
        fork t1  /* fork to include app sym, op, and self
        mov #2 t1
        receive #2         /* receive operator
                cmp t1 fork_id = .
                br - 1
                mov r_key2 nsymbol
                mov r_arg1 naln
                label 1 add #1 t1
                endfilter
        cmp #1 fork_id = .
        br - 2
        nselect #1 "( #-1 .
        label 2 endsegment
destination 0* 0* 0* 0*
        keep     /* all other symbols keep themselves
        endsend  /* don't need to send
        fork #1  /* maintain sync
        endsegment
endprogram
```

2.4.1.8. Element-by-element

In FP, *element-by-element* creates parallelism in a way similar to *apply-to-all*, but automatically brings together corresponding elements of two lists to create arguments for a binary operator. It may be defined as follows.

Def

$$\beta f : x \equiv x{=}{<}{<}y_1, \ldots, y_n{>},{<}z_1, \ldots, z_n{>}{>} \rightarrow$$
$$<f{:}{<}y_1,z_1{>}, \ldots, f{:}{<}y_n,z_n{>}>; \perp$$

```
  <EE1 f>  -- result is a sequence of applications of f to
              paired elements from the two argument sequences
  Restrictions:  argument is a pair of sequences of equal length
                 whose elements are atomic

         Summary of Analytic Model Parameters:
         n = number of applications to be created
         h = log(2n+5) + 1
                 program size: 126
                 cycles required: 1
                 cycle1: messages: n+1 (wave=1; msize=1)
                         forks: n forking 5 (completed)

sym:     ( < 7 4 < < 1         2         3 < 4 5 6
aln:     0 1 2 2 1 2 3         3         3 2 3 3 3

dir:     0 1 1 1 2 2 2         2         2 2 2 2 2
         0 0 1 2 0 1 1         1         1 2 2 2 2
         0 0 0 0 0 0 1         2         3 0 1 2 3
         0 0 0 0 0 0 0         0         0 0 0 0 0

sym:     <           ( 4 < 1 4 ( 4 < 2 5 ( 4 < 3 6
aln      0           1 2 2 3 3 1 2 2 3 3 1 2 2 3 3
```

```
Method: The operator to be applied is sent to the first arg list,
        and the second arg list sends itself to the first arg list
        members.

program 007
        destination 1 2 0 0                 /* The atomic binary function
                send #1 - . #0 symbol 0 /* sends itself with key1=0
                        endfilter           /* to the first list of the arg
                endsegment
        destination 2 2 1* 0                /* The second list of the arg
                mov symbol m_arg1           /* also sends itself
```

```
          send #1 - . #1 d3 1        /* to the first list with key1=1
                    endfilter
          endsegment
destination 2 1 1* 0                 /* The first list of the arg
          receive #1                 /* gets fcn and list elems
                    cmp #1 r_key1 = .  /* is this a list elem?
                    br + 1             /* if so, go check
                    mov r_key2 t1  /* otherwise hold the fcn for later
                    br . 2
                    label 1 cmp r_key2 d3 = .  /* is this elem my pair?
                    br - 3           /* if not goto loop
                    mov r_arg1 t2    /* otherwise hold symbol for later
                    label 2 label 3 endfilter
          cselect "( #-2 t1 #-1 "< #-1 symbol #0 t2 #0 .
          forkc #5           /* fork to create application
          endsegment
destination 0 0 0 0                  /* The app symbol becomes seq
          nselect #1 "< #0 .
          endsegment
destination 0* 0* 0* 0*              /* Everybody else goes away.
          endsegment
endprogram
```

2.4.1.9. Meta-composition

This is the functional form corresponding to FFP meta-composition. There is no corresponding FP definition. The following LPL program correctly implements metacomposition for all FFP functional forms, but is only used when the controlling operator is a sequence.

```
< f ... > -- result is a new application as defined by the
             FFP rule for meta-composition
Restrictions:  none (but only used when f is a sequence)

       Summary of Analytic Model Parameters:
       n = size of operator
               program size: 120
               cycles required: 2
               cycle1: messages: 1 (wave=1; msize=1)
                       forks: n (executing)
               cycle2: messages: n (wave=2; msize=1)
                       forks: 0

sym:    (             < < < a b c d x
aln:    0             1 2 3 4 4 3 2 1

dir:    0             1 1 1 1 1 1 1 2
        0             0 1 1 1 1 1 2 0
        0             0 0 1 1 1 2 0 0
        0             0 0 0 1 2 0 0 0

nsym:   ( < < a b c < < < < a b c d x
naln:   0 1 2 3 3 2 1 2 3 4 5 5 4 3 2
```

```
Method: In the first cycle, the number of symbols of the controlling
        operator are counted.  The application symbol forks off
        enough lcells to hold itself, the controlling operator, and
        a sequence symbol (in left-right order). This sequence symbol
        is used to enclose the application operator (i.e., the
        functional form) and the application argument as required by
        the rule for meta-composition. In the second cycle, the
        controlling operator sends itself to the forked application
        symbols, where it is received and placed. Original operator
        and argument symbols increase their nesting level by 1.

program 060
        destination 0 0 0 0             /* the application symbol
                keep                    /* remains
                receive #1              /* get size of controlling
                mov r_arg1 t1       /* operator of the functional
                endfilter           /* form
        add #2 t1                   /* must include self and seq sym
        fork t1                     /* make room for ( op <
        cmp fork_id t1 = .          /* should I be the seq sym?
        br - 1                      /* if not, go receive op
        nselect #1 "< #1 .          /* else place seq sym
        label 1
        receive #2                  /* receive controlling operator
                cmp fork_id r_key1 = .   /* this symbol for me?
                br - 2    /* go around unless need this symbol
                mov r_key2 nsymbol       /* load symbol
                mov r_arg1 naln          /* and aln
                label 2
                endfilter
        endsegment
destination 1 1 0* 0*               /* the controlling operator
        keep                        /* remains
        mov #1 m_arg1               /* counts itself
        send #1 + + #0 #0 1
                endfilter
        smanage                     /* allow app sym to fork
        mov aln m_arg1              /* the controlling operator
        sub #1 m_arg1               /* is lifted one level
        send #2 + . symbol_index symbol 1 /* for sending
                endfilter
        add #1 naln                 /* and is nested an additional
        endsegment                  /* level locally
destination 0* 0* 0* 0*  /* rest of operator, and argument
        keep             /* remain
        endsend          /* let message wave 1 go
        fork #1          /* allow sm for forking
        add #1 naln      /* increase nesting level
        endsegment
endprogram
```

CHAPTER 3

Implementation — The DOT Model

3.1. Introduction

3.1.1. What DOT is (and what it isn't)

In the preceding chapter, the FFP and LPL languages were defined and LPL was used in the capacity of ρ, the FFP representation function, to define a variety of FFP primitive operators. As indicated at that time, the purpose of DOT is to:

(1) locate innermost applications of FFP operators and reduce them using the appropriate LPL operator definition, and

(2) provide a model informally suggestive of an actual realization for DOT as a tree-structured network of cellular processors.

DOT is not an architecture; there is no machine language associated with detailed control of its operation. DOT is an implementation in exactly the sense Blaauw and Brooks [Bla83] suggest -- it is a description of the logical organization of data flow and control utilized to support the LPL and FFP architectures. The reason why DOT is able to additionally suggest a realization is that DOT objectifies the means of data flow and control through the use of abstract data types that correspond to realizable entities.

Objects of the DOT implementation model include lcell and tcell classes. These contain processes and represent the cells of an anticipated multiprocessor realization. Io and virtual memory classes represent the "outside" world, and communication channel classes represent the means of communication between cells. Thus, DOT uses communication channels to specify the tree-structured linkage of its cellular objects, and in addition to its overall cooperative function as an implementation, this interconnection of objects naturally suggests high-level aspects pertaining to realization. Although descriptive of a realization in this way, DOT is not a realization; it does not specify detailed hardware design.

DOT was originally conceived as an attempt to represent a design concept whose scope includes a spectrum of concerns from architecture to realization [Mag79]. Indeed, dividing this spectrum up into separate pieces (architecture-implementation-realization) is an abstraction only vindicated historically by the successes and flexibility in computer system design that it has enabled. In this case, however, an original and revolutionary design concept was made possible by an all-embracing concern for the complete spectrum -- from a highly-parallel realization able to make effective use of the replication-based technology of VLSI, to a general-purpose architecturally-concurrent programming language whose implementation would make effective use of the realization. It therefore seemed desirable to encompass as much of this overall concept as possible in one unified framework.

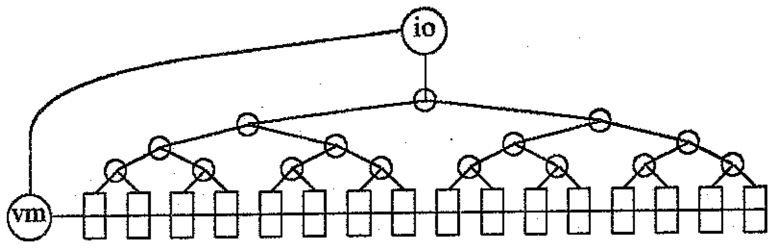
DOT is moderately successful in encompassing a complete design concept. LPL, although only in its compiled form, is implied by the LPL interpreter process. FFP is implied by DOT's embodiment of an evaluation function for μ, and, as already indicated, DOT suggests high-level aspects of a parallel realization. For these reasons, we feel justified in using the term "DOT" to refer to the complete programming system, and in speaking of a "DOT machine". But regardless of this larger and implicit function, DOT is formally only an implementation model.

As mentioned earlier, the DOT representation is executable. This fact will not concern us in this chapter; aspects relating to execution are left for Chapter 4 on simulation.

3.1.2. Overall DOT Structure

Figure 3.1 shows the overall structure of the DOT model. Links between cells represent point-to-point communication buses. The io and vm nodes represent the "world" external to the main tree of processors. Within the main tree structure, leaf nodes represent the lcell processors that appear in the LPL architecture, and the internal nodes represent processing cells called *tcells* (for tree cells) that are used to implement the LPL message subsystem, and perform functions related to μ.

FIGURE 3.1 – Overall DOT Structure



3.1.3. A Language for Representing DOT

Corresponding to Figure 3.1 is a textual description of DOT in the language we have chosen as a representation language. This language is C augmented with abstract data types [Str83], or *ClassC*, as we will refer to it. Figure 3.2 shows an abbreviated top-level ClassC representation for the DOT machine. In this and following figures that display DOT code, a rudimentary familiarity with the C programming language [Ker78] and the concept of classes or abstract data types [Fra77, Han77, Str82] is assumed.

As seen from Figure 3.2, a ClassC class definition can be viewed as a list of objects, a way of putting these components together into a new object, and (optionally) a specification of operations that are appropriate to the new object. The *new* entry point for a class describes how a new object of the type being defined is created, and other *public* entry points (not used in the DOT_machine class) describe the allowable operations on the new object. At lower levels of detail than depicted so far, the processes that actually move data around in the machine become visible. The tree of processing cells referred to in Figure 3.2 is represented in ClassC as shown in Figure 3.3.

FIGURE 3.2 – The DOT Machine

```
class DOT_machine
{
          /* declare the objects that make up a DOT machine */
          class     io        *io;
          class     vm        *vm;
          class     tree      *tree;
          class     e_bus     *io_vm_comm;
          class     t_bus     *io_tree_comm;
          class     l_bus     *vm_tree_comm;

          /* say how a new DOT_machine is built */
          DOT_machine.new(tree_height)
          int tree-height;     /* the height of the processor tree */
          {
                    /* build the communication buses */
                    io_tree_comm = new class t_bus();
                    vm_tree_comm = new class l_bus();
                    io_vm_comm = new class e_bus();

                    /* build and connect the machine */
                    io = new class io(io_tree_comm, io_vm_comm);
                    vm = new class vm(vm_tree_comm, io_vm_comm);
                    tree = new class tree
                              (tree_height, io_tree_comm, vm_tree_comm);
          } /* end new DOT_machine */
}
```

FIGURE 3.3 – The DOT Tree of Processing Cells

```
class tree
{
        /* declare the objects that make up a tree */
        class    tcell     *root;
        class    tree      *l_subtree, *r_subtree;
        class    lcell     *l_lcell, *r_lcell;
        class    t_bus     *to_left, *to_right;
        class    l_bus     *conn_l_with_r;

        /* say how a tree is built */
        tree.new(level,to_parent,on_left,on_right)
        int level;                              /* level of this tree root */
        class t_bus *to_parent;                 /* connection to parent */
        class l_bus *on_left, *on_right; /* connections to l/r lcell */
        {                                       /* boundaries at tree base */
                /* build communication links */
                to_left = new class t_bus();
                to_right = new class t_bus();
                conn_l_with_r = new class l_bus();

                /* build tree root, and its children */
                root = new class tcell(to_parent, to_left, to_right);
                if (level==1)
                {       /* use lcells for children */
                        l_lcell = new class lcell
                                (to_left, on_left, conn_l_with_r);
                        r_lcell = new class lcell
                                (to_right, conn_l_with_r, on_right);
                }
                else
                {       /* use trees for children */
                        l_subtree = new class tree
                          (level-1, to_left, on_left, conn_l_with_r);
                        r_subtree = new class tree
                          (level-1, to_right, conn_l_with_r, on_right);
                }
        } /* end new tree */
}
```

So far, we have shown how the DOT design is decomposed into the following major components:

(1) Processing Cells
- *io*
- *vm*
- *lcell*
- *tcell*

(2) Communication Buses

• *external bus*	*(e_bus used for comm between io and vm)*
• *tree bus*	*(t_bus -- any comm involving tcell)*
• *lcell bus*	*(l_bus -- any comm at lcell level only)*

(3) Explicit Connections between (1) and (2)
- *represented as parameters of processing cell classes*

In the remainder of this chapter, we will show how the above components function together as an implementation of the LPL and FFP architectures.

3.1.4. A Process-Oriented Design

Recall that an implementation should specify the control and flow of data. To this end, DOT uses multiple processes within each processing cell to control the flow of data on communication buses. By using multiple processes within each cell, DOT avoids overly constraining a VLSI realization, and succeeds in stating, in a concise and intellectually manageable manner, just what control and data flow is necessary in an efficient implementation. Representing an efficient implementation involves a great deal of complexity. In efficient support of both FFP and LPL, DOT must do a great many things (many of which are only peripherally related) at the same time.

To describe the behavior of the implementation, DOT takes a *process-oriented design* approach. By this, we mean that underlying all data movement and manipulation are individual, relatively simple processes, each one of which is designed to perform a specific and easily grasped *sequential* task exhibiting conceptual integrity within its limited scope of concern. The processes of the DOT model exist statically; they are not created dynamically except as the machine is brought into existence initially, and they exhibit cyclic behavior. Processes within DOT never wait on non-deterministic events, and a wait for communication is never interrupted.[*] There is no global clock in DOT, and processing cells operate asynchronously with respect to each other.

Approaching the design in this way, using relatively simple sequential processes that operate fairly independently, is appropriate in a VLSI context. Also realistic are the use of static processes and the separation of control activities located in different cells from each other (so they must communicate via messages). Within each cell of the DOT model, the multiple processes resident therein are allowed to communicate in whatever way seems most natural -- using shared memory or condition monitors, as appropriate.

3.1.5. Communication Between Processes in Different Cells

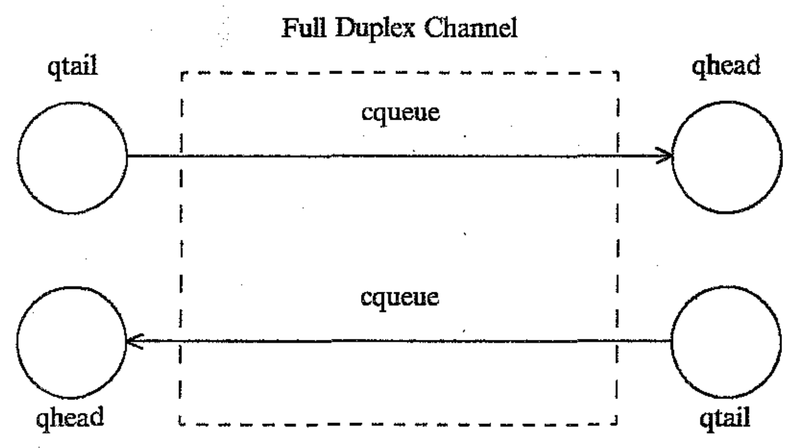
Generally, multiple processes within a given processing cell of DOT will be communicating with their counterparts in other cells at the same time. To enable this, the DOT communication buses are composed of logical channels that are full-duplex in nature -- that is, the end of each channel has separate send and receive ports, both of which may be in use (by separate processes) concurrently.[**] An actual realization might wish to multiplex these channels, but DOT simply assumes the existence of the required logical channels. Certain channels -- those associated with support for the message subsystem of the LPL architecture -- are circuit-switched in the course of the machine's operation to provide dedicated support for FFP RAs. As shown in Figure 3.4, DOT channels are built from two single-direction message pipes called *cqueues*. Each cqueue has a *qtail* (for sending) and a *qhead* (for receiving) as shown in the diagram. When a process wants to send a message through a cqueue, it deposits the data (a single

[*] If one process waits for the arrival of a message from another process, a message is guaranteed to arrive; processes are never interrupted from attempting to effect a message transfer.

[**] Sending and receiving ports are not restricted by the model to be located in separate cells; this is simply the more general case.

byte) into the associated qtail. When a process want to receive a message from a cqueue, it picks up the data from the associated qhead.

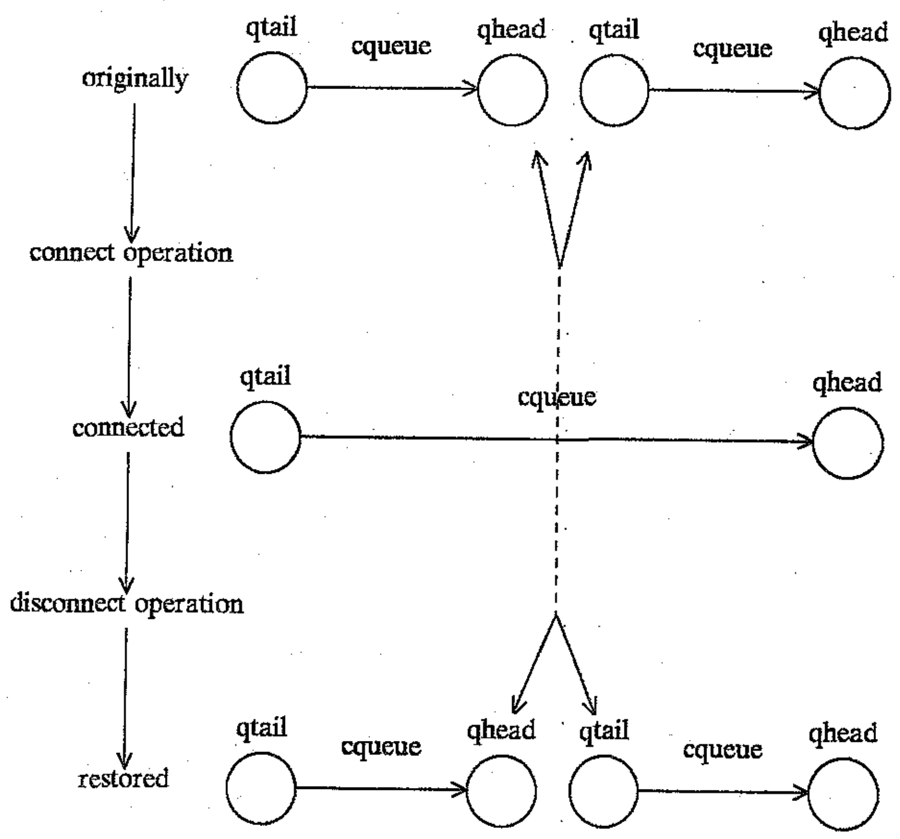
FIGURE 3.4 – A DOT Full Duplex Communication Channel



A cqueue has a variety of interesting properties. Most importantly, a cqueue implements a "safe" message transmission mechanism. Both sender and receiver are synchronized by a message transfer; a sender is delayed until a receiver requests data, and vice-versa. This corresponds to the synchronization that is necessary between hardware processes that operate from different clocks. DOT thus encapsulates concern for the synchronization that must be present in the realization at this level.

Cqueues may be connected and disconnected in a manner that corresponds to circuit-switching. Figure 3.5 depicts such a procedure. A succession of cqueues may be connected to form a "long distance" connection through the tree, and then be disconnected in any order. This allows DOT to perform circuit-switching of communication channels.

FIGURE 3.5 – Connecting and Disconnecting Cqueues



The ClassC entry templates for the cqueue qhead and qtail objects (which summarize the operations of which a cqueue is capable) are shown in Figure 3.6. The **connectw** operation connects two cqueues as discussed above, and delays the connecting process until an **eot_alert** (for end-of-transmission) is performed by a sending process on the cqueue qtail.

FIGURE 3.6 – The DOT Cqueue Head and Tail – public entries

```
class qhead
{
          public:
          /* allowed operations on the qhead of a cqueue */
          char      get(*char);
          void      connect(class qtail *);
          void      connectw(class qtail *);
          void      disconnect(class qtail *);
};
class qtail
{
          public:
          /* allowed operations on the qtail of a cqueue */
          void      put(char);
          void      connect(class qhead *);
          void      connectw(class qhead *);
          void      disconnect(class qhead *);
          void      eot_alert();
};
```

3.2. Overall DOT Operation

Having introduced the top-level constituents of DOT, including the mechanism used for communication between cells, we can now establish the relationship of processes within DOT to the activities that must be performed in support of FFP and LPL. As a first step, we present an overview of the combined effects of the cooperative behavior of these processes. This will introduce important terminology, informally mention the different process types and describe their essential functions. Once this has been done, the internal process structuring of the lcell and tcell classes will be given, and important algorithms used by the processes will be discussed.

3.2.1. The Basic Machine Cycle

A DOT machine cycle starts with looking at the lcell array to see what is in it. During this phase of the machine's operation, RAs are discovered, and the machine is *partitioned* to correctly allocate circuit-switched communication channels and tcell processing power to the discovered RAs. This *partitioning phase* involves all operations necessary to prepare for LPL execution within RAs. The first time a particular RA is encountered (RAs may exist over a period of many machine cycles), DOT processes within the tcells and lcells build the LPL environment directories, and LPL code segments are loaded using the io subsystem. Partitioning completes separately for each RA, so the duration of this phase is shorter for RAs that are restarted (their containing lcells already have environment directories and code segments). Immediately following completion of the partitioning phase within each RA, execution of LPL code segments begins.

At this point, the notion of a single machine is misleading; each RA has its own dedicated multiprocessor hardware and is completely independent of the others. Nevertheless, after the RAs are started (or restarted), the overall machine may be thought of as being in an *execution phase.* The LPL programs run, with the aid of DOT-provided services, until they become blocked awaiting additional lcells to hold expanding FFP text, or are preempted by DOT for the purpose of storage management.

The *storage management phase* includes stepping forward RAs whose code segments have completed, determining the new storage requirements of the FFP programs within the lcell array (due to LPL fork statements that have been executed), and shifting LPL program segments and their contexts within the lcell array to make room for newly required symbols. The shifting process is performed using the lateral lcell connections and may result in:

- overflow of contexts into the virtual memory subsystem if enough lcells are not available;
- reentry of previously overflowed lcell contexts back into the lcell array if there is room;
- or entry of new FFP programs if there is room after previous overflow has been taken care of.

The prescription for exactly how the lcell contents are to be shifted about is called the *specification for storage management.* Calculation of this information is called *preparation for storage management.*

The basic machine cycle is thus partitioning, execution, and storage management. Each phase will now be described in more detail.

3.2.2. Partitioning Phase

Partitioning creates *active areas* , each of which is composed of the communication channels and the lcell and tcell hardware required to support computation in an individual RA. An active area is essentially a small dedicated multiprocessor, which is structured as a binary tree and dynamically embedded within the overall tree-structured multiprocessor.

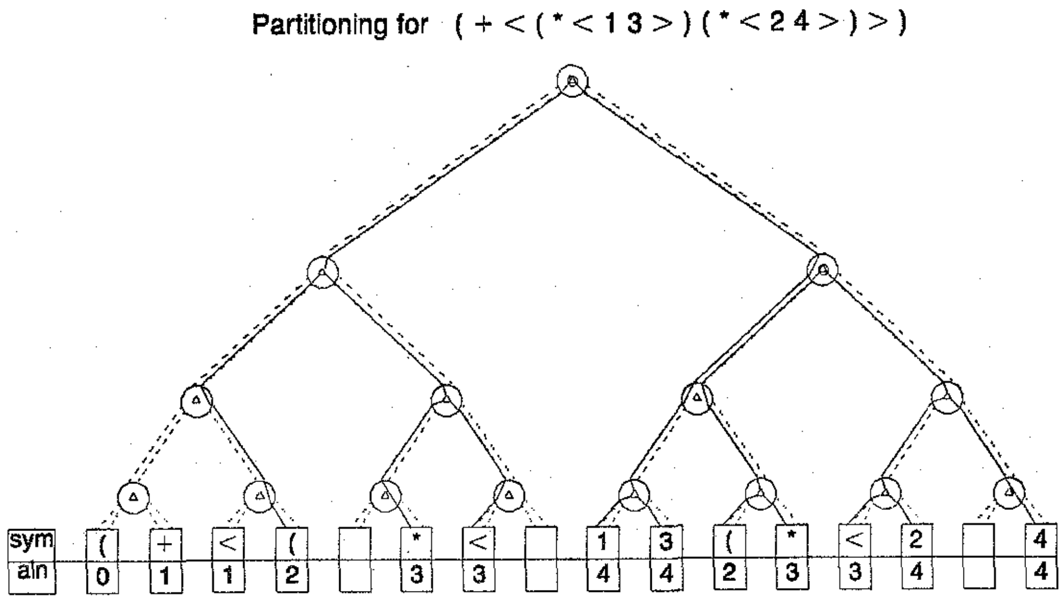
Partitioning begins in the lcells, with information being sent upwards into the tcells. Each tcell receives (from its two children) and sends (to its parent) a code containing the information necessary for an *initial partitioning* of the tcells. The initial partitioning (a pipelined upsweep of information starting at the lcells, and terminating in the io subsystem) allocates and connects dedicated area communication channels (called *area channels*) and dedicated tcell processing power (called *area nodes*) to each underlying group of lcells that may contain a different RA. While area channel connections are modified (to simulate circuit-switching) with each partitioning, the information required for the initial partitioning travels upwards on *cell manager channels* whose connections are never modified.

The initial partitioning is terminated within the io subsystem, which may be thought of as the parent of the root of the tree.* RAs are finally located and their corresponding active areas created with the aid of concurrent downsweeps of information within each of the candidate areas created by the initial partitioning. This downsweep is called the *pruning downsweep* of partitioning. During this downsweep, information sent downwards on area channels connected in the partitioning upsweep is used to disconnect any channels that lead to lcells not contained in an RA. In each active area that remains, the lowest tcell area node above all lcells of the RA (the least common ancestor) is located and configured as the *top of area* (or *toa*) where rising LPL messages turn around for broadcast back down to the lcells.

Figure 3.7 shows the area channels and nodes for a partitioned DOT machine. The circles in this figure represent tcells, and interior triangles represent the area nodes. Solid lines between area nodes represent connected full-duplex area channels, and the dotted lines represent unused area channels. There are two active areas in the figure, each supporting an RA whose operator is multiplication. The two top of area nodes are shown by circling the appropriate triangular node representations. Note that empty lcells, interspersed among the FFP text, are not included in the active areas.

* In addition to its io-related activities, the io subsystem offloads special termination processing from the tree root.

FIGURE 3.7 – A Partitioned DOT Machine



3.2.2.1. Area Nodes

As suggested by Figure 3.7, a tcell need only provide processing power for one active area. Even though area channels for more than one active area may pass through a given tcell, it is always possible to route (via circuit-switching) all but one set of area channels directly through the tcell. Only channels that lead to two children in the same RA are connected to an area node, whose purpose is to support all subsequent area-related processing within the tcell.

This support begins with a pruning downsweep to help complete the partitioning phase. Pruning is performed entirely within down-going area channels, and encompasses the following activities:

- discovery of whether underlying FFP text is truly part of an RA (i.e., whether the connected area channels and nodes are to be active during the coming execution phase) -- if not, area channels connected on the upsweep are disconnected;
- disconnecting channels that are within an active area but are not required for area processing because they lead to empty lcells;
- discovery of the FFP operator if the area is active;
- creation of the top of area node where LPL messages turn around.

In an active area, partitioning is then followed by support for the LPL message subsystem **send** and **receive** statements until the execution phase comes to an end. This is followed by correctly shutting down area operation prior to the storage management phase of the machine cycle. This shutdown must disconnect area channels (that were created during partitioning), but only after stopping messages in such a way as to guarantee that all lcells in the area will have seen exactly the same messages during the execution phase. This must be done in order to guarantee a consistent restart following storage management and re-partitioning.

3.2.2.2. Directory Creation

During the pruning downsweep, each top of area node returns to its descendent lcells notification of their active status and the LPL program to be used if one is necessary. Given this information, an lcell will decide to create a directory if it is contained in a new RA. This requires an upsweep and a downsweep of information within area channels, and the result is to load *symbol_index*, and the 4-tuple directory, *d1* ... *d4*, in the RA lcells with the correct values. If the RA is not new, the old directory is still valid and execution may begin immediately without this step.

3.2.2.3. Loading LPL Programs

The LPL programs are delivered from the io subsystem on *io channels* that follow the hardware tree structure. Within a tcell, each parent io channel splits into two child io channels and data movement is as follows: input to the lcell array comes from "above" and is broadcast to all lcells by successively splitting data so what comes in from a parent input channel is sent down both child input channels; output comes from "below", and is sequenced by handling the child output channels in cyclic left-to-right order. There are two very simple processes in the tcell that perform these functions. At present, the input channels are used to deliver LPL programs from the library, and output channels are used to return execution results and trace information to the outside world.

3.2.3. Execution Phase

The lcell LPL interpreter is a process that receives starting addresses from a queue. It begins execution at a requested address, performs local data movement and manipulation as indicated by the loaded LPL object code, and continues until encountering one of the following DOT service requests that require special handling: **send, receive, endfilter, fork,** and **endsegment**.

These special services are initiated by setting up an LPL context area associated with the particular service required. These areas are checked by the DOT processes whose job it is to provide the services. Having set up the service request area, the interpreter then cycles back for another start address. The reason for this approach will be seen in the following discussion of message support.

3.2.3.1. Lcell Message Support

When a message resulting from a **send** or **receive** for the present message wave arrives, it should be filtered. A DOT lcell message input process first puts the newly received message into a receive area (accessible to filter statements using environment variables such as *r_arg1)*, and then uses information deposited earlier (by the interpreter) in the LPL program context to insert the beginning address of the message filter statements into the interpreter start address queue. The interpreter executes the message filter for the message instance, and then encounters the **endfilter** statement, which then halts the interpreter as described above in Section 3.2.3. This is done for each message that arrives on the present message wave. When the wave has completed, the lcell message input process places the continue address (i.e., the address of the first statement following the **endfilter** statement) into the interpreter start address queue, and LPL execution then continues.

Message waves are sequenced activities whose completion requires agreement among all of the lcells of an RA. The basis of this agreement is an *end-of-wave* or *eow* that is sent for each message wave by all lcells of an RA, merged into a single message by the time it reaches the top of area, and then returned to the lcells in the RA. Lcells keep a counter that contains the present message wave number.

When the message wave counter is incremented, the LPL program context is checked for a **send** or **receive** request for the new wave number. If there is such a request, an *eow* is sent (after message transmission if the request was **send**). If the request is for a a higher numbered wave, *eow* is also sent. This indicates that no sending is desired for the current message wave by the LPL segment executing locally. If, however, the last message request handled by the lcell is for a lower numbered message wave (and the segment has not completed), a fork has been executed. In this case, *eow* is not sent. Instead, the lcell waits for storage management to complete the fork operation. The result is that the new message wave cannot pass through the top of area node until after storage management (and completion of the fork operation).

Following storage management and re-partitioning, an interrupted message wave is continued by re-sending any messages that were sent up but not received during the preceding execution phase. Everything is restarted correctly so that the message wave interrupted by storage management can complete and the next one can begin (all transparent to the LPL program). As explained in Section 2.3.3.1, this allows implicit synchronization of a fork operation with a corresponding send designed to copy information.

3.2.3.2. Fork Support

A **fork** statement halts execution within the requesting lcell until the operation can complete during storage management (when LPL program contexts are shifted in the lcell array). Execution then resumes in child LPL contexts (i.e. those created by a fork operation) as well as the parent. LPL program contexts begin each execution phase acting as if they had requested a forksize of one. The **fork** statement merely modifies the *forkn* LPL context register (not directly available by name to an LPL program) in which the forksize is stored -- so that multiple copies of an LPL program context are shifted during the next storage management.

3.2.4. Storage Management Phase

This phase is necessary to accommodate growth and compaction of the FFP text while retaining the necessary ordering of FFP symbols. It is unfortunate that execution of LPL programs should in general require interruption in order to implement this phase of the machine cycle. One alternative is to let all LPL programs complete (or become blocked as in a fork operation) before storage management is performed, but this could put RAs with quickly executing LPL programs at a disadvantage, and would likely result in inferior utilization of the available processing power in a large machine.

Attempts have been made to do storage management in locally restricted segments of the lcell array (as computation proceeds elsewhere) by Tolle [Tol81], but the complexity of the overall solution is considerable, and the resulting performance is not always superior to the preemptive approach that we use.

In the present design, lcells send permission to start storage management upwards on the cell manager channels to the io subsystem. Lcells that are not active do this following partitioning. Active lcells wait for the LPL program to complete, fork, or execute an **smanage** statement before sending permission. The resulting *sm_grant messages* are merged on their way up the tree, and, upon reaching the io subsystem, they result in a *stop message* which then travels down the tree and shuts down message activity.*

This approach places control of the processing cycle explicitly within LPL, and allows a system manager to tailor FFP operators for large operands if this is desired. Another possibility would be to allow the io subsystem to use heuristics based on lcell contents (discovered during partitioning) to determine an

* Due to the variety of messages that are sent between multiprocessor cells, it is useful to give them names corresponding to their purpose. In the case of the stop message, special emphasis may be appropriate since a "stop packet" will be referred to later in the context of LPL messages. The stop message, as explained above, originates in the io subsystem and travels down to the lcells on cell manager channels. The stop packet, to be discussed in Section 1.4.2.1, travels on area channels.

appropriate cycle time.*

3.2.4.1. The Specification for Storage Management

Once the LPL programs are shut down, a specification for storage management must be computed. This is done by sending and merging forksize information up the cell manager channels until it arrives at the io subsystem, where, as in partitioning, the upsweep is terminated. There, a specification for storage management is computed, and sent back down the tree in such a way as to distribute the necessary information to each lcell. A variation on the scheme suggested by Mago [Mag79] is used, so that total compaction will be performed only when necessary.

3.2.4.2. Overflow and Program Entry

The virtual memory concept used in the model is based on the work of Siddall [Sid83] and Frank [Fra79], who have examined various ways to accommodate overflow from the lcell array. The approach used in DOT is to allow movement of lcell contents into and out of the left lcell tree boundary. To the left of this boundary is a deque structure (interfaced with a file system), which receives from its right any lcell contents that overflow from the tree, and from its left new programs for execution in the tree. The state of the overflow and program entry subsystem (e.g., whether there is presently overflow in virtual memory, if so how much, how large the next FFP program to be entered is, etc.) is used by the io subsystem in its determination of the actual storage management specification.

* The simplest of such heuristics, a fixed cycle time, was originally used. For initial performance studies, however, it was desired that the machine execute FFP text as rapidly as possible, so the LPL architecture was modified to allow LPL programs to help schedule storage management.

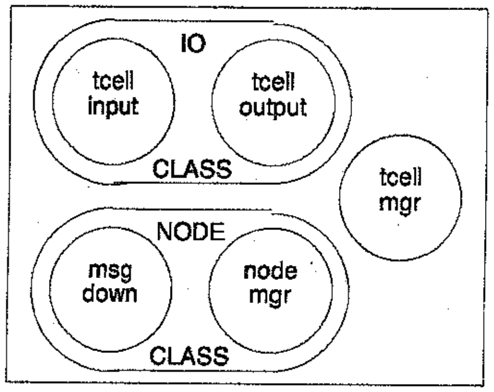
3.3. Process Structuring of DOT Cells

We have described the overall operation of the DOT implementation as the combined result of processes within the cells of DOT. These cells and their resident processes will now be examined. Sections 3.3.1 - 3.3.4 detail the tcell, lcell, io subsystem, and vm subsystem classes, respectively.*

3.3.1. Tcell Structure

Tcells of the DOT model contain five different processes, as shown in Figure 3.8.

FIGURE 3.8 – DOT Tcell Processes



The rationale behind this choice of processes is provided by the process-oriented design methodology. Each tcell process performs duties that are best viewed separately from the others, and for which there is a simple sequential (cyclic) description.

* For readers uninterested in the top-level structure of the contained processes, Figures 3.8 and 3.14 show the basic composition of the tcells and lcells. After glancing at these figures, the reader may skip to Section 3.4, which presents the detailed algorithms used by these processes, or turn to Chapter 4, on simulation.

The tcell_input and tcell_output processes bring input (presently in the form of compiled LPL programs) down to the lcells, and send output (trace information and completed programs) back up through the tree-structure to the outside world. Both of these processes run forever, awaiting the arrival of data and then re-sending it with a broadcast protocol (in the case of input), or a sequencing protocol (in the case of output).

Output originates within the lcells. With the completion of the execution phase, before the specification for storage management is computed, RAs that have completed are stepped forward. At this time, each lcell does optional output followed by an **eot_alert** (to signal end-of-transmission) on its output channel.* This output is relayed up and out of the tree by the tcell_output processes, which loop forever (alternately accepting output from left and right children). Figure 3.9 shows the ClassC representation of the tcell_output

FIGURE 3.9 – Tcell Output Process

```
/*
Tcell output is done by alternately switching
output to parent from left and right child channels
*/

tcell_output.new(top,p,l,r)
short top;              /* true if at top of machine */
class qtail *p;                 /* connection to parent */
class qhead *l,*r;              /* connections to children */
{
        cycle   {
                l->connectw(p); l->disconnect(p); /* relay left */
                r->connectw(p); r->disconnect(p); /* relay right */
                if (!top) p->eot_alert();        /* signal parent cell */
                else p->put(0);                  /* or signal io subsystem */
                }
}
```

* At present, output includes a snapshot of important lcell registers in all non-empty lcells.

process.

Input originates from the io subsystem in response to requests for LPL programs during partitioning. A tcell_input process supports delivery of all arriving information by relaying it to both child subtrees. The combined effect is to broadcast all LPL programs to all lcells. The tcell input process is shown in Figure 3.10.

The message-down process component of an area node (called *node_dmh* within DOT -- for downwards message handler) operates in a similar fashion to the tcell_input process, and is used to support the LPL message broadcast protocol. However, since only active areas support messages, and since area channel connections are changed with partitioning, the message-down process must be started and stopped. The node manager process (to be described below) uses a condition monitor to signal its message-down process that area channels are connected and messages should be relayed downwards. Arrival of a special stop packet for broadcast to the lcells tells the message-down process to stop handling messages for the current cycle. The down-message process is shown in

FIGURE 3.10 -- The Tcell Input Process

```
/*
Input of lcell programs to the tree is done via broadcast to
both children.
*/

tcell_input.new(p,l,r)
class qhead *p;
class qtail *l,*r;
{
        cycle   {
                p->get(&c);
                l->put(c); r->put(c);
                }
}
```

Figure 3.11.

FIGURE 3.11 -- Area Node Downwards Message Handler Process

```
/*
Downwards Message Handler for area node within a tcell. A cnt of
STOPCNT indicates that no more messages should be broadcast.
*/

node_dmh.new(messages_started,phead,ltail,rtail)
class condition *messages_started;        /* condition signaled by node mgr */
class qhead *phead;               /* connection to parent */
class qtail *ltail, *rtail;               /* connections to children */
{
char c, cnt;

cycle       {
            /* wait for messages to be started on the area channels */
            messages_started->await(TRUE);

            /* relay messages downward until the stop packet arrives */
            while ((cnt=phead->msg()) != STOPCNT) {
                        ltail->put(cnt); rtail->put(cnt);
                        while (cnt--) {
                                    c = phead->msg();
                                    ltail->put(c); rtail->put(c);
                                    }
                        }

            /* output stopcnt, reset condition, and cycle back */
            ltail->put(STOPCNT); rtail->put(STOPCNT);
            messages_started->assert(FALSE);
            }
}
```

The remaining two processes are the work-horses of a tcell. These are the tcell manager and the node manager processes. The tcell manager is responsible for correctly overseeing and implementing the overall machine cycle (partitioning, execution, and storage management). The node manager is responsible for all processing that takes place on area channels. The top-level of the tcell manager is shown in Figure 3.12, and the top-level for the node manager is given in Figure 3.13.

FIGURE 3.12 -- Tcell Manager Process

```
/*
The tcell manager does initial partitioning, and other activities not
directly associated with area execution, such as relaying LPL program
requests upward to the io subsystem. The parameters of a tcell manager
include whether it is at the top of the tree, and the communication
channels (qheads and qtails) which enter and leave its tcell.
 */

tcell_mgr.new(at_top_of_tree,
                    cm_dn_head, a1_dn_head, a2_dn_head,
                    cm_up_tail, a1_up_tail, a2_up_tail,
                    lcm_up_head, la1_up_head, la2_up_head,
                    lcm_dn_tail, la1_dn_tail, la2_dn_tail,
                    rcm_up_head, ra1_up_head, ra2_up_head,
                    rcm_dn_tail, ra1_dn_tail, ra2_dn_tail,
                    npheads, nptails)
{
          /* initialize class objects */
          lspf=rspf=0;
          node_pgm_ready = new class condition(FALSE);
          node_task_ready = new class condition(FALSE);
          tcell_eom = new class condition(FALSE);

          /*** MAIN EXECUTION LOOP ***/
          cycle  {
                    node_eom = FALSE;
                    tcell_eom->assert(FALSE);
                    node_pgm_ready->assert(FALSE);
                    if (lspf || rspf) disconnect_partitioning();
                    initial_partitioning_up();
                    relay_area_pgms_requests_up();
                    await_stop_signal_dn();
                    compute_sm_specification_up_dn();
                    }
}
```

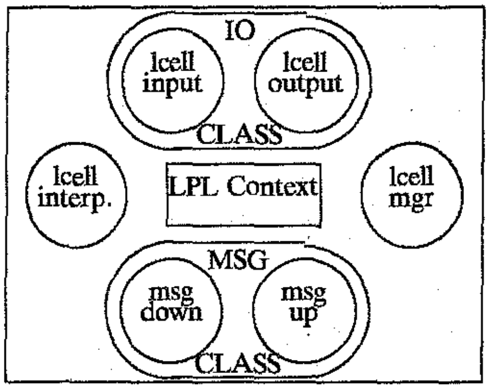
FIGURE 3.13 – Tcell Node-Manager Process

```
/*
This class handles all area-related processing in a tcell when two
children of the tcell are involved in the same area. The parameters
of a node manager include whether the tcell is at the top of the tree,
access to memory shared with the tcell manager, and the area channels
(qheads and qtails) to which it has been temporarily connected by the
tcell manager. The node manager uses an upwards-message-handler
class, (node_umh) to handle the details of message processing.
The downwards message handler process for the node is also created here.
*/

node_mgr.new(top_of_tree, cel_mgr,
                    np_up_head, np_dn_tail,
                    np_dn_head, np_up_tail,
                    lnp_up_head, lnp_dn_tail,
                    rnp_up_head, rnp_dn_tail)
{
         /* init the upward message handler class */
         upm = new class node_umh(cell_mgr,np_tail,lnp_head,rnp_head);
         /* startup the downward message handler task */
         messages = new class condition(FALSE);
         down_messages = new class node_dmh
                             (messages,np_head,lnp_tail,rnp_tail);

         /*** MAIN EXECUTION LOOP ***/
         cycle {
                  /* wait for area assignment */
                  (cell_mgr->node_task_ready)->await(TRUE);
                  (cell_mgr->node_task_ready)->assert(FALSE); /* reset */
                  top_of_area = finished = FALSE; /* initial guess */

                  finish_partitioning();
                  if (!finished && (state==GROUND)) build_directory();
                  if (!finished) {
                           messages->assert(TRUE); /* dmh has work */
                           upm->up_messages(top_of_area);
                           messages->await(FALSE); /* dmh is finished */
                           }
                  else (cell_mgr->tcell_eom)->await(TRUE);
                  disconnect_partitioning();
                  } /* end of cycle */
}
```

3.3.2. Lcell Structure

The DOT lcell contains six processes. These include two io processes -- one for LPL code input, and one for program output -- that are connected via io channels to the tcell io processes. There are two processes associated with handling messages, and an lcell manager process. The last process is the LPL interpreter which executes compiled LPL code. Figure 3.14 shows the overall lcell structure, including the (passive) LPL user context area in which the LPL code and environment is located.

FIGURE 3.14 – DOT Lcell Processes (and User Context)



The lcell_input process is connected to the end of an io channel through which all LPL programs required by any RA in the machine arrive, and it filters this flow of information to select and load only the LPL code segment that is locally required (if any). Everything else is thrown away. As soon as the correct code segment is loaded, the input process starts the LPL interpreter by sending it a beginning code address. Figure 3.15 shows the lcell input process.

Once the LPL program begins execution, it may request message services. The lcell message-up process (called *lcell_msend* within DOT) sends an LPL

FIGURE 3.15 -- Lcell Input Process

```
/*
The lcell input process accepts and loads LPL programs. When a segment
is loaded, the interpreter is started up. The input process must wait
until the LPL directory has been created (during partitioning) in order
to filter input and select the correct code segment.
*/

lcell_inp.new(input_head, area_tail, user, input_possible, interpreter)
class qhead *input_head; class qtail *area_tail; class lcell_usr *user;
class condition *input_possible; class stail *interpreter;
{
char pgm, garbage, found, loaded;
short i,cnt;
  cycle {
    input_possible->await(TRUE); /* wait for directory ready */
    /* filter lcell programs */
    loaded = user->state!=GROUND;  /* i.e., executing, or completed */
    while (pgm = in_head->msg()) {
      if(user->lcell_pgm==pgm && user->active)
        found = TRUE;
      else found = FALSE;
      for (i=0; i<DLEVELS; i++)
        found &= match(user->directory[i],in_head->msg());
      cnt = in_head->msg();
      cnt = 256*cnt + in_head->msg();
      if (found && !loaded) {
        if (cnt>CODESIZE) fprintf(stderr,
          "lcell_inp(%d):!!! segment too large",this);
        else {
          for(i=0; i<cnt; i++) user->code[i] = in_head->msg();
          loaded = TRUE; user->state = EXECUTING;
          user->nsymbol_cnt=0;
          user->mwave=user->mfilt=user->msend=0;
          user->mcomplete=user->endsend=FALSE;
          interpreter->put(0);    /* start program */
          input_possible->assert(FALSE); /* reset and signal msgs*/
          }
        }
      else while (cnt--) in_head->get(&garbage);
      }
  input_possible->assert(FALSE);
  if (!loaded && user->active && (user->state==GROUND))
    fprintf(stderr, "lcell_inp(%d)!!! no cell_pgm %d");
  } /* end cycle */
}
```

message up into the tree when notified to do so by the message-down process. It either sends an eow (end-of-wave) for the current message wave, or sends a message followed by eow, depending on the contents of the lcell user context. The message-up process is shown in Figure 3.16.

FIGURE 3.16 -- Lcell Message-up Process

```
/*
This process takes care of the details involved in actually sending
messages, and is the one who waits when message bytes are handed off
to the parent tcell. The upward-going message channel used by this
process is of class ptail. A ptail is like a qtail, but uses a
locking mechanism to prevent interference during transmission of
a sequence of message bytes. The lock is set by a ptail.put_first, and
released by ptail.put_last. This lock is necessary because the lcell
manager signals eom (end of messages) asynchronously by sending a stop
packet up through area channels at the end of an execution phase.
*/

lcell_msend.new(user,msg_tail,msg_ready)
class lcell_usr *user;
class ptail *msg_tail;
class condition *msg_ready;
{
short i;

        cycle {
                msg_ready->await(TRUE);
                msg_ready->assert(FALSE); /* reset */
                if ((user->mwave == user->msend) &&
                        (!user->mcomplete) &&
                        (!user->endsend) &&
                        (user->fork_id) &&
                        (user->state==EXECUTING)) {
                          msg_tail->put_first(2);
                          msg_tail->put(user->mspec[MORD]);
                          msg_tail->put_last(user->mspec[MCOP]);
                          msg_tail->put_first(user->mspec[MARGC]+2);
                          msg_tail->put(user->mspec[MKEY1]);
                          msg_tail->put(user->mspec[MKEY2]);
                          for (i=0; i<user->mspec[MARGC]; i++)
                                  msg_tail->put(user->margs[i]);
                          }
                msg_tail->put_last(0);
                } /* cycle */
}
```

The message-down process (called *lcell_msg* within DOT) is the main message handler in the lcell. It must delay an LPL program that requests message handling until the service is complete, interact with the interpreter to start up a message filter when appropriate, and then continue execution following completion of the message operation. It is this process that sees the eow that signals the end of one message wave and the beginning of the next, and it must tell the message-up process when to send messages. Figure 3.17 shows this process.

The lcell interpreter is organized as usual -- with an interpretive loop. It receives start addresses from a queue, and executes local operations until it encounters a request for a special service (such as **send**) which is supported by other DOT processes. All special service requests are distinguished by not loading the code pointer with the next instruction to execute. In these cases, the next instruction to execute will be indicated by an arrival on the start address queue. The subroutine *execute_seg* is used to execute a code segment until a special service request is encountered. Figure 3.18 shows the top level of the lcell interpreter process. Details concerning support for the special services are hidden at this level (they involve setting register values in the user-context).

In a way similar to the tcell manager, the lcell manager supports the basic machine cycle and coordinates the behavior of the other processes in the lcell. After storage management, an executing LPL context may have to be restarted, and the lcell manager takes care of this. The top level for the lcell manager is shown in Figure 3.19.

FIGURE 3.17 – Lcell Message-Down Process

```
lcell_msg.new(msg_ready,directory_ready,msg_head,user,
              interp_start,interp_idle,shutdown)
class condition *msg_ready,*directory_ready,*interp_idle,*shutdown;
class qhead *msg_head; class lcell_usr *user; class stail *interp_start;
{ char finished,garbage,cnt,*msg; short i;
  cycle { /*** MAIN EXECUTION LOOP ***/
    do { /* first, wait for a valid active directory */
      directory_ready->await(TRUE);
      if (!user->active) directory_ready->await(FALSE);
      } while(!user->active);
    directory_ready->await(FALSE); /* signals LPL seg loaded */
    while ((cnt=msg_head->msg()) != STOPCNT) { /* service msgs */
      if (cnt) { /* get msg, check origin, filter if necessary */
        msg = user->mtmp; while (cnt--) *msg++ = msg_head->msg();
        if ((user->mtmp[RKEY1] == user->mspec[MKEY1]) &&
          (user->mtmp[RKEY2] == user->mspec[MKEY2]) &&
          (user->mwave == user->msend)&&(user->state==EXECUTING))
            user->mcomplete = TRUE;
        if ((user->mwave > user->mfilt)&&(user->state==EXECUTING))
           interp_idle->await(TRUE);
        if ((user->mwave == user->mfilt)&&(user->state==EXECUTING)){
          interp_idle->assert(FALSE);
          interp_start->put(user->filt_addr);
          interp_idle->await(TRUE); }
        } /* end handling msg */
      else /* end-of-wave, so start next wave */ {
        user->mwave++; interp_idle->await(TRUE);
        if (!user->fork_id || (user->state!=EXECUTING))
          msg_ready->assert(TRUE); /* need to send eow */
        else { /* handle executing user */
          if ((user->mfilt) && (user->mwave > user->mfilt)) {
            /* continue user following msg services */
            interp_idle->assert(FALSE);
            interp_start->put(user->cont_addr);
            interp_idle->await(TRUE); }
          if (user->endsend || (user->mwave<=user->mfilt))
            msg_ready->assert(TRUE);
          } /* end handling executing user */
        } /* end handling start of new wave */
      } /* end while message activity */
    /* cnt == STOPCNT, so time to do shutdown */
    user->shutdown(); shutdown->assert(TRUE);
    } /* end main cycle */
}
```

FIGURE 3.18 – Lcell LPL Interpreter Process

```
lcell_int.new(start,user,idle,smgrant)
class shead *start;
class lcell_usr *user;
class condition *idle, *smgrant;
{
char *code = user->code;

  cycle { /*** MAIN EXECUTION LOOP ***/
           idle->assert(TRUE);
           addr = start->msg();
           if (user->active && (user->state==EXECUTING)) {
                    addr = execute_seg(addr,user);
                    switch (*(code+addr)) { /* handle special request */
                    case SEND:
                             user->filt_addr = user->setup_send(addr);
                             break;
                    case RECV:
                             user->filt_addr = user->setup_recv(addr);
                             break;
                    case FORK:
                             user->cont_addr = user->setup_fork(addr);
                             sm_grant->assert(TRUE);
                             break;
                    case FORKC:
                             user->cont_addr = user->setup_forkc(addr);
                             sm_grant->assert(TRUE);
                             user->endsend = TRUE;
                             break;
                    case ENDFILT:
                             break;
                    case ENDPROG:
                             sm_grant->assert(TRUE);
                             user->state = COMPLETED;
                             user->endsend = TRUE;
                             break;
                    default:
                             printf("lcell_int(%d):%d is no request",
                                      this,*(code+addr));
                    } /* end switch */
           } /* end cycle */
}
```

FIGURE 3.19 -- The Lcell Manager Process

```
/*
This class is the lcell manager.  It performs partitioning, directory
creation, restarting execution after storage management, preparation for
storage management, and storage management.
*/

lcell.new(io_dn_head,cm_dn_head,area_dn_head,
          io_up_tail,cm_up_tail,area_up_tail,
          lb_rs_head,lb_ls_tail,rb_ls_head,rb_rs_tail)
{
          /* init lcell objects */
          user = new class lcell_usr();
          interp_idle = new class condition(FALSE); /* (since last use) */
          input_possible = new class condition(FALSE);
          sm_grant = new class condition(FALSE);
          shutdown = new class condition(FALSE);
          interp_start=new class shead(); /* start address queue */
          input = new class lcell_inp(io_head, area_tail, user,
                              input_possible, interp_start->tail());
          interpreter = new class lcell_int(interp_start, user, interp_idle,
                                             sm_grant);
          msg_tail = new class ptail(area_tail);
          msg_ready = new class condition(FALSE);
          msg_service = new class lcell_msend(user,msg_tail,msg_ready);
          messages = new class lcell_msg(msg_ready,input_possible,
            area_head,user,interp_start->tail(),interp_idle,shutdown);

          /* MAIN EXECUTION LOOP */
          cycle {
                    do_partitioning();
                    if (user->active&&(user->app_state==GROUND))
                              build_directory();
                    input_possible->assert(TRUE);
                    sm_grant->assert(FALSE);
                    shutdown->assert(FALSE);
                    msg_tail->clear_priority();
                    if (user->active) restart_execution();
                    terminate_cycle();
                    prepare_for_storage_management();
                    do_storage_management();
            }
}
```

3.3.3. IO Subsystem

The io subsystem, located "above" the processing tree composed of tcells and lcells, comprises three processes. Among these is a main process that terminates the initial partitioning, interfaces with requests for LPL programs, and computes the specification for storage management. Additionally, there is a tree-input process that sends LPL programs down into the tree to the lcells, and a tree-output process that accepts output from the tree. The main io process is shown in Figure 3.20.

FIGURE 3.20 -- The Main IO Process

```
/*
This is the main io processes. Its duties are to terminate initial
partitioning, filter and hand off LPL program requests to the tree-input
process, and compute the global sm-specification.
*/

io.new(io_up_head,cm_up_head,a1_up_head,a2_up_head,
            io_dn_tail,cm_dn_tail,a1_dn_tail,a2_dn_tail,
            ov_up_head,ov_dn_tail)
{
          /* init class objects */
          in_req_head = new class qhead();
          in_req_tail = in_req_head->tail();
          output= new class io_output(io_up_head);
          input = new class io_input(in_req_head,io_dn_tail);

          cycle     {
                    terminate_partitioning_upsweep();
                    accept_operator_requests();
                    prepare_for_storage_management();
                    }
}
```

3.3.4. VM Subsystem

The virtual memory subsystem (also called the program overflow and entry subsystem) is implemented with a single process, as shown in Figure 3.21.

The vm subsystem serves two purposes. It accommodates overflow out of the

FIGURE 3.21 – The VM Overflow and Program Entry Process

```
overflow.new(lb_head,lb_tail,io_head,io_tail)
class qhead *lb_head, *io_head; class qtail *lb_tail, *io_tail;
{ /*** Main Execution Loop ***/
  cycle{  /* Tell IO subsystem what the situation is */
    io_tail->put(ovr_cells); io_tail->put(pgm_cells);
    /* Get amount to shift from IO */
    bcond = io_head->msg();
    if (bcond<0) /* then handle overflow */ {
      lseek(ovr_fd,ovr_next,0);
      while (bcond<0) {
        lb_head->get(&s); write(ovr_fd,&s,1);
        lb_head->get(&tmp); write(ovr_fd,&tmp,1);
        lb_head->get(&tmp); write(ovr_fd,&tmp,1);
        lb_head->get(&tmp); write(ovr_fd,&tmp,1);
        ovr_next += (i=4);
        if ((s==EXECUTING) || (s==COMPLETED))
          for ( ; i<SM_USERSIZE; i++) {
            lb_head->get(&tmp);
            write(ovr_fd,&tmp,1);
            ovr_next++; }
        write(ovr_fd,&i,1); /* num chars written */
        ovr_next++; bcond++; ovr_cells++; } }
    else if (bcond>0) /* then handle symbol entry into lray */ {
      /* first re-enter from overflow */
      while (bcond && ovr_cells) {
        lseek(ovr_fd,--ovr_next,0);
        read(ovr_fd,&i,1); /* get last context size */
        ovr_next -= i; /* start of last context */
        lseek(ovr_fd,ovr_next,0);
        for (ti=0; ti<i; ti++) {
          read(ovr_fd,&tmp,1);
          lb_tail->put(tmp); }
        bcond--; ovr_cells--; }
      /* then new program */
      while (bcond && pgm_cells) {
        pgm_cell_enter(pgm_fd,lb_tail);
        pgm_cells--; bcond--; }
     if ((pgm_cells==0) && (pgms_left)) {
        lseek(pgm_fd,next_ptr,0);
        pgms_left = read(pgm_fd,&pgm,1);
        if (pgms_left)  {
          read(pgm_fd,&pgm_cells,1);
          next_ptr=lseek(pgm_fd,2*pgm_cells-2,1)+2; }
        }
     } /* end overflow */
  } /* end cycle */
}
```

tree at its left boundary when there are more requested lcells than are available, and when there are more available cells than requested, previous overflow and then new programs are shifted to the right back into the tree. At present, for the purpose of simulation support, the vm subsystem is initially loaded with the FFP programs that are to be entered into the tree. The vm subsystem interacts with the io subsystem in order to determine whether overflow or program entry should be performed, and operates in the following way. It tells the io subsystem how many cells are in overflow, and how many are in the next program. The io subsystem then replies with the storage management boundary condition (i.e., how many cells to shift, and in which direction). Once the cells have been successfully shifted, the vm subsystem process cycles back to begin the above procedure once again.

3.4. Important Algorithms

In the last section, the overall process structuring of the DOT model was described. Partitioning, directory creation, message handling, and preparation for storage management deserve a more detailed treatment than given above because of their central importance to the working of the implementation, and also because their implementation is important to the analytic model given in Chapter 5. Sections 3.4.1 - 3.4.4 detail the algorithms used within the lcells and tcells to perform these functions.*

Of these operations, partitioning is the most complex, followed by message handling. The algorithm used for creation of the LPL directory corresponds in a direct manner to its definition given in Chapter 2, and the algorithm used for calculating the specification for storage management is also straightforward. In

* Readers uninterested in the fine structure of these algorithms may turn to Chapter 4, on simulation.

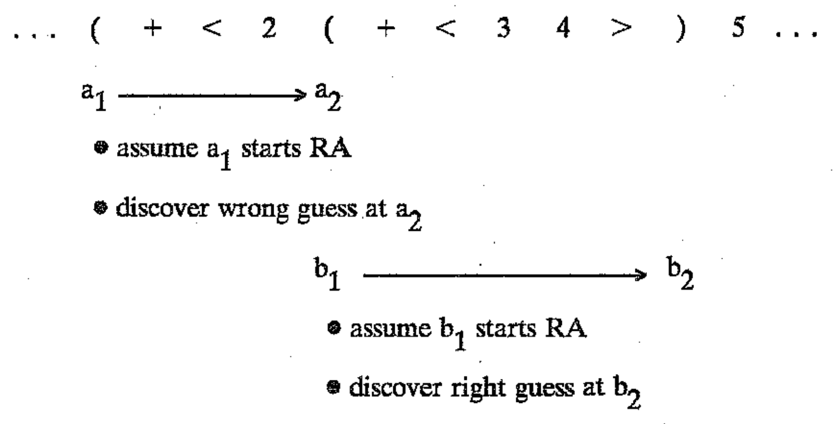
all of these operations, the main issue is efficient use of the tree-structure to perform global operations.

3.4.1. Partitioning

Partitioning is centered around the detection of RAs. It is the RAs, or lcells containing innermost applications, that are allowed to execute LPL within the active areas created during partitioning. Once RAs are found, their containing lcells must be told that they are active and that they should participate in the execution phase of the machine cycle. All other lcells must be told that they are not active. Thus, the partitioning process involves an upsweep of information through the tree to locate RAs, and a downsweep of information to notify lcells of their status. As this is done, area channels within the tree-structure are circuit-switched between area nodes (implemented by the node manager and message-down processes) to form individual tree-structured multiprocessors for support of subsequent execution within the detected RAs.

How are innermost applications found? One way of finding RAs, shown in Figure 3.22, might be to examine symbols in the lcell array from left to right, and assume that a new RA has been found every time a left application symbol is encountered. On reaching the next application symbol to its right, we discover whether or not our assumption was correct; if the next application symbol is a balancing symbol, then we have found an RA, otherwise not.

FIGURE 3.22 – Finding a RA



Since our FFP-level representation does not store balancing symbols (in the interest of conserving lcells), we must go on to the next symbol past where a balancing application symbol would be to decide whether we have discovered an RA, and the decision is then based on the aln value stored there as part of the FFP-level representation. The procedure in this case is the following:

After encountering an application symbol, $(_1$, we sequentially examine symbols to its right until we find:

(1) another application symbol $(_2$ at a deeper level (i.e., $aln_1 < aln_2$), or

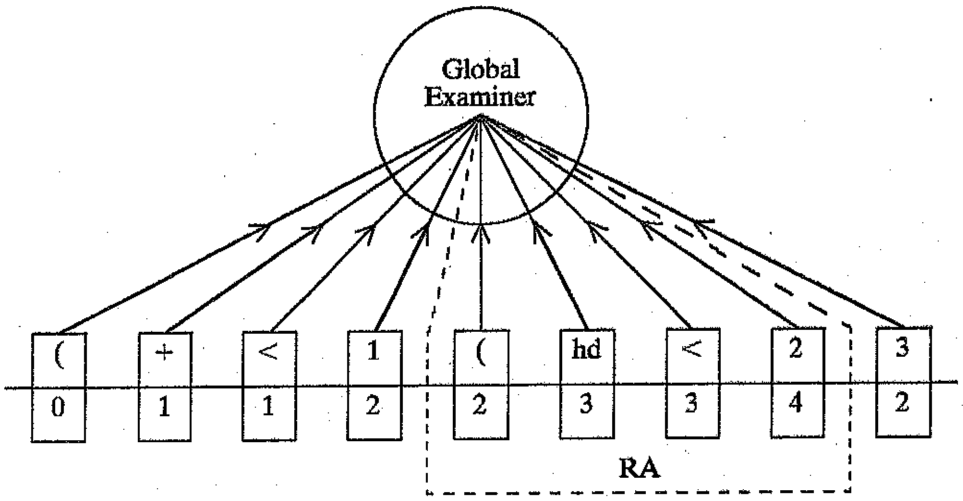
(2) a symbol at the same or higher level (or run out of symbols).

- *In the first case, the application beginning with $(_1$ is not an RA.*
- *In the second case, it is.*

3.4.1.1. Partitioning Upsweep – Locating RAs

In the above discussion, we implicitly assumed a single agency, or process, capable of examining the FFP-level text representation from left to right, one cell at a time. This is equivalent to letting the lcells send this information into a "global examiner" by sweeping the symbol and aln information up from left to right, as shown in Figure 3.23.

FIGURE 3.23 -- Sweeping Lcell Contents Upwards Globally



Of course, we need to accomplish the same thing using a binary tree-structure in place of the n-ary tree-structure of Figure 3.23. The n-ary tree method is easy to understand because one process with unlimited access to information is used within the single parent node. When using a binary tree structure, on the other hand, we must contend with a number of tcell processes, each of which has access only to the limited amount of information available from its two children.

Within the binary tree structure of DOT, in order to detect innermost applications and connect area channels to support them, information is swept up into the tree and each tcell accepts information from its two children descriptive of their respective underlying FFP text segments. Not all the information in a segment of lcells is needed by all of its tcell ancestors, however. This limits the amount of information that must be sent from any one level to the next.

3.4.1.1.1. Merging Segment Descriptors

During the initial partitioning upsweep, beginning with the lcells, each level of the tree-structure organizes information required by the next higher level into a *segment descriptor* and passes it upwards. When a tcell receives a segment descriptor from each of its children, this information is selectively merged into a new segment descriptor which is relayed upwards. When this information indicates an application (possibly an RA) entirely contained within the underlying lcell segment, the receiving tcell takes appropriate action locally, and elides information required solely by the discovered application (should it be innermost) from the segment descriptor which is sent upwards. Thus, area creation and denial is done as early as possible, and information not required at higher levels drops out of the upsweep.

One way to describe the partitioning upsweep is as follows: the segment descriptor for each application symbol in the lcell array moves upwards in the tree structure, accumulating its left and right symbol contexts (whose descriptors are sent up to support this process, and are merged with application symbol descriptors whenever possible) until it meets the segment descriptors belonging to the application symbols on its left and right. On the way up, until an application descriptor meets its right neighbor, area channels necessary to support the application (if it should turn out to be innermost) are connected. When the application descriptor finally sees its rightmost application neighbor, it has the information necessary to determine whether it is indeed part of an RA. If this is not the case, the channels just connected will be disconnected on the pruning downsweep. Although an application symbol meeting its right neighbor application symbol in this way has all the information it needs to make this decision, an appropriately modified descriptor must still be sent up further to enable an application symbol to its left to make a similar decision. When application symbols to the right and left of the given application have both encountered their "middle" application descriptor, this middle descriptor is no longer needed and drops out.

The segment descriptor has the basic format shown in Figure 3.24, and is composed of four logical fields. Within a segment descriptor (corresponding to a given segment of the lcell array) the leftmost symbol field, S_l, represents non-application symbols located to the left of the leftmost application symbol within the segment. The leftmost application symbol is represented by the $(_l$ application field. The rightmost field, S_r, represents the non-application symbols to the right of the rightmost application symbol within the segment, which in turn is represented by the $(_r$ application field. As shown in Figure 3.24, the symbol and application fields of a segment descriptor are composed of different subfields.

FIGURE 3.24 – The Segment Descriptor Format

$$S_l \quad (_l \quad (_r \quad S_r$$

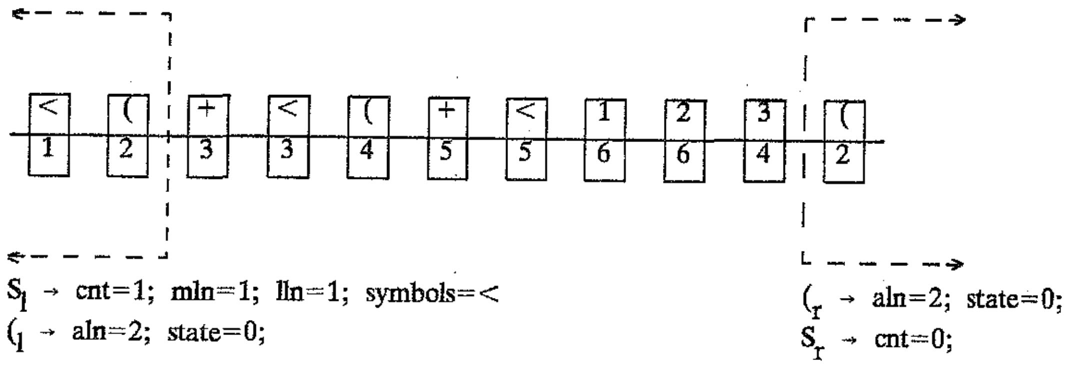
S	$\equiv$	*symbol_cnt : mln : lln : symbols*
$($	$\equiv$	*aln : state*

S_l and S_r don't actually contain all symbols of their appropriate subsegment. Only the two leftmost symbols of the represented subsegment are required in an S field (to guarantee finding the FFP operator for an RA), so *symbol_cnt* always has the value 0,1, or 2.[*] The *mln* value in an S field is the

[*] If an FFP operator is primitive, the first symbol to the right of the application symbol for its RA will be the LPL program op-code. If this symbol is a sequence symbol, however, the operator is a functional form. In this case, the second symbol to the right of the ap-

minimum aln within the entire subsegment described by S. The *lln* is the aln of the leftmost symbol of S's subsegment, and *symbols* are the leftmost symbols of the subsegment (if they exist). The *mln* and *lln* are used by the area nodes during the pruning downsweep portion of partitioning. If there are no symbols in a subsegment (empty lcells must participate in partitioning too), only a *symbol_cnt* of zero is sent, and the other *S* fields are not used. An application field always represents a single application symbol -- *aln* is its aln value, and *state* describes whether the application is in the *ground state* (the RA is new, so it will require an LPL program) or the *executing state* (the application is an RA that was executing last cycle and was interrupted for storage management, so it will not require an LPL program). Figure 3.25 shows an example (assuming ground state for applications) of a segment descriptor.

FIGURE 3.25 -- An Example of a Segment Descriptor



<
1
(
2
+
3
<
3
(
4
+
5
<
5
1
6
2
6
3
4
(
2
$S_l \rightarrow$ cnt=1; mln=1; lln=1; symbols=<
$(_l \rightarrow$ aln=2; state=0;
$(_r \rightarrow$ aln=2; state=0;
$S_r \rightarrow$ cnt=0;

The example in Figure 3.25 shows that all the information required by neighboring application symbols on either side of the example segment is

plication symbol contains either the op-code for the controlling operator of the functional form, or another sequence symbol. If it is a sequence symbol, the LPL program for meta-composition is requested. Otherwise the LPL program for the controlling operator is used.

represented in its segment descriptor, while information concerning the symbols between $(_l$ and $(_r$ is not included (it has dropped out earlier in the upsweep). The information in S_l and S_r is needed because of the absence of balancing application symbols, and because this information will contain the operator for an RA when it is detected. The tcell which detects an RA, therefore, can both request the appropriate LPL program to be brought in, and notify descendant lcells which LPL program to accept and use.

Description of a given segment of the lcell array may not warrant use of all of the fields of a segment descriptor. This is certainly the case at the first (lcell) level, where a segment descriptor will describe only a single symbol (application, or otherwise). For this reason, four different formats are used to express increasingly complex segment types. A segment descriptor is therefore preceded by a format code, or *SPF* (for segment pattern format) to indicate what format follows.* The four segment descriptor formats and their corresponding SPF codes are shown in Figure 3.26.

FIGURE 3.26 -- The Four Segment Descriptor Formats

FORMAT	SPF
S_r	1
$(_r\ S_r$	2
$(_l\ (_r\ S_r$	3
$S_l\ (_l\ (_r\ S_r$	4

* This was suggested by Peter Chen during early work on partitioning. The value of this approach is that the SPF code, alone, contains useful information that allows the partitioning upsweep to be pipelined.

3.4.1.1.2. Switching to Area Channels

While the segment descriptor represents the information that is used by the processes of a tcell in support of partitioning, this information is not sent up on a single communication channel, nor is it handled by a single process within each tcell.

One of the most difficult issues confronting the design of the partitioning algorithm was performing a smooth changeover from cell manager channels, which must start partitioning, to the area channels, which support area-related processing. The process-oriented design approach we took suggested that it would be a mistake to allow the tcell managers to complete partitioning. This is because more than one potential area may pass through a tcell. Requiring the tcell manager to completely handle all details of partitioning (including the pruning of area channels that should be disconnected) would involve non-deterministic actions on its part to support concurrent progress of partitioning on area channels of logically unrelated areas.

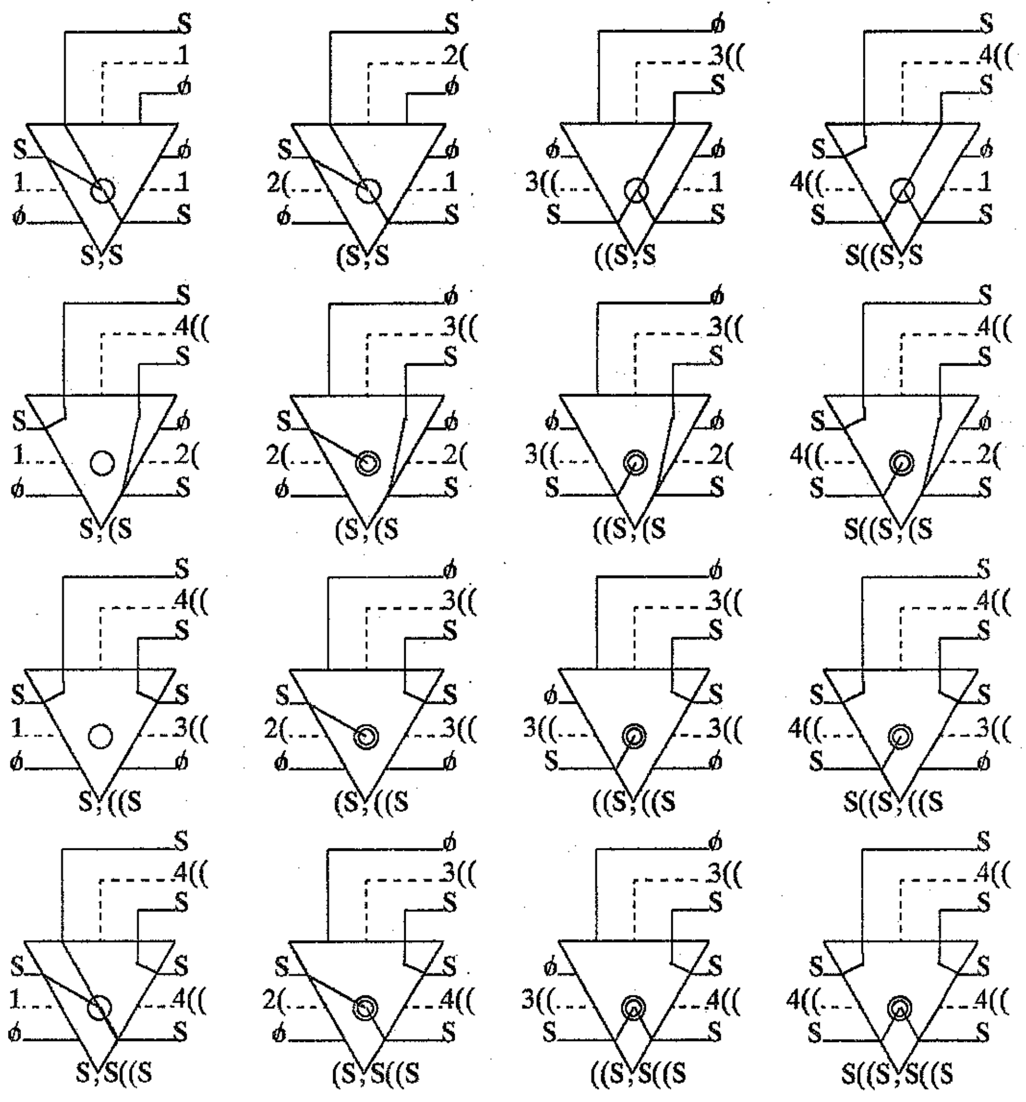
For this reason, support for partitioning is divided among the tcell manager processes and the node manager processes in a way that guarantees well balanced, efficient, and completely sequential processing by all involved processes. The SPF and the information associated with $(_l$ and $(_r$ is sent upwards and received by tcell manager processes from both left and right tcell manager children. The information associated with S_l and S_r is sent upwards and received by the area node manager processes from left and right node manager children, respectively.

When a tcell manager receives SPFs from both of its children, it has enough information, even before receiving the $(_l$ and $(_r$ which may follow, to send the required SPF to its parent, and perform an initial partitioning of the tcell by circuit-switching area channels. In general, some area channels will be switched to provide a direct routing through the tcell, and some will be connected to the input and output channels of the local area-handling node. This is the initial partitioning referred to earlier in the overall discussion of the DOT machine cycle. When the tcell manager receives left and right values of $(_l$ and $(_r$ as indicated by the left and right SPFs, it then has enough information to create and send merged $(_l$ and $(_r$ values to its parent (as required by the SPF it just sent up), and to signal its node manager process (if an area-node was just connected to area channels) to begin one of five possible tasks.

Once a node manager process is given the go-ahead by its cell manager, it processes the S_l and S_r from its left and right children, respectively. Depending on the task given to it by the tcell manager it will either start a pruning downsweep, because it now has access through the area channels to all necessary lcells below it, or send up an appropriately merged S value to its parent (which will be received either as S_l or S_r, depending on which side of its parent the area-node is located) and then await the pruning downsweep that will be started by an overlying node manager.

Figure 3.27 shows how the initial partitioning is done within an arbitrary tcell, and details the upwards moving flow of information on area and cell manager channels as well as the circuit-switched connections that are made by the cell manager. This figure completely specifies the algorithm used for merging segment descriptors. There are 4 SPFs, so there are 16 different possibilities for left and right SPF arrivals from child tcell managers. The area node within a tcell is depicted as a small circle. The tcell manager is not shown. Node managers with a double circle are those that are tasked to start pruning. Whether they will be active or not depends on the actual segment descriptor

FIGURE 3.27 – Analysis of the Initial Partitioning by Tcell Managers



field values that are received.

Subscripts for S_l and $(_l$, etc., are dropped since the SPF makes these redundant. In Figure 3.27, child area channels are shown entering from the left and right sides of the tcells (which are represented by triangles), and channels to support communication with a parent leave from the top of the tcells. Cell manager channels are dotted, and the area channels are solid.

As indicated in Chapter 2, two distinct area channels are provided to handle a situation where two areas pass through the same tcell. These area channels -- for convenience call them area_1 and area_2 -- must be correctly distinguished from each other during partitioning. For tcells in Figure 3.27, the top channel entering from a left child is always the area_1 channel and the bottom left channel is area_2. The top channel entering from a right child is always area_2 and the bottom left channel is area_1. The left channel leaving for a parent is always area_1, and the right top channel is area_2. This arrangement allows tcells to be connected together so that an area_1 channel leaving a child tcell will always be connected with an area_1 channel entering its parent.

The DOT specification for beginning the partitioning upsweep within an lcell is shown in Figure 3.28.

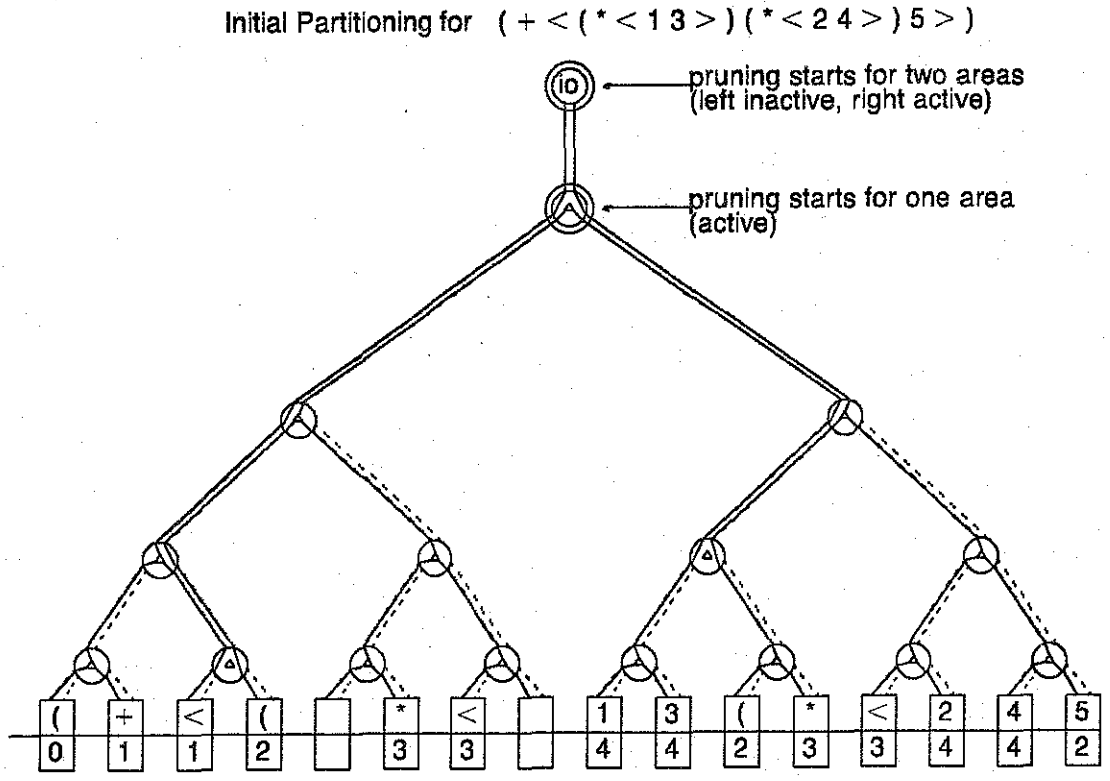
FIGURE 3.28 – The Start of Partitioning in the Lcells

```
/*
This is the lcell partitioning algorithm. Lcells start partitioning
by sending up the appropriate segment descriptor information on
the cell manager and area channels.
*/

void lcell.partition()
{
        /* initiate partitioning upsweep */
        if (user->state==EMPTY) {
                cm_tail->put(1);                /* spf = 1 */
                cm_tail->eot_alert();           /* no pgm requests */
                area_tail->put(0);              /* symbol-cnt = 0 */
                }
        else if ((user->symbol == APPSYMBOL) &&
                (user->fork_id == 1)) {         /* handle app. symbol */
                cm_tail->put(2);                /* spf = 2 */
                cm_tail->put(user->aln);        /* application aln */
                cm_tail->put(user->state);      /* application state */
                cm_tail->eot_alert();           /* no pgm requests */
                area_tail->put(0);      /* symbol-cnt = 0 */
                }
        else { /* handle regular or forked application symbol */
                cm_tail->put(1);                /* spf = 1 */
                cm_tail->eot_alert();           /* no pgm requests */
                area_tail->put(1);              /* symbol-cnt = 1 */
                area_tail->put(aln);  /* mln = aln */
                area_tail->put(aln);  /* lln - aln */
                area_tail->put(user->symbol);
                }

        /* terminate pruning downsweep */

        ... shown in Figure 3.32
}
```

3.4.1.2. Partitioning Downsweep – Pruning

The initial partitioning upsweep must be complemented with a pruning downsweep to complete the construction of active areas, and disconnect unnecessary channels. To motivate this, let's use the information provided in Figure 3.27 to do the initial partitioning upsweep for the small segment of FFP text shown in Figure 3.29.

FIGURE 3.29 -- An Example of Initial Partitioning


Initial Partitioning for (+ < (* < 1 3 >) (* < 2 4 >) 5 >)
io
pruning starts for two areas (left inactive, right active)
pruning starts for one area (active)
(0
+ 1
< 1
(2
* 3
< 3
1 4
3 4
(2
* 3
< 3
2 4
4 4
5 2

Besides connecting area channels for applications that will not be active, the initial partitioning may extend area connections for what will become an RA past the RA's rightmost lcell, thus incorrectly including symbols that are not part of the RA.* This is because FFP text symbols and aln values that occur between application symbols are not available to the tcell managers during the initial partitioning upsweep. This information is given to the node managers, however, and is used during the pruning downsweep to correctly prune off

* In the example, this is done for the rightmost symbol of the lcell array, "5" at level 2, which is part of the outermost application -- not the rightmost RA as the initial partitioning guesses.

rightmost portions of active areas created during the initial partitioning.

Each potential area created during the initial partitioning upsweep (composed of a separate set of connected area channels and area nodes) has a topmost or root node which initiates the pruning downsweep on command from its tcell manager. The pruning downsweep disconnects all nodes and area channels that do not lead to an lcell containing a symbol within an RA, and in addition, locates and creates the top-of-area node where LPL messages turn around within each active area overlying an RA. The information flow in the pruning downsweep is contained entirely within area channels connected during the upsweep, and is manipulated entirely by the node manager processes. Tcell managers are finished with all area-related duties as soon as they complete the initial partitioning within their tcell, and then become involved in relaying requests for LPL programs (these are signaled by the root node managers for active areas before they start pruning) up to the io subsystem.

The pruning information sent down within each set of connected area channels indicates whether or not these channels are required for supporting an RA. This decision is initially made by the node manager that starts pruning, and is subsequently used by lower level node managers to disconnect area channels and node managers that are not needed for the upcoming execution phase. Often during pruning of an active area, only the left or right child of a node is discovered to be part of the associated RA. In such a case, the node manager signals and then disconnects the unnecessary child, and circuit-switches the remaining child to the overlying parent node.*

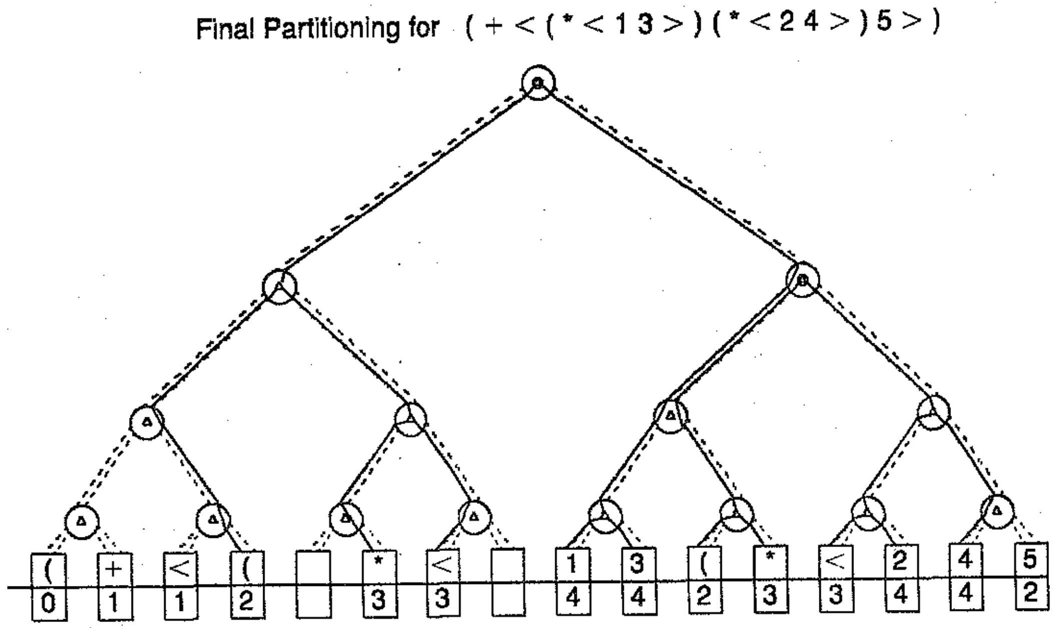
The top of area node is usually lower than where pruning starts. It is always located in the least common tcell ancestor of the lcells which comprise an RA,

* A node is required only if two children are present.

and always has two children. When the root of a newly discovered active area begins pruning, therefore, it checks to see if it has two active children, and if not, it disconnects itself after signaling both children appropriately. The first active node on the way down with two children becomes the top of area, and circuit-switches its up-going output channel to the input of the down-message process in the area node to implement message turn-around.

The result of pruning the example of Figure 3.29 is shown in Figure 3.30. Note how the top of area for the rightmost of the two active areas has moved down from where pruning starts in the io subsystem, and how the other area for which pruning starts in the io subsystem has been completely disconnected. Also, empty and non-active lcells have been correctly pruned from both RAs and

FIGURE 3.30 -- After Pruning the Initial Partitioning Example


Final Partitioning for (+ < (* < 1 3 >) (* < 2 4 >) 5 >)
(0
+ 1
< 1
(2
* 3
< 3
1 4
3 4
(2
* 3
< 3
2 4
4 4
5 2

area channels re-routed appropriately. The final position of the two top of area nodes is shown by circling their nodes.

In an active area channel, the pruning information contains the FFP operator, the aln of its application symbol, and whether the RA is new. When received in the lcells of an RA, this information indicates which LPL program the lcell-input process should look for if the RA is a new one, and if so, each lcell can use the application symbol aln to compute the local rln value and start an upsweep to compute the LPL environment directory. Figure 3.31 shows the specification for the start of pruning in an area node. Figure 3.32 shows the termination of pruning as it occurs in the lcells.

FIGURE 3.31 – The Area Node Algorithm to Start Pruning

```
/*
This is the algorithm for starting pruning.  Active is only the
tcell_mgr's guess, and will be wrongly false when an intermediate
symbol between two app symbols causes an RA. Such a situation is
checked for and taken care of here.  Lcnt and rcnt are the left and
right symbol-cnts received in the partitioning upsweep.
*/

node_mgr.start_pruning(active,rchild)
short active,rchild; /* both boolean */
{
  if (active || ((lcnt||rcnt)&&(mln<=aln))) /* we're really active */ {
    if ((lcnt && (lmln<=aln)) || !rcnt || (rcnt && (rlln<=aln))) {
      /* cut off in left child */
      top_of_area = FALSE;
      lnp_tail->put(pck(PARTLY_ACTIVE_TOA,aln));
      if (rchild) rnp_tail->put(pck(NOT_ACTIVE,aln));
      lnp_tail->put(pgm);
      lnp_tail->put(state); }
    else /* cut off in right subtree */ {
      top_of_area = TRUE;
      lnp_tail->put(pck(ALL_ACTIVE,aln));
      rnp_tail->put(pck(PARTLY_ACTIVE,aln));
      lnp_tail->put(pgm); rnp_tail->put(pgm);
      lnp_tail->put(state); rnp_tail->put(state); }
    if (top_of_area) wrap_head->connect(wrap_tail);
    finished = !top_of_area; }
  else /* we're not active */ {
    pgm = 0;
    top_of_area = FALSE;
    finished = TRUE;
    lnp_tail->put(pck(NOT_ACTIVE,0));
    if (rchild) rnp_tail->put(pck(NOT_ACTIVE,0)); }
} /* end of pruning initiation */
```

FIGURE 3.32 -- Termination of Pruning in the Lcells

```
/*
This is the lcell partitioning algorithm. Lcells start partitioning
by sending up the appropriate segment descriptor information on
the cell manager and area channels.
*/

void lcell.partition()
{
          /* initiate partitioning upsweep */

          ... shown above in Figure 3.28

          /* terminate pruning downsweep */
          unpck(area_head->msg(),&pflag,&aln);
          switch (pflag) {
          case NOT_ACTIVE:
            user->active = FALSE;
            break;
          case PARTLY_ACTIVE:
          case ALL_ACTIVE:
            user->active = TRUE;
            user->rln = user->aln - aln;
            user->lcell_pgm = area_head->msg();
            user->app_state = area_head->msg();
    } /* end switch */
} /* end partitioning */
```

3.4.2. Message Support

Next to partitioning, the most complex operations in DOT involve message handling. Many details associated with messages have already been covered in the discussion of the lcell message-input and message-output processes. What remains is to show how messages are handled within the overlying tree structure of an active area. The details of this operation are contained in the upwards-message class (called *node_umh* within DOT) used by the node-manager process. The *up_messages* entry of this class handles all message activity (in particular the sorting and merging of messages required by the LPL send statement) for an area node during a single execution phase.

All information that flows on area channels during the execution phase is broken up into individual packets. Each packet is introduced by a byte count, and packet data then follows immediately. Although a realistic implementation would deal with checksums and error recovery, this has not been done here. A reliable transmission mechanism is assumed.

3.4.2.1. Message Packets

As shown in Figure 3.33, there are four basic packet types. Three are used for LPL messages, and the fourth is used to signal the end of the execution phase and message activity.

FIGURE 3.33 – The Four Message Packet Types

Prefix Packet	≡	*: 2 : order : combine-op :*
Data Packet	≡	*: byte-cnt : data :*
Eow Packet	≡	*: 0 :*
Stop Packet	≡	*: 1 :*

LPL messages sent up from the lcells are composed of three packets. The first packet, a message introduction or ***prefix packet***, specifies the type of handling that is desired. This information is provided in the LPL **send** statement, and is composed of the sort-order and combine-operation information. The message prefix packet always has a byte count of two, and is merged in a single pipelined upsweep through the area. It is not returned to the lcells. Following the message prefix out of an lcell is the main message or *data packet.*

Each data packet has a byte count of at least two, since the *key1* and *key2* values in a send statement are always sent. Additionally, each requested message argument accounts for an additional byte. Thus the byte count for the message packet will be 2 + *msize* as specified in the send statement. Message bytes follow immediately to complete the packet.

Finally, each message out of an lcell is terminated with an end-of-wave, or *eow packet*. This packet contains no information, and has a zero byte count. When received by a node manager, it indicates that no further message packets for the current wave will be received on that channel. When an eow has been received by a node manager from both of its children, an eow is sent up to its parent. When the top of area node relays the eow, it is broadcast to all underlying lcells, the current message wave comes to an end, and the next one is started. An lcell that does not wish to send a message for a particular message wave sends just the eow packet for that wave.

The fourth packet type is a *stop packet*. Since processes in the DOT model are never interrupted, there must be some way to free up a node manager that is waiting for the next packet arrival from a child area channel after execution phase message processing in the lcells has come to an end. The stop packet is thus sent up on area channels by the lcell manager when the execution phase ends, and this guarantees correct flushing of area channels before they are disconnected. A stop packet has a byte count of one, and is special in that there is no following data.

3.4.2.2. Message Handling

The top-level for area node message handling is given in Figure 3.34. The approach is organized to allow pipelined operation. Initially, the prefix must be merged and sent up. This is handled in the *start-new-wave* entry, which reads and passes up the sort and combine selectors coded in the **send** statement. Besides passing it up, start-new-wave also loads this information into a data structure called *mspec*. Once this is done, message packets from the children of the node can be handled.

The general approach (assuming that eow has not been received from either child) is to read child message counts and pass a message count up to the node parent. Then key1 values from both children are read. The appropriate key1 is then sent up (based on the sort order) and the other saved in a buffer. Then key2 is handled, and the correct (selected) key2 is sent up, and the other saved in the appropriate buffer. This buffer holds key1 and key2 values for the "losing" message (i.e., the message that is not selected for immediate relay upwards). The rest of the message packet for the winning message is relayed up. Looping back to handle the next message, the key values for the message that lost out last time are already available, so the byte count for the next prefix following the successful message is read. Processing continues as before, but only one channel needs to be read to get key values this time.

If the key values for two messages entering a node are the same, the messages should be combined. The correct keys will already have been sent up, and the primary difference between sorting as explained above and combining is that a combined message is then created from the two entering messages, and the entering messages are thrown away. Following this, there will be no buffered key values (since the message packets from both children were used up), so processing continues as initially explained.

3.4.2.3. Stopping Messages

Besides pipelining messages as described above, the primary complication involved in message processing is knowing when to stop. There are two possibilities. First, the node manager has access to a memory location shared with the tcell manager which is set when the stop message comes down through cell manager channels to signal the end of the execution phase. This location is checked by the node manager before it attempts to read a new message packet, and, if the stop message has come through the tcell, the node manager immediately relays a stop packet to its parent, stops processing messages, and starts flushing them. Nothing further is sent to the parent.

It is also possible that the stop message may go through a tcell just after the node manager checks for it, so the node manager misses it and goes on to await the next message packet arrival. This is the reason for using a special stop packet. Even if node managers miss the stop message on its way down, they must see the stop packet as it rises from the lcells. In any case, messages are flushed until a rising stop packet is seen from both children. Nothing ever follows the stop packet up area channels.

When the stop packet is relayed through the toa, and is detected back at the lcells, the lcells know that all LPL messages for the current execution phase have been received and that they can shut down and save their LPL programs.

FIGURE 3.34 -- Upwards Messages in a Tcell Node

```
node_umh.up_messages() /* hand up messages for one machine cycle */
{ mspec.valid = FALSE;          /* don't know how to handle message wave yet */
  while (!cell_mgr->node_eom) { /* stop message hasn't come down yet */
    if (!mspec.valid) start_new_wave(); /* get handling instructions */
    if ((lbuf.msize!=STOPCNT) && (rbuf.msize!=STOPCNT)) {
      /* handle next message or eow for present wave */
      if (!lbuf.full && lbuf.msize>EOW) lbuf.msize=lhead->msg();
      if (!rbuf.full && rbuf.msize>EOW) rbuf.msize=rhead->msg();
      if ((lbuf.msize!=STOPCNT) && (rbuf.msize!=STOPCNT) &&
        (lbuf.msize>EOW || rbuf.msize>EOW)) {
        /* handle message. start with message size */
        msize = (lbuf.msize>rbuf.msize)?lbuf.msize:rbuf.msize;
        handup(msize);
        mselect=2; /* assume equal keys initially */
        /* accept key1 values and see if ordered yet */
        if (!lbuf.full && lbuf.msize>EOW) lbuf.key1=lhead->msg();
        if (!rbuf.full && rbuf.msize>EOW) rbuf.key1=rhead->msg();
        mselect=select(lbuf.key1,rbuf.key1);
        switch (mselect) {
          case 0: handup(lbuf.key1); break;
          case 1: case 2: handup(rbuf.key1); }
        /* accept key2 values and see if ordered if not already */
        if (!lbuf.full && lbuf.msize>EOW) lbuf.key2=lhead->msg();
        if (!rbuf.full && rbuf.msize>EOW) rbuf.key2=rhead->msg();
        lbuf.full = rbuf.full = TRUE;
        if (mselect==2) mselect=select(lbuf.key2,rbuf.key2);
        switch (mselect) {
          case 0: handup(lbuf.key2); break;
          case 1: case 2: handup(rbuf.key2); }
        /* relay or merge to produce result message */
        switch (mselect) {
          case 0: up_remaining(lhead,lbuf.msize-2); break;
          case 1: up_remaining(rhead,rbuf.msize-2); break;
          case 2: combine(); }
        } /* end handing up one message */
      else { /* either end of wave from both children, or eom */
        mspec.valid = FALSE;
        if (lbuf.msize!=STOPCNT && rbuf.msize!=STOPCNT)
          handup(EOW); /* it was an end of wave */
        } /* end handling eow or eom */
          } /* end handling message or eow/eom */
    } /* end execution phase */
  handup(STOPCNT);
  flush_messages(lhead,&lbuf); flush_messages(rhead,&rbuf);
}
```

3.4.3. Directory Creation

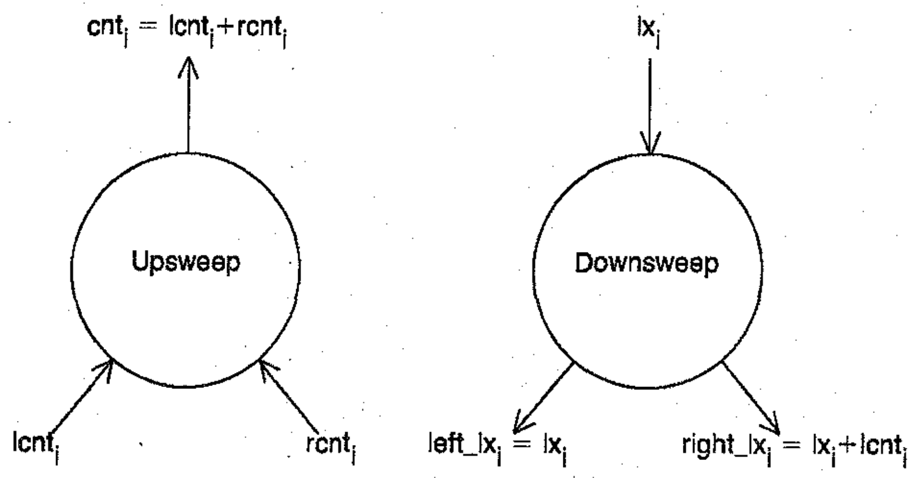
As described in Chapter 2, the LPL directory is composed of a ***symbol index*** and a ***directory tuple***. We now present the algorithms used within the lcells and the tcell area nodes to compute and initialize these values for the LPL code segments that execute within RAs. The correctness of the algorithms for directory creation is established using the principle of mathematical induction.*

3.4.3.1. Computation of the Symbol Index

Computation of the LPL environment variable *symbol_index* is performed using an upsweep to accumulate information into the area nodes, and a downsweep to distribute the correct information to the lcells. Figure 3.35 summarizes the information flow. Note that a merge operation (addition) is performed by area nodes during both the upsweep and downsweep. During the upsweep, a result based on two incoming values is sent to the parent; during the downsweep, a result based on the incoming value and a value remembered from the upsweep is sent to the right child.

* Mathematical induction is based on the idea of inheritance -- the idea that if P(i) is true, then so is P(i+1). This is called an inductive hypothesis, and establishment of P(i) for a particular i is called an inductive base. An inductive proof must show that the inductive hypothesis is true for all i of interest and give an inductive base in order to establish P(i) for all i that are successors to the base.

FIGURE 3.35 – Diagram for Symbol Index Calculation



3.4.3.1.1. Upsweep

For the upsweep, let $P_u(j)$ mean that the number sent up from each area node at level j correctly represents the count of all the leaf cells in its subtree. Call this number cnt_j. Now, if $P_u(j)$ is true, and each node at level i=j+1 receives the resulting cnt_j values passed up by its left and right children as $lcnt_i$ and $rcnt_i$ respectively, and then sends up

$$\boxed{cnt_i = lcnt_i + rcnt_i}$$

then clearly $P_u(i=j+1)$ holds as well. This establishes an inductive hypothesis for the upsweep. An inductive base is provided by the lcells of an area, which send up 1. Because $P_u(0)$ is therefore true (the correct symbol count within an active lcell is always 1), $P_u(\text{toa-1})$ also holds by induction. At the toa, then, following such an upsweep, we know that the symbol index of the leftmost symbol of the left area subtree is 0, and the symbol index of the leftmost symbol of the right area subtree is $lcnt_{toa}$ (because this is the count of the number of symbols in the left area subtree). These values provide an inductive base for the downsweep.

3.4.3.1.2. Downsweep

For the downsweep, assume the *lcnt* values have been saved by the nodes during the preceding upsweep, and let $P_d(k)$ mean that the numbers sent down to the left and right children of level k nodes correctly represent, respectively, the true symbol index of the leftmost symbol in the left child's area subtree, and the true symbol index of the leftmost symbol in the right child's area subtree. Call these numbers $left\_lx_k$ and $right\_lx_k$ respectively. If $P_d(k)$ is true, and each node at level i=k-1 receives the resulting value passed down by its parent as lx_i, and then sends to its left and right children the values

$$\boxed{\begin{array}{l} left\_lx_i = lx_i, \textit{ and} \\ right\_lx_i = (lx_i + lcnt_i) \end{array}}$$

then $P_d(i)$ also holds.* This establishes an inductive hypothesis for the downsweep. An inductive base is provided by the toa which sends down $left\_lx_{toa}$ = 0 and $right\_lx_{toa} = lcnt_{toa}$. Because P_d(toa) is therefore true (as noted at the end of the upsweep discussion), $P_d(0)$ also holds by induction. Since at level 0 (the lcell level) the leftmost symbol is the only symbol, the value received there as lx_0 is the desired symbol index. Figure 3.36 gives the portions of the lcell and area node directory creation algorithms responsible for creating the symbol index. As can be seen, there is a close correspondence with the above inductive reasoning.

* Clearly, if lx_i is the beginning symbol index of the left subtree, the beginning symbol index for the right subtree is lx_i plus the count of symbols in the left subtree.

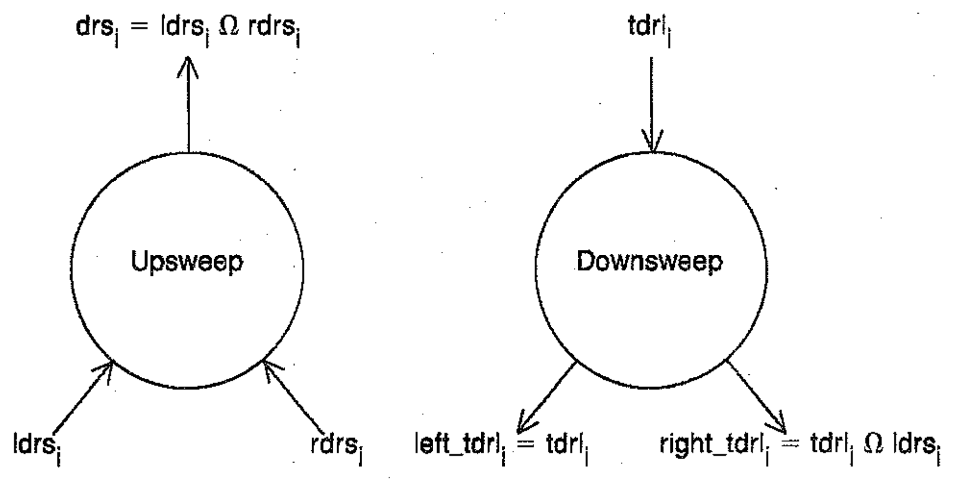
FIGURE 3.36 – Algorithms for Symbol Index

```
lcell.build_directory() /* symbol index portion */
{
        /* upsweep -- establish inductive base */
        area_tail->put(1); /* send up symbol count for this level */

        /* downsweep – terminate using inductive hypothesis */
        user->symbol_index=area_head->msg(); /* receive symbol_index */
}

node_mgr.build_directory() /* symbol index portion */
{
        /* upsweep -- preserve inductive hypothesis */
        lcnt = lnp_head->msg(); rcnt = rnp_head->msg();
        if (!top_of_area) np_tail->put(lcnt+rcnt); /* send up total count */

        /* downsweep to place symbol_index and addresses into leaves */
        if (top_of_area) {
                /* start downsweep -- establish inductive base */
                lnp_tail->put(0);               /* left-lx for toa */
                rnp_tail->put(lcnt);}           /* right-lx for toa */
        else {  /* continue downsweep – preserve inductive hypothesis */
                lx = np_head->msg();            /* get lx */
                lnp_tail->put(lx);              /* send left-lx */
                rnp_tail->put(lx+lcnt);}    /* send right-lx */
}
```

3.4.3.2. Computation of the Directory Tuple

Computation of the directory tuple also involves an upsweep to accumulate information into the area nodes, and a downsweep to distribute the desired information to the lcells. While the symbol index computation uses a simple merge function (addition) and P_u and P_d predicates based on the values of single numbers passed between the lcells and nodes, the directory tuple computation involves a more complex merge function and P_u and P_d predicates based on tuple values. Figure 3.37 summarizes the information flow. A merge operation (Ω, to be described below) is performed by area nodes during both the upsweep and downsweep. During the upsweep, a result based on two incoming values is sent to the parent; during the downsweep, a result based on the incoming value and a value remembered from the upsweep is sent to the right child.

FIGURE 3.37 – Diagram for Directory Tuple Calculation


$drs_i = ldrs_i \Omega rdrs_i$
Upsweep
$ldrs_i$
$rdrs_i$
$tdrl_i$
Downsweep
$left\_tdrl_i = tdrl_i$
$right\_tdrl_i = tdrl_i \Omega ldrs_i$

The definition of the LPL environment directory tuple, $D=[d_1 \ldots d_n]$, was given in Chapter 2. As explained in Section 2.3.1.2 with the help of Figures 2.12 and 2.13, the directory tuple for an FFP text symbol is directly related to the parse tree of its RA, and the value of a general d_j represents the left-to-right count of RA symbols at level j (up to the symbol whose directory is of interest) that are within the scope of the last constructor (sequence or application symbol) with nesting level j-1.* To provide a frame of reference for the following discussion, Figure 3.38 contains an example RA located within an active partition of a DOT machine. This figure represents a typical active area; area channels have been pruned during partitioning, and as a result the area is not height balanced.

* The level of a symbol in the parse tree for its RA is the ***rln*** value that is locally computed by lcells upon learning the ***aln*** of the RA application symbol.

FIGURE 3.38 – An Example RA Within an Active Area

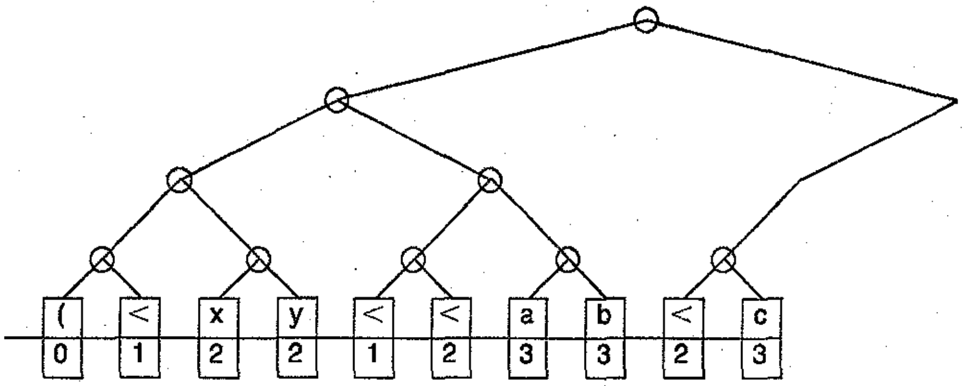
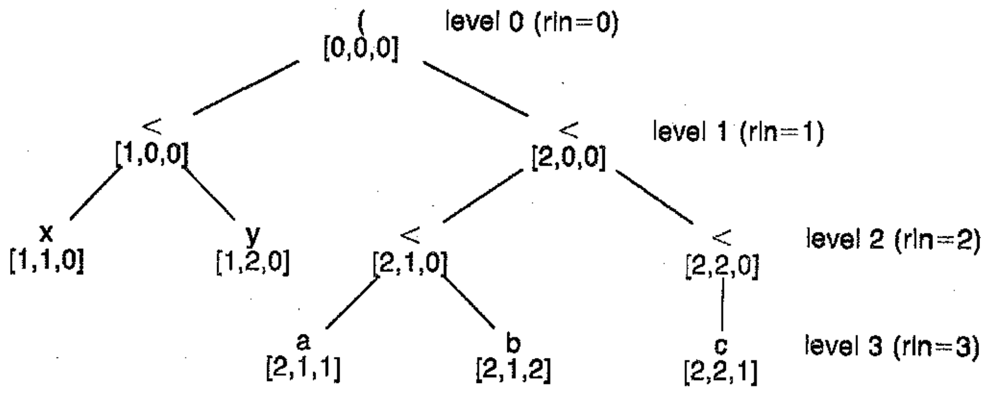


Figure 3.39 shows the parse tree corresponding to the RA of Figure 3.38, and gives directory tuples with truncation at level 3. The directory tuple for "b" is [2,1,2] -- d_1 is 2 because there are two level 1 symbols (the operator and the argument sequence symbol) within the scope of the last level 0 symbol (the application symbol) before "b"; d_2 is 1 because there is 1 level 2 symbol (the second sequence symbol) within the scope of the last level 1 symbol (the argument sequence symbol) before "b"; d_3 is 2 because there are 2 level 3 symbols ("a" and "b") within the scope of the last level 2 symbol before "b".

FIGURE 3.39 -- Example Parse Tree and Directory Tuples



We now consider the tuple values that are passed in the upsweep and downsweep to compute the directories for the symbols of an RA. We make no assumptions here concerning the level of truncation. DOT presently provides LPL programs with a four-level directory, but the following algorithms work with truncation at any level. All examples will perform truncation at level 3, as in Figure 3.39.

3.4.3.2.1. Upsweep

For the upsweep, let $P_u(j)$ mean that the tuple value sent up by each node at level j correctly represents the directory of the rightmost symbol within its subtree (i.e., correct relative to only those symbols in the subtree). Call this tuple drs_j (for relative directory of the rightmost symbol).

In order to establish an inductive hypothesis for the upsweep, we must determine an operation Ω such that if each node at level i=j+1 receives the *drs* tuple values passed up by its left and right children as $ldrs_i$ and $rdrs_i$ respectively, and sends up

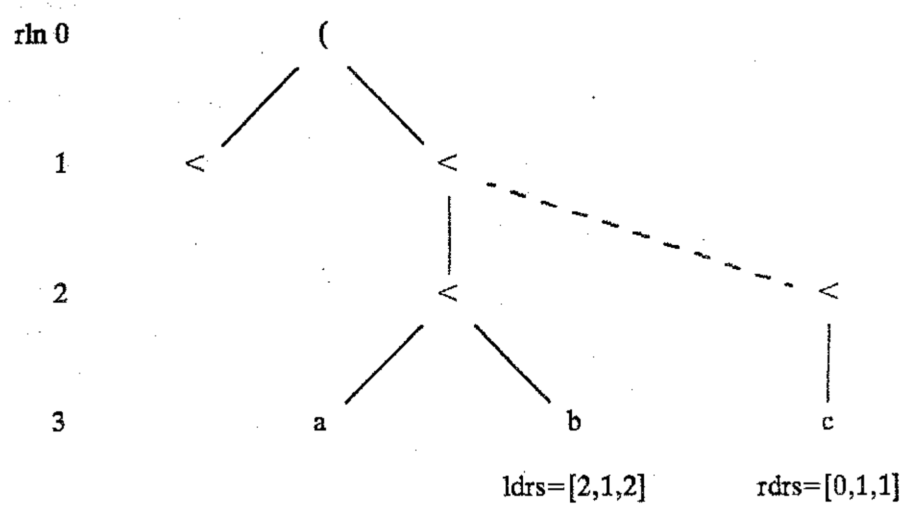
$drs_i = ldrs_i \ \Omega \ rdrs_i$

then $P_u(i=j+1)$ also holds. The appropriate operation for the symbol_index calculation was addition; here we must refer to the definition of the directory tuple for guidance. The trick is to realize that we are merging truncated representations of partial parse trees.

A directory tuple contains certain information about the parse tree from which it is derived. It doesn't contain *all* information about the parse tree because the d_j values of a directory tuple only represent the count of symbols within limited scopes, as indicated by the definition of the directory tuple. Each *drs* tuple sent on the upsweep therefore represents a class of parse trees which conform to the structure implied by that tuple. The merge operation which Ω must reflect is the joining of two such parse tree classes into a new one, and the representation of the result with a new directory tuple (i.e., *drs* value). Figure 3.40 shows the two *drs* tuples that are received by the top area node of Figure 3.38, and portrays the partial parse trees that are implied by these tuples. The *ldrs* tuple is the relative directory of "b", and the *rdrs* tuple is the relative directory of "c".*

* These values are easily calculated using the definition of the directory tuple (within the confines of the respective subtrees). What we must discover is how to define Ω so that these values are actually sent during the upsweep.

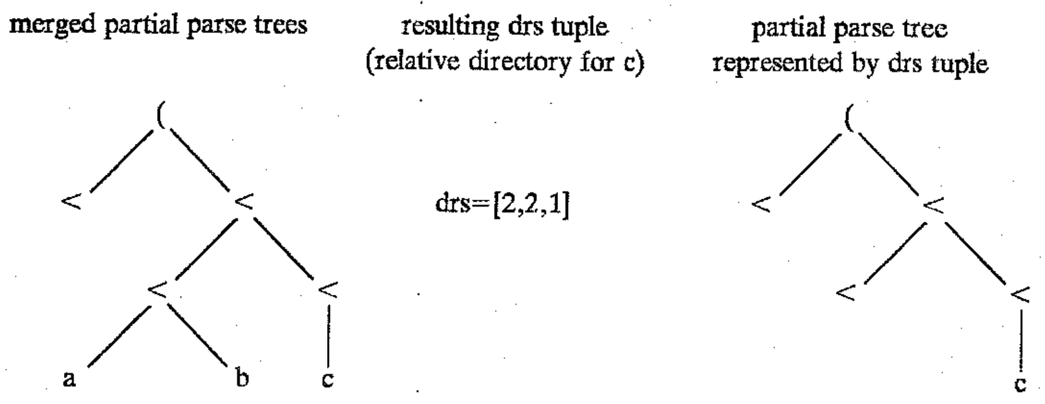
FIGURE 3.40 – Before Merging Partial Parse Trees



Note that the parse tree class implied by *ldrs* in Figure 3.40 does not include nodes for all the operator symbols of the actual RA. This is a result of the limited precision of *ldrs*, the relative directory of "b". Truncation effects are absent because no symbol of the RA is nested deeper than three levels.

During the upsweep, partial parse trees represented by *ldrs* and *rdrs* are merged in the obvious way -- by connecting them according to their implied levels. The dotted line in Figure 3.40 illustrates this, and Figure 3.41 shows the result of merging the partial parse trees of Figure 3.40. The rightmost symbol in the result is "c", and from the definition of the directory tuple we know that its relative directory is [2,2,1]. For completeness, the partial parse tree implied by a directory tuple of [2,2,1] is shown on the right in Figure 3.41.

FIGURE 3.41 – After Merging Partial Parse Trees



We now give a procedure to calculate the result of the merge operation Ω on two directory tuples D_l and D_r.

> *To get result tuple values, from left to right add D_l and D_r directory values ($d_{lj} + d_{rj}$) to produce corresponding result values, with the following exception: after the first non-zero d_r value has been encountered, following d_r values are the correct corresponding result values.*

This procedure produces the correct result of [2,2,1] for the example in Figures 3.40. That it will always produce the correct result may be seen from the following reasoning. The objective of Ω is to produce the directory tuple of the rightmost symbol of two merged partial parse trees, and an *rdrs* value is already the correct directory for this symbol within the context of its containing subtree. The d_j values following the first non-zero *rdrs* directory entry therefore require no modification when the left context implied by *ldrs* is also considered. This is because the definition of the directory tuple requires that d_j values represent a count of level j symbols only within the scope of the last symbol at level j-1. Leading zero directory values preceding the first non-zero *rdrs* directory entry indicate that the corresponding partial parse tree has no symbols at these levels. In this case, therefore, left context symbol counts represented by the *ldrs* directory tuple should be included in the resulting *drs* tuple because the symbols represented by the *rdrs* tuple must be within the scope of sequence symbols located in the left context.* The result corresponding to the first non-zero *rdrs* value requires addition since, as shown in Figure 3.40, this symbol should be counted along with others at the same level that are within the scope of the last symbol with less nesting (which symbol, if it exists, is in the left context implied by *ldrs).*

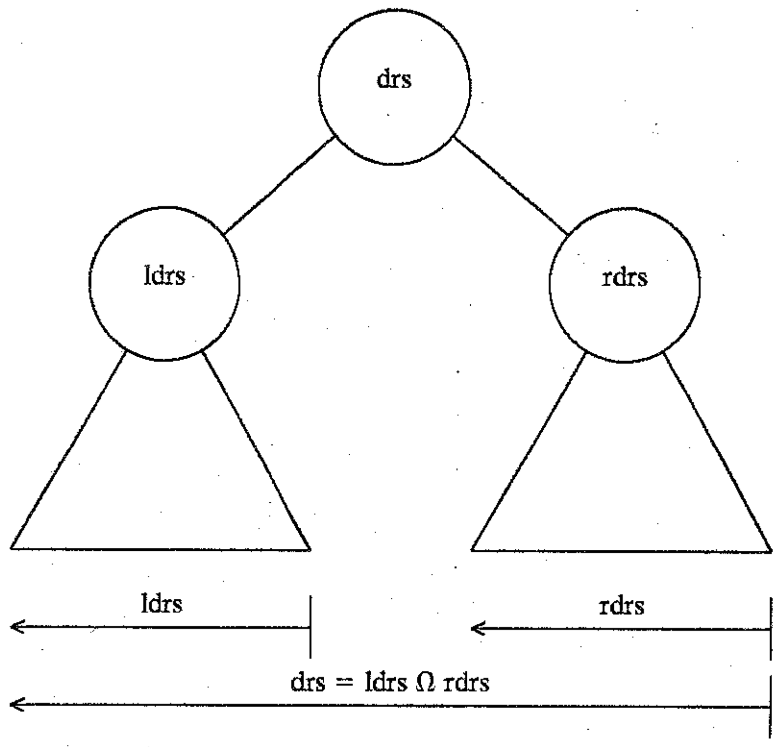
Thus, on the upsweep, in order to always send up the relative directory of the rightmost underlying symbol, a general node i should send

$$\boxed{drs_i = ldrs_i \, \Omega \, rdrs_i}$$

with Ω computed as described in the above procedure. Figure 3.42 shows how the *ldrs* and *rdrs* relative directories for two subtrees are used by their parent to determine a *drs* relative directory. This figure also portrays the scope of the relative directories involved.

* In the procedure for calculating Ω, addition of leading zero *rdrs* directory values to the corresponding *ldrs* values to produce a result is equivalent to simply using the *ldrs* values.

FIGURE 3.42 – Directory Upsweep Computation



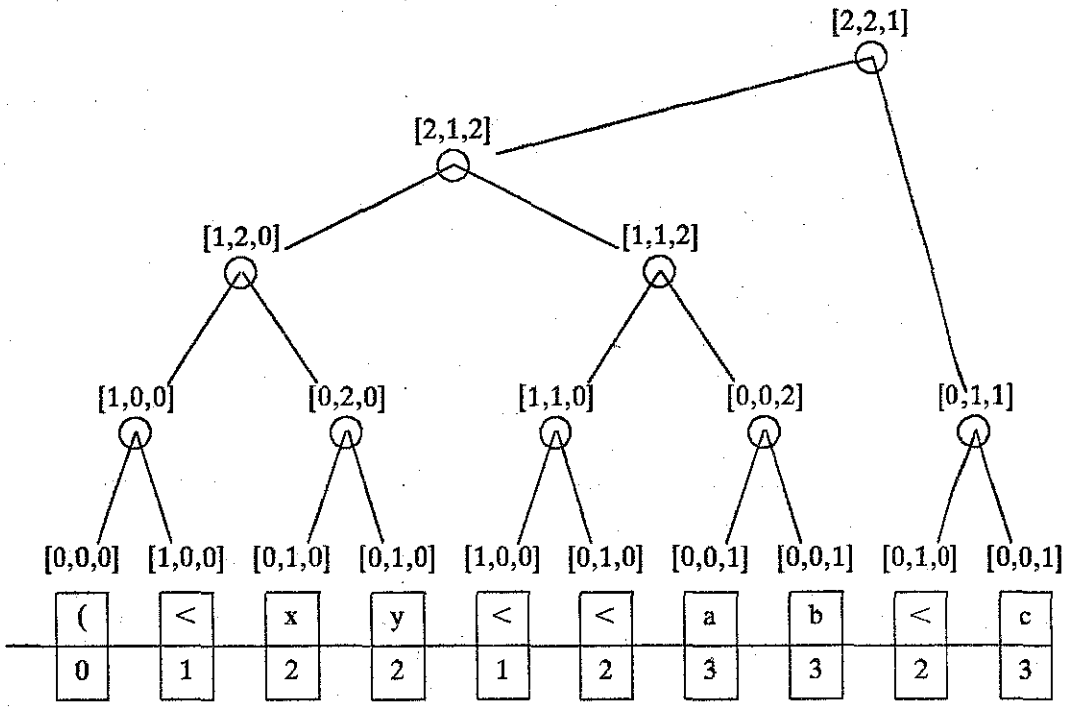
With the merge operation taken care of, and thus our inductive hypothesis, all that is left for the upsweep is a basis step. The definition for the directory tuple indicates that the correct directory tuple, drs_0, for an lcell symbol is

$$drs_0 = \delta_{rln} \equiv D=[d_1 \ldots d_n], \text{ where}$$

$$d_j = 0 \text{ for } j \neq rln$$
$$d_j = 1 \text{ for } j = rln$$

Figure 3.43 shows the example of Figure 3.38, and includes the *drs* tuples that are calculated within each area node during the upsweep.

FIGURE 3.43 – An Example Upsweep



With lcells and area nodes behaving as described above, $P_u(0)$ is true (the δ_{rln} tuple sent up by an lcell is the correct relative directory of a contained symbol), therefore, by induction, $P_u(\text{toa-1})$ holds as well. At the toa, then, following such an upsweep, we know the correct directory for the rightmost symbol of the left area subtree (it is $ldrs_{toa}$ since there are no RA symbols to the left of the left area subtree to change this value). Since this directory tuple is correct with respect to the entire RA we call it the true directory for the symbol of interest. Also, since there are no symbols to the left of the left area subtree, we know the true directory for the rightmost of these symbols (there are none, so it is [0 ... 0], vacuously).

3.4.3.2.2. Downsweep

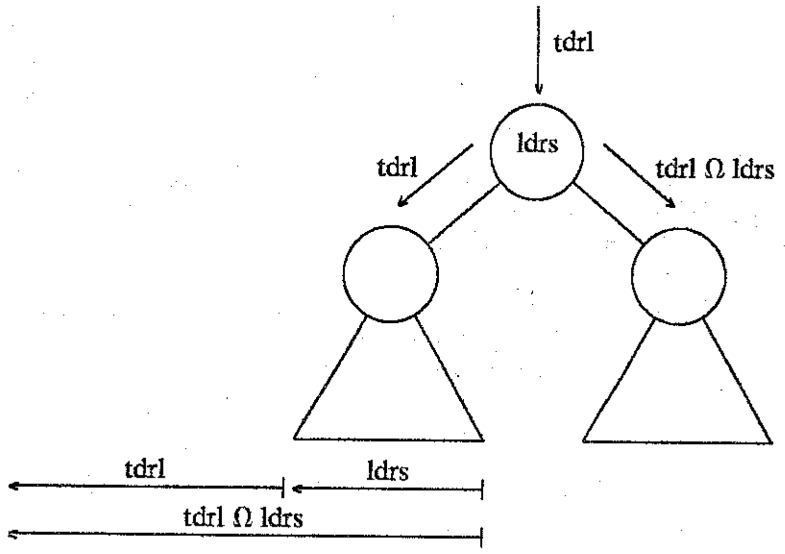
For the downsweep, assume the ***ldrs*** values have been saved by the nodes during the preceding upsweep, and let $P_d(k)$ mean that the tuples sent to left and right children of level k nodes correctly represent the true directory of the rightmost symbol to the left of their subtree. Call these tuples $left\_tdrl_i$ and $right\_tdrl_i$ (where tdrl stands for "true directory of the rightmost symbol to the left"). If $P_d(k)$ is true, and each node at level i=k-1 receives the resulting tuple passed down by its parent as $tdrl_i$, and then sends to its left and right children the values

$$\boxed{\begin{aligned} &left\_tdrl_i = tdrl_i \\ &right\_tdrl_i = tdrl_i \,\Omega\, ldrs_i \end{aligned}}$$

then $P_d(i{=}k{-}1)$ also holds.[*] This establishes an inductive hypothesis for the downsweep. Figure 3.44 shows how the ***tdrl*** tuple passed down by a parent and the saved ***ldrs*** tuple are used to determine the values to be passed to subtrees. Also included in this figure are the scopes of the relative directories involved.

[*] The reasoning is similar to that used for the upsweep. The received ***tdrl*** tuple indicates the directory of the rightmost symbol to the left of the subtree of the receiving node, and the ***right_tdrl*** (to be sent to the node's right child) must indicate the directory of the rightmost symbol of the left child subtree. Therefore the partial parse trees represented by ***tdrl*** and the saved ***ldrs*** tuple are merged to produce this result.

FIGURE 3.44 -- Directory Downsweep Computation



An inductive base is provided by the toa, which sends down $left\_tdrl_{toa}$ = [0 ... 0] and $right\_tdrl_{toa} = ldrs_{toa}$. Since P_d(toa) is true (as noted at the end of the upsweep discussion), $P_d(0)$ also holds by induction. Since at level 0 (the lcell level) the value that would be sent down as $right\_tdrl_0$ is the true directory of the symbol stored therein, this tuple is the desired LPL environment directory tuple.* Of course this directory tuple is not passed down further, but is kept as the local LPL environment directory tuple.

* The lcells use δ_{rln} for the *ldrs* value to merge with the received *tdrl* value. This is correct, because at the lcell level there is but one symbol to consider.

FIGURE 3.45 -- An Example Downsweep

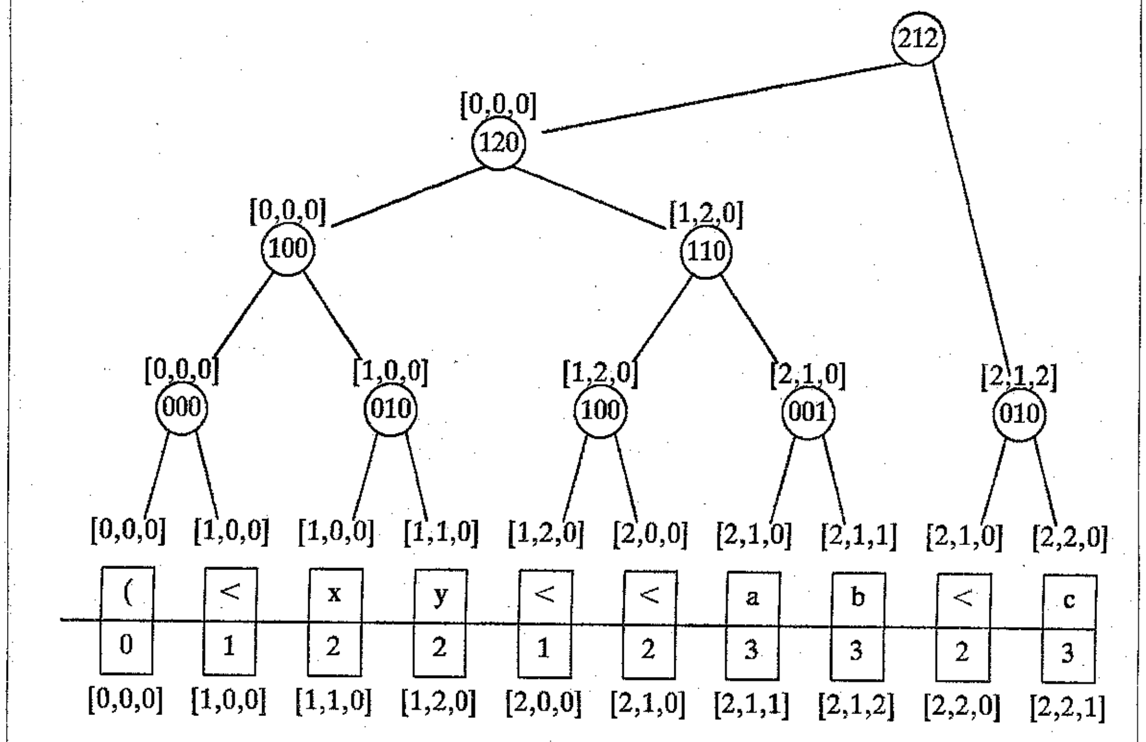


Figure 3.45 continues the example of Figure 3.43 by showing the *left_tdrl* and *right_tdrl* tuples that are sent during the downsweep. Tuples in this figure are displayed above the area nodes to which they are sent. Tuples beneath the lcell array are the correctly computed directory values for the LPL environments. While examining this figure, recall that the tuple value received by a node represents the context to the left of its complete subtree, and the *ldrs* tuple that is held within the node (shown using a compressed format with no brackets or commas) represents the left child context. The *ldrs* values used by the lcells to finally determine the directories are not shown, but are implied by the *rln* values locally stored.

Figure 3.46 gives the portions of the lcell and area node directory creation algorithms responsible for creating the directory tuple. As can be seen, there is a close correspondence with the above inductive reasoning.

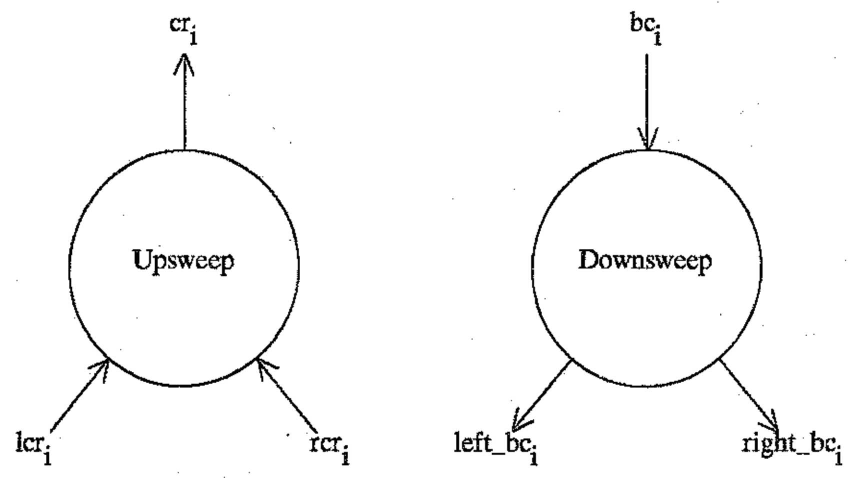
FIGURE 3.46 – Algorithms for Directory Tuple

```
lcell.build_directory() /* the directory tuple portion */
{
char *dir = user->directory; short i;

        /* upsweep -- establish inductive base */
        for(i=0; i<DLEVELS; i++)  /* send up correct drs tuple */
                area_tail->put((*(dir+i) = (user->rln==i+1)?1:0));

        /* downsweep – use inductive hypothesis to terminate */
        for (i=0; i<DLEVELS; i++) /* receive tdrl and merge */
                if (i<user->rln) *(dir+i) += area_head->msg();
                else area_head->get(&garbage);
}

node_mgr.build_directory() /* the directory tuple portion */
{
char tdrl, rdrs, ldrs[DLEVELS], zeros, i;

        /* upsweep -- get ldrs and rdrs tuples and merge */
        zeros = TRUE; /* all rdrs d have been zero so far */
        for (i=0; i<DLEVELS; i++) {
                ldrs[i] = lnp_head->msg(); rdrs = rnp_head->msg();
                if (!top_of_area) np_tail->put
                        ((zeros)?rdrs+ldrs[i]:rdrs);
                zeros &= (rdrs == 0);
                }

        /* downsweep – get tdrl and send left- and right-tdrl */
        if (top_of_area) for (i=0; i<DLEVELS; i++) {
                /* must start downsweep */
                lnp_tail->put(0);       /* left-tdrl for toa */
                rnp_tail->put(ldrs[i]);/* right-tdrl for toa */
                }
        else  {
                /* must continue downsweep */
                zeros = TRUE; /* all ldrs d have been zero so far */
                for (i=0; i<DLEVELS; i++) {
                        tdrl = np_head->msg();
                        lnp_tail->put(tdrl);  /* left-tdrl */
                        rnp_tail->put(                 /* right-tdrl */
                                (zeros)?ldrs[i]+tdrl:ldrs[i]);
                zeros &= (ldrs[i]==0); }
                }
}
```

3.4.4. Calculating the Specification for Storage Management

The algorithms used for calculating the specification for storage management involve support by lcell managers and tcell managers, as opposed to the use of node managers for directory creation. The method used involves an upsweep and downsweep of information through the entire tree, and can be analyzed in the same way as directory creation. Figure 3.47 summarizes the information flow.

FIGURE 3.47 -- Calculating a Specification for Storage Management



Both upsweep and downsweep use information tuples composed of two values. On the way up, the information sent is the capacity of a subtree (i.e., how many lcells are in the subtree), and the storage requests of a subtree (i.e., how many lcells are required by the local FFP text and executing LPL contexts for the next execution cycle). In the downsweep, shift values are sent to describe the number of lcell user contexts that should be shifted into a subtree through its left lcell boundary, and the number to be shifted out through its right lcell boundary. These shift values are signed; positive values indicate right shifting, and negative values indicate left shifting. When the shift values reach the lcell level, they form the specification for storage management for each lcell.

3.4.4.1. Upsweep

For the upsweep let $P_u(j)$ mean that each node at level j of the tree of processors sends up to its parent the correct capacity of its subtree, and the correct request total from its underlying lcells. Call these values cap_j and req_j, and their combination into a tuple cr_j. Now, if $P_u(j)$ is true, and every node at level i=j+1 receives the resulting values passed by its left and right children as lcr_i and rcr_i respectively, and then sends up

$$cr_i = \quad \begin{array}{l} cap_i = lcap_i + rcap_i \\ req_i = lreq_i + rreq_i \end{array}$$

then clearly $P_u(i{=}j{+}1)$ holds as well. This establishes an inductive hypothesis for the upsweep. An inductive base is provided by the lcells of the machine, which send up values as follows:

$$\begin{array}{lll} cap_0 = & 1 & \\ req_0 = & & \\ & (lcell\ empty) & \rightarrow 0 \\ & (otherwise) & \rightarrow usercontext.forkn^{*} \end{array}$$

Since $P_u(0)$ is therefore true, $P_u(io\text{-}1)$ is also true by induction. At the io node, then, following such an upsweep, we know the capacity of the entire tree, and the total number of lcells requested for the next cycle.

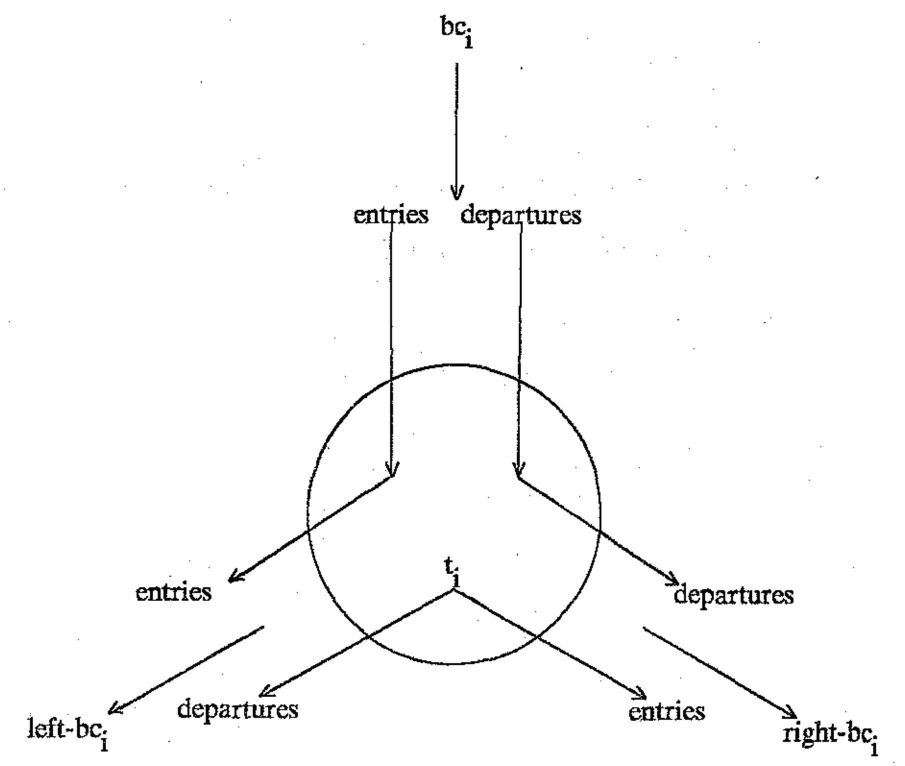
* As stated earlier, execution of an LPL fork statement modifies the forkn context variable which starts every active execution phase with a value of one. Non-active lcells always send one.

3.4.4.2. Downsweep

For the downsweep, assume that the *lcr* and *rcr* tuples have been saved by the tcells during the preceding upsweep, and let $P_d(i)$ mean that the numbers sent down to left and right children of level i nodes represent boundary conditions that allow feasible solutions to the storage management problem within their respective subtrees. Call these boundary conditions *left_bc*$_i$ and *right_bc*$_i$ respectively, where a boundary condition is a tuple composed of a *left_entry* and a *right_departure* value.

By feasible solution, we mean that there exists a solution to the storage management problem locally within a child subtree that is consistent with the left and right boundary conditions which the subtree tcell root receives from a parent at level i+1 as bc_i. Figure 3.48 shows that a solution to the storage management problem for a subtree rooted at level i is represented by a single transfer value, t_i, that describes the direction and amount of context flow that should take place on the lcell shift register that joins the two child subtrees rooted at level i-1.

FIGURE 3.48 – Transfer Value Solution for Storage Management



As shown in Figure 3.48, once t_i, a solution for storage management, is computed by the tcell at level i, it can send off *left_bc* and *right_bc* tuples to its left and right children as follows:

```
left_bc_i =
        left_entries       = bc_i.left_entries
        right_departures   = t_i
right_bc_i =
        left_entries       = t_i
        right_departures   = bc_i.right_departures
```

Calculation of t_i is analogous to the downward merge functions of directory creation. Let us therefore define the transfer function (call it Γ) as a function that takes three tuples (lrc_i and rrc_i from the upsweep, and bc_i from the downsweep) and produces a t_i that specifies the transfer of contexts between the two subtrees of a tcell. This function is not unique; a variety of approaches toward allocation of lcells to user contexts is possible.

Our approach is that suggested originally by Mago [Mag79] -- contexts are moved between left and right subtrees of a tcell only if absolutely necessary, and then the minimum possible number of contexts are shifted. This heuristic reduces information shifting between subtrees. It therefore avoids total compaction of contexts within adjacent lcells, which is important to an efficient storage management phase. As will be discussed in Chapter 6, the method is not optimal in minimizing the maximum distance for symbols to be shifted, but it is efficiently implemented with a minimum of information flow within the tree. In any case, it is not clear that a locally optimal storage management during one phase will necessarily produce the best long-term performance over a number of cycles. More study of this tradeoff is required.

Figure 3.49 presents the procedure presently used to calculate the transfer function.

FIGURE 3.49 – Calculation of the Transfer Function

```
Γ(lrc,rrc,bc)
{
        if lrc.capacity >= (lrc.requests + bc.leftentries)

                /* no need to shift right between subtrees */
                if rrc.capacity >= (rrc.requests - bc.rightdepartures)

                        /* no need to shift between subtrees */
                        Γ=0;

                else    /* need to shift left between subtrees */
                        Γ = rrc.capacity -
                                (rrc.requests - bc.rightdepartures)

        else    /* need to shift right between subtrees */
        Γ = (lrc.requests + bc.leftentries) - lrc.capacity
}
```

With the merge operation taken care of (i.e., calculation of the $t_i = \Gamma(lrc_i, rrc_i, bc_i)$), and with the resulting left_bc$_i$ and right_bc$_i$ thus determined as shown in Figure 3.49, the inductive hypothesis for the downsweep is established, and all that is left is establishment of an inductive base.

The left_bc$_{io}$ tuple goes to the vm subsystem, and right_bc$_{io}$ tuple goes to the processor tree. Calculation of the transfer function at this level is done with bc.left_entries and bc.right_departures both implicitly zero.[*]

A heuristic used to determine a t_{io} value (required to start the downsweep) need only generate feasible boundary conditions for the vm subsystem and the processing tree, and our approach is as follows. Overflow from the processing tree *must* be accommodated by the vm subsystem, so if there is overflow, t_{io} is set to the amount of overflow required (actually its negative, to indicate left shifting into the vm subsystem). If overflow is not required, as much previous

[*] This is another way of saying that the vm subsystem is self-contained so no symbols enter it from the left, and there is no shifting out of the right boundary of the processor tree.

overflow as possible is returned to the processor tree by shifting right from the vm subsystem. If there is enough further room in the processor tree to hold the next program, it is also shifted in. To perform this calculation, the io subsystem needs the cr_{io} tuple received from the root of the processing tree, and the number of overflowed contexts and size of the next program held by the vm subsystem.[*] Figure 3.50 shows the resulting preparation for storage management downsweep as seen within the io subsystem

FIGURE 3.50 – Starting the Downsweep of Preparation for SM

```
io.prep_for_sm()
{
        /* pick up capacity and requests from Mago tree */
        capacity = cm_head->msg();
        requests = cm_head->msg();
        available = capacity - requests;

        /* pick up overflow and next program size from VM */
        overflow = vm_head->msg();
        nextsize = vm_head->msg();

        /* calculate a feasible transfer solution */
        if (available <= 0) /* forced overflow */
                transfer = available;
        else  /* we have room for right shifting  */
                if (overflow+nextsize <= available)
                        transfer = overflow+nextsize;
                else
                        if (overflow <= available)
                                transfer = overflow;
                        else transfer = available;

        /* tell the tree and VM about it */
        cm_tail->put(transfer);         /* left entries into processor tree */
        cm_tail->put(0);                /* no right departures from tree */
        vm_tail->put(transfer);         /* right departures from vm */
}
```

[*] This approach avoids total compaction of FFP symbols within the lcell array, and is satisfactory for simulation of single programs. An effective heuristic for handling multiple user programs would be more flexible in the entry of new programs.

The start of the downsweep of preparation for storage management -- described above, and shown in Figure 3.50 -- is clearly a feasible solution, so $P_d(io)$ is true. Thus, by the inductive hypothesis established above, $P_d(1)$ holds as well. Thus the values received at level 0 (by the lcells) are the desired local specifications for storage management; lcells shift bc_0.left_entries in through their left boundaries, and bc_0.right_departures out through their right boundaries. Figure 3.50 showed the io subsystem algorithm. The lcell algorithm for preparation for storage management (followed by the top-level of the ensuing storage management phase) is given in Figure 3.51. The tcell algorithm used in preparation for storage management is given in Figure 3.52.

FIGURE 3.51 -- Lcell Preparation and Storage Management

```
lcell.sm_prep()
{
        /* upsweep */
        /* first the available lcells  -- i.e., capacity */
        cm_tail->put(1);
        /* then the number requested */
        cm_tail->put((user->state==EMPTY)?0:user->fork_n);

        /* downsweep */
        left_entries = cm_head->msg();
        right_departures = cm_head->msg();
}

lcell.storage_management()
{
        while (left_entries<0) { emit(0); left_entries++; }
        while (right_departures>0) { emit(1); right_departures--; }
        load_local();
}
```

FIGURE 3.52 – Tcell Preparation for Storage Management

```
tcell_mgr.prepare_sm_up_dn()
{
        /* upsweep */
        lcm_head->get(&l_cap); rcm_head->get(&r_cap);
        cm_tail->put(l_cap + r_cap);
        lcm_head->get(&l_req); rcm_head->get(&r_req);
        cm_tail->put(l_req + r_req);

        /* downsweep */
        cm_head->get(&l_entries); cm_head->get(&r_departures);
        lcm_tail->put(l_entries); /* left entries for left child */

        if (l_available>=(l_requested+l_entries)) /* no right shift */
                if (r_available>=(r_requested-r_departures))
                        { /* no left shift either */
                        rcm_tail->put(0);   /* right dep left child */
                        lcm_tail->put(0);   /* left ent right child */
                        }
                else { /* no right shift, but must shift left */
                        rcm_tail->put
                                (r_available-(r_requested-r_departures));
                        lcm_tail->put
                                (r_available-(r_requested-r_departures));
                        }
        else { /* must shift right */
                rcm_tail->put(l_requested + l_entries - l_available);
                lcm_tail->put(l_requested + l_entries - l_available);
                }

        /* right departures for right child */
        rcm_tail->put(r_departures);
}
```

3.5. Summary

This completes the discussion of the DOT implementation. The overall strategy and operation of the different phases of the machine cycle was introduced, and the process structuring of the multiprocessor cells was detailed. In addition, the most important algorithms used by these processes to cooperatively implement the phases of the machine cycle were described.

Of these algorithms, partitioning is the most complex in terms of its ClassC representation, and numerous examples were given to illustrate our solution to this difficult design problem. LPL message handling was detailed, and the different types of message packets were presented. Finally, the algorithms for directory creation and preparation for storage management were examined, and shown to be fairly simple. These algorithms require only small amounts of ClassC code for their representations, and correspond directly to the inductive reasoning used to establish their correctness.

CHAPTER 4

Simulating DOT

4.1. Introduction

Chapter 3 presented an implementation model made up of relatively independent, asynchronous cellular processors connected via point-to-point bus links to form a binary tree. The primary role of the DOT representation is to provide a formal and unambiguous description of this model. An important and beneficial result of such a representation is the aid it provides in reasoning about design decisions and the operational characteristics they imply. Expressing ideas in a precise and unambiguous form often highlights errors and points out areas for which concern has been lax or omitted. This has certainly been confirmed by our experience with DOT. Representing DOT thus provided a base for the early stages of an iterative design process.

Since the language used to represent DOT is executable, the above benefits are extended to provide even further assistance to the design process -- during later phases of the design cycle -- through simulation of the implementation, and, in fact, emulation of the complete programming system which it supports.* Architecture emulation, in turn, allows further iterations of the overall design cycle. The LPL architecture is the result of such design iterations.

* This requires the construction of two ancillary programs: *assm*, an assembler for LPL source programs; and *mkusr*, a program to collect FFP user programs into a form appropriate for loading into the vm subsystem.

4.2. The Place of Simulation Within the Design Cycle

Shannon gives the following definition for simulation [Sha75]:

"Simulation is the process of designing a model of a system and conducting experiments with this model for the purpose of either understanding the behavior of the system, or evaluating various strategies for the operation of the system."

Thus, in addition to representing a design, the use of a simulation language can assist understanding and supply a means for evaluation.

Shute describes the role of simulation in the design and study of multiprocessor systems [Shu83]. He identifies major objectives of simulation, and lists the important properties of simulation that can be used in meeting these objectives. Figures 4.1 and 4.2 summarize these important aspects of simulation. Subsidiary benefits of simulation given by Shute are listed in Figure 4.3.

In an initial design, attention to specificity and clarity of expression is paramount. If a simulation model is expressed at the chosen level of detail

FIGURE 4.1 -- Objectives of Simulation

- *Specifying the Operation of the System*
- *Understanding the Operation of the System*
- *Validating the Design*
- *Calculating Performance*
- *Optimizing Performance*

FIGURE 4.2 -- Properties of Simulation

- *Unambiguous Description of the Design*
- *Discipline of Designing in a Rigorous Fashion*
- *Ability to Test the Design*
- *Ability to Emulate the Machine*
- *Predictable, Repeatable Nature of Digital Processing*
- *Ease of Duplicating and Modifying Computer Data*

FIGURE 4.3 – Subsidiary Benefits of Simulation

- *Documentation*
- *Encouragement to try out Alternative Ideas*
- *A Cushion against Production Errors*
- *Provision of a Software Substitute for Hardware*

during this stage, the design can undergo several modifications simply because syntactic errors are reliably captured through the use of a language compiler. Once the simulation is running, insight into the actual working of the design becomes available, and semantic checks may be used to discover operational errors or further confirm the validity of the design.

Given semantically correct operation, the simulation can be used in concert with formal or informal analytic performance models to further iterate on the design process. This phase may involve modifying of the design to achieve performance in accordance with expectations (e.g., errors in pipelining may be discovered in the design), or it may involve modifying the analytic model to more realistically express the restrictions imposed by the design and reflected in its simulation (e.g., perhaps a pipe cannot always be kept full). Finally, a complete and operational system can be emulated, allowing programming and user evaluation, leading to architectural modifications and reentry into the overall design cycle.

The progress of a design performed in this way moves iteratively from the initial design concept through the following stages:

1) unambiguous design representation,
2) valid design,
3) efficient design, and finally
4) overall architectural modification.

4.3. The Cost of Simulation

From the above, it is clear that simulation has a great deal to offer over the hardware oriented approach of "let's build it and see". As processors become more and more complex, and the importance of multiprocessor designs is enhanced by VLSI and wafer-scale integration, it will rapidly become economically unfeasible to approach the design process without the aid of simulation. Simulation aids the design process, making it an accountable and verifiable procedure. What are the disadvantages of simulation? These may be characterized in terms of *cost*.

First, there is the initial cost of the simulation package that is used. Simulation may be performed at a variety of levels: from top-level implementation simulation, as we have done here, to register-transfer-level (low-level implementation) simulation, to circuit-level (realization) simulation. While we use a general purpose language for our simulation, and thus amortize the cost of the compiler product over many users, the specialized concerns of register-transfer-level and circuit-level simulations are of use to a more restricted set of users. Nevertheless, the alternative cost of hardware fabrication may make these specialized packages attractive in price. In addition, the existence of satisfactory simulation languages for each of these levels seems to preclude the necessity for prototyping until a late stage of system development.*

Additionally, there is the cost of writing and debugging the simulation, the cost of using the package to generate results, and the cost of analyzing results.

* Shute mentions the lack of a means of easily moving from one level of simulation to another. This would be desirable from the standpoint of stepwise refinement of design, but would require either a simulation language of extremely wide scope, or automated translation mechanisms. At present, the only feasible approach would seem to be offline development of analytic models for lower level systems to be used within the simulation model of the next higher level.

Although it is reassuring to observe the design of a system actually working, simulation can be very costly in terms of execution time -- especially for extensive lower-level simulations in which large amounts of detail must be explicitly handled. Even for top-level implementation simulations such as we use, massively parallel multiprocessor architectures can be very costly to simulate because of the large number of individual components that are present in the model.*

Validity of design may be established fairly inexpensively, even granting a large number of offline design iterations to eliminate semantic errors. This is because such errors are quickly discovered and at relatively low cost. Optimization of performance, on the other hand, may require more detailed and extensive simulation runs, employing a wide variety of data and runtime configurations.

Shute points out another disadvantage of simulation: the suspicion of users in general, and computer scientists in particular, concerning the correctness of output from a computer. Doubts as to the correctness of simulation results must be answered with a scrupulous concern for the scope of the simulation (thus clearly delimiting the area of applicability of the simulation), and when possible, with an analytic model based on the overall design concept that corroborates the results of the simulation. In our case, use of an analytic model was feasible, and comparison of initial simulation results with predictions of the analytic model pointed out errors in the design that were performance oriented rather than semantic in nature.

* Lower-level implementation or circuit-level simulations for large multiprocessors can be prohibitively expensive. Leung, et. al. [Leu76] have noted this problem in conjunction with packet-switched communication architectures to support dataflow languages, and suggest the use of multiple microprocessor modules to emulate the behavior of groups of system units.

The scope of the DOT simulation is the topic of the remaining portion of this chapter. The analytic model that supports and is in turn supported by the simulation will be described in Chapter 5.

4.4. Introducing Global Time into DOT Operation

Within DOT, there is no a-priori concept of a global time because of the asynchronous nature of the processing cells. How, then, is the ongoing operation of the model during its execution communicated to the designer in useful terms? Building in a knowledge of global time must be approached with care, since this is not part of the implementation and DOT is, first and foremost, an implementation model. It is therefore necessary to create a "meta-level" for the DOT simulation in which the concept of global time is recognized.

4.4.1. Discovery of a Critical Path

One approach might be to seek the critical path within each machine cycle. Each cycle may be considered to begin and end with the calculation of the specification for storage management within the io subsystem. For each such cycle there is a critical path associated with the return of information required for the next storage management calculation. The time associated with the critical path is the cycle time for that particular cycle.

Unfortunately, because of the asynchronous and data-driven character of the machine, this critical path will move with data from process to process, and migrate up and down between cells of the machine. Tracing machine execution in this way to obtain the required information, perhaps through the use of timestamps associated with data, would be a difficult task.

To achieve the desired simulation accuracy, we use a technique based on *discrete event simulation.* Current approaches to discrete event simulation may be divided into two classes: *event-scheduled* and *process-interaction* modeling [Law82]. The approach we use involves a combination of these two common simulation techniques.

4.4.2. Event-Scheduled Modeling

In the event-scheduled approach, a system is modeled by identifying its characteristic events, and routines are written that implement the appropriate state changes associated with each event [Mar80]. A simulation then evolves over time by executing events in increasing order of their time of occurrence. Thus the passage of global time "drives" events, which in turn "drive" the model. Within our model, the highly parallel and asynchronous nature of the components prevents an a-priori knowledge of state changes as a function of events ordered in time, so this approach is not directly applicable.

4.4.3. Process-Interaction Modeling

In the process-interaction approach a system is modeled by explicitly representing the entities that drive state changes (i.e., processes), and providing a mechanism for these entities to communicate the progress of global time (and the enabling of other processes) to a scheduler [Law82, Fra77]. In this case, model processes drive events and explicitly define the passage of global time. This requires a mapping between individual process activities and the progress of global time. Our desire to separate the processes of the implementation model from concern for global time prevents direct use of this approach as well. Since processes of the DOT model are not aware of time, there is no direct way for processes of the DOT model to schedule other processes with respect to a global time.

4.4.4. Process-Interaction with Implicit Events

With appropriate assumptions, an approach based on a mixture of process-interaction and event-scheduled simulation is possible. This approach has been recently used to support simulation of a packet switching network by Aggarwal [Agg82]. In our case, the process-oriented view corresponds well to the way in which the executable model is represented, and event-scheduled simulation provides an attractive alternative to discovery of critical paths.

We therefore incorporate implicit event-scheduling on top of the data-driven process scheduling already present. First, a set of events that may be sequentially ordered in global time must be identified. Within our model, the natural events to focus upon are those involving communication of information between cells of the architecture. This is because, as a first approximation, the primary overhead associated with processing on this machine is the time required for communication between cells.

We define an event as the parallel transfer of information out of all cells that wish to send at that (global) time. The interval between such events is taken to correspond to the time it takes a cell to perform required internal processing after receiving a message, and then send a byte (the basic unit of information exchange in the model) through an intercell communication channel (i.e., a cqueue object).*

The system times available with this approach represent a count of the number of parallel communication transfer events that have occurred since machine initialization. An estimate for the time between events might be τ =

* This simplifying assumption is justified during most of the machine's operation. As shown in Chapter 3, the basic operations of DOT are pipelined. The possible exception lies within the lcells, which may perform expensive operations within message filters, thus slowing the rate of message movement through an active area. Without a more detailed model of the lcell realization, therefore, the assumption of uniform internal processing time between messages is reasonable.

100 nanoseconds.* Times reported in simulations should be considered to be relative to such a multiplicative factor. The resulting level of simulation exactly matches that used by the analytic model, which is also based on counting parallel message transfers. Correspondence between the level of the analytic model and the simulation model is important when verifying the efficiency of the design.

4.5. Implementing Implicit Events

Having defined our events, it remains to guarantee that each process that should participate in an event will do so. This is done by modifying the cqueue class (the only means of communication between cells) to provide scheduling for these events. All processes that wish to send information (during the next event) accumulate on a queue. Only when the model becomes quiescent does the event finally occur, incrementing the global simulation time and allowing accumulated processes to then send their information effectively in parallel.

Implementing the desired simulation facility is straightforward in ClassC. ClassC employs a top-level process scheduling mechanism similar to that used by SIMULA [Fra77], and provides a delay primitive that allows other processes to catch up to a process that delays itself. When all processes have caught up to the delayed process, it is once again eligible for scheduling. The basic difference between process interaction scheduling, as explained above, and our method is that no DOT process ever does an explicit delay within the scope of its representation. With this approach, the DOT process representations are cleanly separated from aspects relating to their accurate simulation in time. Figure 4.4

* Of course, this time is dependent on, among other factors, channel widths and the technology used to realize processing cells and intercell data channels. Aspects pertaining to realization are not within the scope of this dissertation, but this estimate seems reasonable given present VLSI technology.

shows the event scheduler that was added to support simulation, and Figure 4.5 shows the DOT cqueue qtail.put entry after its modification to provide implicit events for communication.

FIGURE 4.4 -- The DOT Event Scheduler

```
/*
This task interfaces with the classc delay mechanism in order to
implement the clock tick events that the qtail.put mechanism uses.
(In addition, a message is printed every thousand ticks to aid in
recognition of deadlock situations.)
*/

events.new(event)
class object *event;
{
        cycle {
                delay(1);
                if ((clock%1000) == 0) printf("events: clock=%d",clock);
                event->alert();
                }
}
```

FIGURE 4.5 – Qtail.put Implements Implicit Events

```
/*
The entry used by a process that sends data between cells.
*/

qtail.put(m)
char m;
{
        event->remember(thistask);          /* pay the price in time */
        thistask->sleep();
        event->forget(thistask);
        putfree(m);                         /* then transfer message */
}

void qtail.putfree(m)
char m;
{
        message->mess = m;                  /* put message into object */
        (message->sig)->remember(thistask);/* remember this task */
        x_tail->put((class object *)message); /* initiate transfer */
        thistask->sleep();                  /* sleep until receipt */
        (message->sig)->forget(thistask);
}
```

4.6. Simulation Output

The complete DOT model representation includes 25 classes. These include the process classes (or tasks) discussed in Chapter 3, and the above event handler. When the model is executed, a tree height parameter is given, and the required number of these classes are instantiated and connected to form a machine of the desired size. As the processing cells of the resulting DOT machine come alive, operation begins with partitioning of the machine in response to the (initially empty) lcells. The following preparation for storage management then detects the available lcell array capacity and the vm subsystem shifts in FFP program text as appropriate during the ensuing storage management phase. Execution then continues with partitioning and successive machine cycles. DOT models for machines containing hundreds of lcells may be created and observed within computationally reasonable time periods.

4.6.1. Tracing Machine Operation

The ClassC runtime support package maintains a globally available variable called *clock* that contains the current simulation time, and each process within the DOT model has a unique process id maintained by the ClassC task scheduler (known locally by each process as *thistask*). As the DOT design progressed, trace statements for the processes were written to allow them to record their progress and signal the simulation times of important events. These statements serve the dual function of supporting debugging of the DOT machine's execution, and because of this fact they are pervasive throughout the entire ClassC model representation.

As a result of this approach towards tracing operation of the machine, it is possible to literally pick apart the detailed operation of the machine from whatever vantage point is desired. A wide range of precision is available. Conditional assembly of trace statements allows selection of the desired information from the huge mass of detail potentially available during the execution of a large model instantiation.

This highlights an important aspect of simulation: arbitrary precision (within the limit of the time grain used) may be employed to zero in on design errors once a problem is detected. Digital simulation is entirely repeatable, so repeated runs at ever finer levels of detail are possible with reproducible results. This flexibility would not be available in a hardware prototype, and this points out the importance of carrying an initial simulation approach as far as possible.

We now give examples of the information available from process tracing. The trace for these examples is at a fairly high level, and brings together information concerning the overall machine cycle and its three phases. Two listings are given. Both depict process activity during execution of the example FFP program given in Figure 4.8. The first trace listing is filtered, and provides brevity by omitting all but the first of a succession of messages that refer to the same thing. As seen from the second listing, which only displays a portion of cycle 1, such filtering is quite effective while retaining useful information. For each trace message, the involved process identifies itself by giving its class type (e.g., lcell) and a unique process id (the value of ***thistask*** maintained by the ClassC process scheduler), and then provides informative data, such as the beginning and ending times of an activity (displayed in brackets: [begin-end]), or the present simulation time.

FIGURE 4.6 – Filtered High-Level Trace Output

```
machine.new: starting up with clock=0
lcell(372168): partitioning[0-10] duration 10
io.prep_sm: sm_grant, sending stop message for cycle 0, clock=16
lcell(427704).sm:(4:4) [37-52] duration 15
io_input: got pgm request = 8, clock=105
lcell(466988): partitioning[37-107] duration 70
lcellLinp(169080): starting interpreter. clock=139
lcellLmsg(254084): user's message has returned. clock=242
io.prep_sm: sm_grant, sending stop message for cycle 1, clock=248
lcell(126992).sm:(40:0) [276-276] duration 0
io_input: got pgm request = 12, clock=302
lcell(471952): partitioning[276-303] duration 27
lcellLinp(374972): starting interpreter. clock=335
lcellLmsg(359284): user's message has returned. clock=381
io.prep_sm: sm_grant, sending stop message for cycle 2, clock=389
lcell(126992).sm:(40:0) [413-413] duration 0
io_input: got pgm request = 4, clock=427
lcell(466988): partitioning[413-429] duration 16
lcellLinp(129796): starting interpreter. clock=461
lcellLmsg(398568): user's message has returned. clock=507
io.prep_sm: sm_grant, sending stop message for cycle 3, clock=515
 -- Machine Empty -- halting execution
```

FIGURE 4.7 – Unfiltered High-Level Trace Output

```
          ...
          ...
lcell(427704).sm:(4:4) [37-52] duration 15
lcell(377132).sm:(2:4) [37-55] duration 18
lcell(372168).sm:(60:3) [37-58] duration 21
lcell(337848).sm:(3:4) [37-61] duration 24
lcell(332884).sm:(1:4) [37-64] duration 27
lcell(266060).sm:(60:3) [37-67] duration 30
lcell(261096).sm:(60:2) [37-70] duration 33
lcell(226776).sm:(12:3) [37-73] duration 36
lcell(221812).sm:(8:3) [37-76] duration 39
lcell(171240).sm:(60:2) [37-79] duration 42
lcell(166276).sm:(40:1) [37-82] duration 45
lcell(131956).sm:(4:1) [37-85] duration 48
lcell(126992).sm:(40:0) [37-88] duration 51
lcell(126992): partitioning[88-98] duration 10
lcell(131956): partitioning[85-98] duration 13
io_input: got pgm request = 8, clock=105
lcell(466988): partitioning[37-107] duration 70
lcell(471952): partitioning[37-107] duration 70
lcell(432668): partitioning[37-107] duration 70
lcell(166276): partitioning[82-108] duration 26
lcell(171240): partitioning[79-108] duration 29
lcell(372168): partitioning[58-109] duration 51
lcell(261096): partitioning[70-109] duration 39
lcell(332884): partitioning[64-109] duration 45
lcell(221812): partitioning[76-109] duration 33
lcell(427704): partitioning[52-109] duration 57
lcell(377132): partitioning[55-109] duration 54
lcell(266060): partitioning[67-109] duration 42
lcell(337848): partitioning[61-109] duration 48
lcell(226776): partitioning[73-109] duration 36
lcell(427704).build_directory: [109-123] duration 14
lcell(166276).build_directory: [108-124] duration 16
lcell(171240).build_directory: [108-124] duration 16
lcell(372168).build_directory: [109-125] duration 16
lcell(261096).build_directory: [109-125] duration 16
lcell(332884).build_directory: [109-125] duration 16
lcell(221812).build_directory: [109-125] duration 16
lcell(377132).build_directory: [109-125] duration 16
lcell(266060).build_directory: [109-125] duration 16
lcell(337848).build_directory: [109-125] duration 16
lcell(226776).build_directory: [109-125] duration 16
lcell_inp(169080): starting interpreter. clock=139
lcell_inp(174044): starting interpreter. clock=154
lcell_inp(224616): starting interpreter. clock=163
          ...
          ...
```

4.6.2. Lcell Array Snapshots

In addition to the above trace facility, which provides global information ordered in time, the output channels and processes of the DOT model are used to present snapshots of the FFP- and LPL-level symbol representations located within the lcells. As described in Chapter 3, output from the lcell array is piped out of the tree and into the io subsystem in left-to-right textual order. This information is therefore ordered in space as well as in time. There is one snapshot per cycle, and it records the situation within each (non-empty) lcell at the end of the execution phase -- after detection of completed RAs has been performed, so results of completed applications are available.

The io subsystem output process presently sends these results to a terminal or a file for later examination. Empty lcells do not appear, and cells with symbols in them are listed in left-to-right order. Column headings provided with the output designate the user program id, lcell symbol, lcell state (0=ground, 1=executing, 2=completed), fork_id, aln, rln, symbol_index, and the directory 4-tuple. Columns to the right of the arrow indicate the result of stepping a completed reduction forward.

FIGURE 4.8 – FFP Expression for Reduction

(+ (< apply-to-all * > < < 1 3 > < 2 4 > >))

An example of the lcell array snapshots is now given. To aid understanding, comments have been placed to the right of the snapshot output. The model for this example was created with a height of four, so there were 16 lcells available for holding the program. Prior to execution of the model to generate the following snapshots, the FFP-level text representation for the expression shown in Figure 4.8 was loaded into the vm subsystem using the *mkusr* program. As indicated by the trace output of Figure 4.6, which corresponds to the following execution snapshots, the total time required to execute this FFP expression (including load time) is ~500τ, or, if τ=100 nanoseconds, ~50 μsecs.

```
PGM SYMB S FID ALN RLN NDX-DIR --> NSYM NALN     -- End of Cycle 1
001  (   0 001 000 000 000 0000           app sym not innermost
001 #004 0 001 001 000 000 0000           4 is n-ary add op-code
001  (   1 001 001 000 000 0000           app sym innermost so RA
001  <   1 001 002 001 001 1000           and state = executing
001 #008 1 001 003 002 002 1100           8 is apply-to-all op-code
001 #012 1 001 003 002 003 1200           12 is multiply op-code
001  <   1 001 002 001 004 2000
001  <   1 001 003 002 005 2100
001 #001 1 001 004 003 006 2110
001 #003 1 001 004 003 007 2120
001  <   1 001 003 002 008 2200           This sym forks to receive
001 #002 1 001 004 003 009 2210           copy of operator (mult)
001 #004 1 001 004 003 010 2220           as required by apply-to-all
```

```
PGM SYMB S FID ALN RLN NDX-DIR --> NSYM NALN     -- End of Cycle 2
001  (   0 001 000 000 000 0000
001 #004 0 001 001 000 000 0000
001  (   2 001 001 000 000 0000    <    001    reduction complete
001  <   2 001 002 001 001 1000    (    002    so stepped forward.
001 #008 2 001 003 002 002 1100
001 #012 2 001 003 002 003 1200   #012  003    result is sequence
001  <   2 001 002 001 004 2000                of multiplications
001  <   2 001 003 002 005 2100    <    003
001 #001 2 001 004 003 006 2110   #001  004
001 #003 2 001 004 003 007 2120   #003  004
001  <   2 001 003 002 008 2200    (    002    the fork_id tells
001  <   2 002 003 002 008 2200   #012  003    how to place
001  <   2 003 003 002 008 2200    <    003    these symbols.
001 #002 2 001 004 003 009 2210   #002  004
001 #004 2 001 004 003 010 2220   #004  004
```

```
PGM SYMB S FID ALN RLN NDX-DIR --> NSYM NALN    -- End of Cycle 3
001  (   0 001 000 000 000 0000
001 #004 0 001 001 000 000 0000
001  <   0 001 001 000 000 0000
001  (   2 001 002 000 000 0000     #003  002
001 #012 2 001 003 001 001 1000          both multiplications
001  <   2 001 003 001 002 2000          complete in one cycle,
001 #001 2 001 004 002 003 2100          and are stepped forward.
001 #003 2 001 004 002 004 2200
001  (   2 001 002 000 000 0000     #008  002
001 #012 2 001 003 001 001 1000
001  <   2 001 003 001 002 2000
001 #002 2 001 004 002 003 2100
001 #004 2 001 004 002 004 2200
```

```
PGM SYMB S FID ALN RLN NDX-DIR --> NSYM NALN    -- End of Cycle 4
001  (   2 001 000 000 000 0000     #011  000    Add now innermost
001 #004 2 001 001 001 001 1000                  and completes in
001  <   2 001 001 001 002 2000                  one cycle. 11 is
001 #003 2 001 002 002 003 2100                  the answer.
001 #008 2 001 002 002 004 2200
```

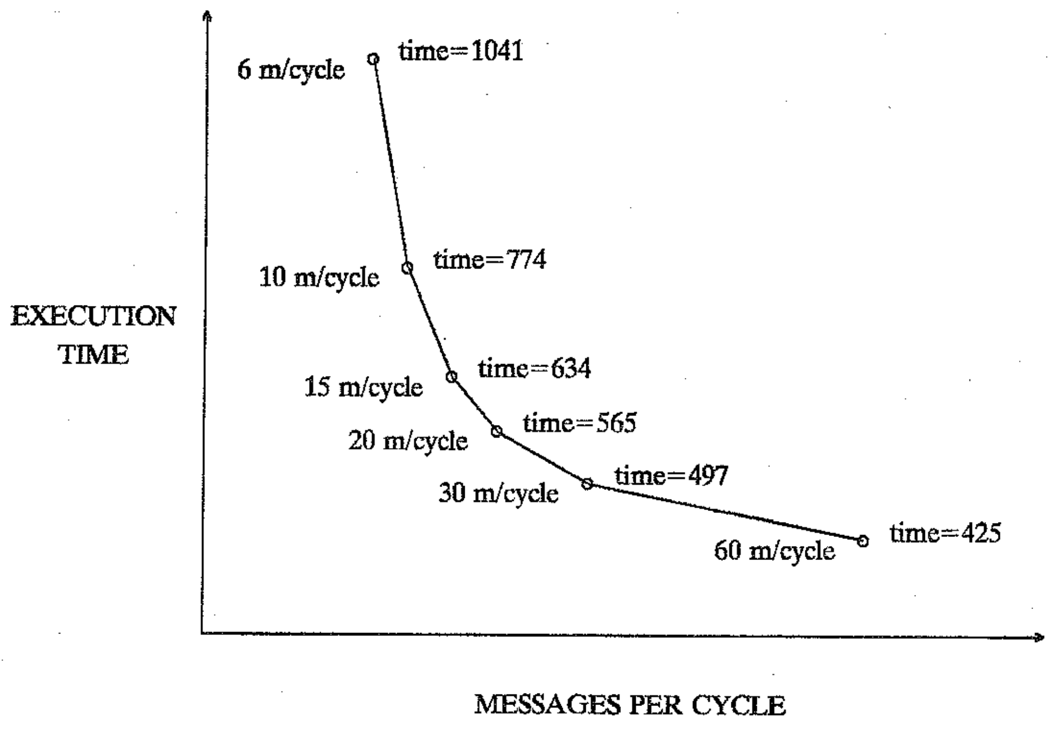
4.7. Simulation Results

The most useful result of the simulation is the way it has aided our understanding of the operation of the implementation model. Nevertheless, other useful results have been obtained. Raw performance figures such as those provided by process traces are invaluable for the synergistic development of an analytic model. Our simulation results for a variety of LPL programs will be displayed in tabular form in the following chapter -- in conjunction with the predictions of the analytic model. This will allow easy comparison of simulation results with analytic model predictions.

Additionally, an initial simulation study designed to assist development of LPL programs tailored for large operands has been performed. Recall that LPL programs exercise a degree of control over when storage management takes place. Therefore, an LPL program may be written so that its execution on large operands does not seriously affect the progress of other reductions in the machine. To do this, an LPL program can keep a count of messages, and execute an *smanage* instruction after every *m* messages. The question, of course, is how to choose *m*. Ideally this value should be large enough to allow a reasonable amount of work to be done, and small enough that other LPL programs performing reductions are not unduly delayed. Figure 4.9 shows the simulation results obtained from studying the behavior of the FFP SORT operator (whose LPL definition was given in Section 2.4.1.11) for different values of *m*.

This study was performed on a DOT machine of 64 lcells, and presents the results obtained for sorting 60 numbers. As can be seen, the minimum time of

FIGURE 4.9 – Sort Time v.s. Number of Messages per Cycle



425, achieved by sending the 60 messages during one cycle, is quite close to the time of 497, achieved when two cycles are used (30 messages per cycle). The optimum execution time is approached asymptotically, so backing off and being "fair" to other programs in the machine is not as costly as might be expected.

CHAPTER 5

Analytic Performance Model

This chapter represents a summation of the DOT design and its behavior. Our concern here will be to reason about the operation of the DOT machine, and ultimately to predict execution times for FFP language programs.

In the previous chapter, we listed virtues of simulation in multiprocessor design, and discussed the approach taken for simulation of DOT. Among benefits identified was the unambiguous description forced upon the designer by the use of a compilable language -- enabling criticism and preventing obfuscation.

An analytic model (when one is available) performs a similar function with respect to performance. It brings into clear focus the ultimate result of design decisions by representing their *intention* with respect to system behavior. It is only after a simulation (or prototype) is running that the satisfactory implementation of these intentions may actually be verified, however. In our experience, a highly dynamic interplay between modifications to the design, and modifications to the analytic model then results.

The analytic model we present began as a set of assumptions concerning the desired operational characteristics of a projected implementation (as suggested originally by Mago [Mag79]), and ultimately matured to reflect the actual operation of the DOT implementation. Chapters 3 and 4 presented this implementation. We now analyze its performance.

We begin by examining the execution cycle in more detail. Upper and lower bounds for the three phases of the machine cycle are presented and related to execution times for RAs. The method developed by Koster [Kos77], Stanat and

Williams [Sta81], and Mago et. al. [Mag83], may then be used to derive data-dependent upper and lower bounds for many FFP language programs. Examples are given, and correspondence with the results of simulation is verified.*

5.1. The Execution Cycle

When measuring the duration of some activity that takes place over time, *events* define the points at which a measurement may be made. In the case of cyclic behavior, a natural event should be the beginning of the cycle. What definite and unambiguous events exist during DOT operation? There are really only two that may be recognized within a machine-wide context: receipt by the io subsystem of the sm_grant message, and receipt by the io subsystem of the preparation for storage management upsweep. Neither of these is particularly valuable from the standpoint of measuring execution times. LPL programs execute within lcells, so we prefer to base our analytic model on events that occur in lcells. Luckily, with appropriate assumptions, another event more useful for this purpose may be identified. This is the beginning of storage management in the lcells.

Although it is convenient to think of *the* (i.e., single) execution cycle for DOT as something descriptive of its overall behavior in time, Chapter 3 showed that the DOT machine is really a highly dynamic and reconfigurable collection of fine-grain cellular processors. Individual processors do synchronize, or come together locally, to exchange information in support of storage management, partitioning, and execution, but it is possible for different cells of DOT to be performing in all three of these different phases simultaneously. Each cell of the DOT implementation goes through the execution cycle we have described,

* Section 5.1 provides information appropriate for a casual reader, while Sections 5.2 - 5.6 contain the details of the analytic model. Although these sections may be skipped if desired, the summary in Section 5.7 should be read.

but only according to its local needs.

From observing the behavior of the simulation model, we note that storage management begins at the same time in all the lcells. This is because the specification for storage management is computed in a parallel downsweep through the tree starting at the io subsystem, where the top-level transfer function is computed and sent down. The data path length is the same from the io subsystem to each lcell, and exactly the same operations are performed by each tcell along the way.

The above observation is relative to our assumption within the simulation that the the data paths are identical in their transmission characteristics. In a hardware realization, this would not be true, of course. Thus, we confine our reasoning to the performance of the simulation, and assume that differences between the simulation and an actual realization are negligible. With this done, we may also speak of the beginning of storage management within the lcells as a definite event.

Although storage management begins at the same time in the lcells, it does not finish at the same time. Storage management is pipelined, with programs and previous overflow being entered from the left. Programs toward the right of the lcell array may not require movement at all (unless they are forking), and thus may complete storage management in no time. In general, whenever information is shifted in the lcell array, text located at the end of shift movement will complete first, with its neighbor finishing next, and so on, down a chain of consecutively shifted symbols, ending at the source of the shift movement. Thus, even symbols of the same RA may not complete storage management at the same time.

Immediately following completion of storage management within an individual lcell, the lcell initiates partitioning. Clearly, the completion of partitioning can occur at widely varying times for different RAs. In addition, partitioning completes at different times within the lcells of a given RA -- not because partitioning begins at different times over the lcells of the machine (after all, a pruning downsweep within a given active area begins in a single tcell much as the preparation for storage management downsweep), but because area trees are not height balanced with respect to communication delays. As shown in Chapter 3, circuit switched area channel connections may bypass tcell area nodes.

Execution thus begins at different times within different active lcells. The io subsystem detects the sm_grant message at some later time, while active lcells continue execution of their individual LPL program segments, and the stop message is sent down to the lcells. This reaches all lcells at the same time, but clearing out an area (to guarantee that all area messages have been received) may require time dependent on its height.* Finally, the upsweep of the preparation for storage management reaches the io subsystem, and the value of the top-level storage management transfer function is computed and sent down into the tree.

As far as the overall DOT machine is concerned, it would make sense to measure times relative to the arrival of the upsweep of preparation for storage management (or equivalently, calculation of the topmost storage management transfer function) within the io subsystem. Our primary concern here is with

* Normally, a top of area node manager is able to detect that the stop message has passed through its tcell and avoid this dependency. If, however, the stop message comes through while the node manager is waiting for an LPL message, a stop packet may have to rise from the lcell level in response to the stop message in order to notify the top of area manager to clear out the channels.

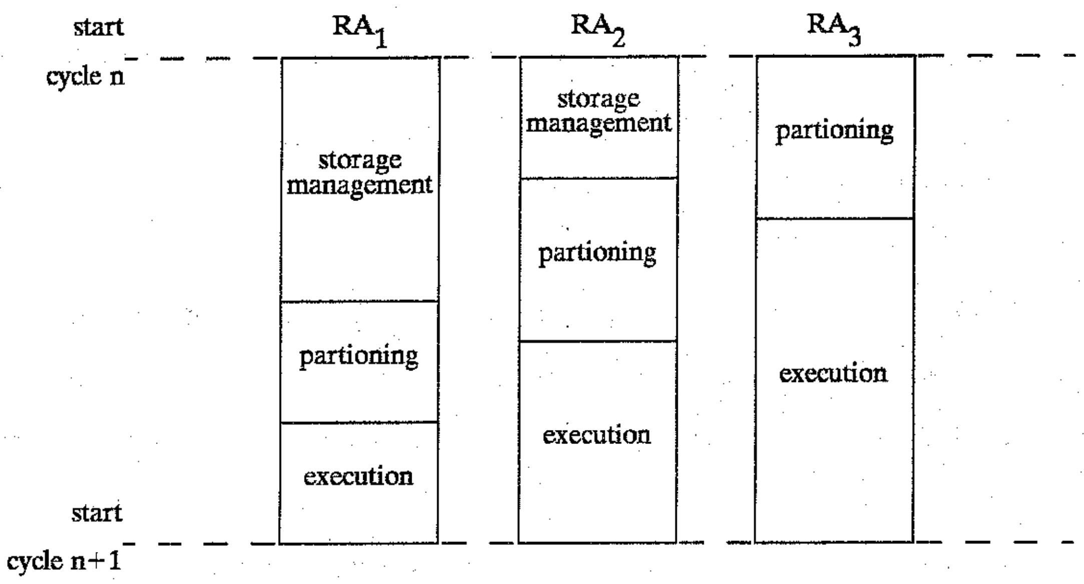
RAs, however, for which a logical starting point of execution is the beginning of the partitioning phase in which they are detected and allowed to come into existence. Unfortunately, the start of partitioning is not a machine-wide (or even RA-wide) event, and our ultimate aim of predicting execution times for complete programs is best served by choosing a beginning for the machine cycle that is common for all RAs of the machine.

From the above discussion, it is clear that our first concern in developing an analytic model is to define exactly what we should measure. With the ultimate aim of predicting the behavior of complete FFP programs, our analysis is initially oriented around predicting execution times for an individual, distinguished RA, in the possible presence of others within the lcell array. Since, from the above summary of DOT operation, we know that storage management begins at the same time for all lcells within an RA, and for all RAs within the machine, this time will be treated as the true *beginning* (and *end)* of the machine cycle.

This will allow us to use the results of analyzing individual RAs to predict execution times for complete programs containing multiple RAs during each machine cycle. While dictated by the practical concerns of the analytic model, this approach is nonetheless reasonable; prior to detection of RAs during the phase we have called partitioning, their FFP-level text representations must first be created, and this is done during storage management.

Figure 5.1 shows a diagram of the machine cycle, and depicts the progress of individual RAs through the three phases. Note that the only machine-wide event recognized is the beginning of the cycle, with progress through the phases being a phenomenon local to individual RAs. RA_3, for example, requires no storage management, so begins partitioning immediately.

FIGURE 5.1 – RA Progress Through a Single Machine Cycle



5.2. Notation

We derive upper and lower bounds for the duration of the machine cycle phases in an RA. These are denoted by UB(phase), and LB(phase). The particular RA of interest to the analysis is designated as *the* RA, and the term "the lcells" refers to the lcells of this distinguished RA. The variable n is used to represent the count of the lcells, and h is used to represent the height of the RA. The height of the largest new RA within the machine (used in analysis of partitioning phase for a new RA) is represented by h'. The number of lcells within the largest new RA is represented by n'. The variable N represents the total number of lcells within the machine, and H is used to represent the total height of the machine (counting the io subsystem). Angle brackets, when used to enclose the name of an FFP operator, denote the size (in bytes) of the corresponding LPL program.

During storage management, user contexts are shifted on the lateral lcell channels. We therefore take the term *user context* to be synonymous with information that is shifted during storage management, and U denotes the size of this information -- irrespective of whether the context is executing (in which case U is large, since an LPL code segment and the LPL environment are included), or the context is not executing (in which case U is small, since only the FFP-level representation is required).

The size of an executing user context is primarily dependent on the size of the LPL program segment. For all examples shown in this dissertation, a code area of 150 bytes suffices. This, plus LPL environment registers (presently requiring 54 bytes of storage), results in a user context size of U = 204 bytes within active areas. A non-active user context requires U = 4 bytes.

As for the simulation, we assume a uniform τ throughout the machine, and predictions of the analytic model are implicitly in these units. In addition, we assume that no new programs enter the machine during the period of time covered by the analysis.

5.2.1. Area Heights

The height parameters H and h given above are related to the size of the underlying lcell segment. For the overall tree, $H = \log_2(N)+1$, where "1" counts the io subsystem. For an active area, we define h as the maximum number of nodes a message may pass through on its way to the top of area.* This definition results from the use of circuit-switched area channels, and assumes that processing and communication delays inherent in sending and receiving processing cells are the predominant source of transmission delays. As shown in

* Note that we only count area nodes (i.e., places where area channels come together and processing is required) -- not tcells through which an area channel is circuit-switched.

Figure 5.2, h can vary from $\lceil \log_2(n) \rceil$ to n-1, depending on the distribution of RA contexts among the available lcells.* Since log(n) is the height of a balanced binary tree with n leaves, log(n) is clearly a lower bound for h. This value will be used in lower bound formulas. Since h<H, a least upper bound for h is min(n,H)-1. This value will be used in upper bound formulas.

FIGURE 5.2 -- Trees for Lower and Upper Area Height Extremes

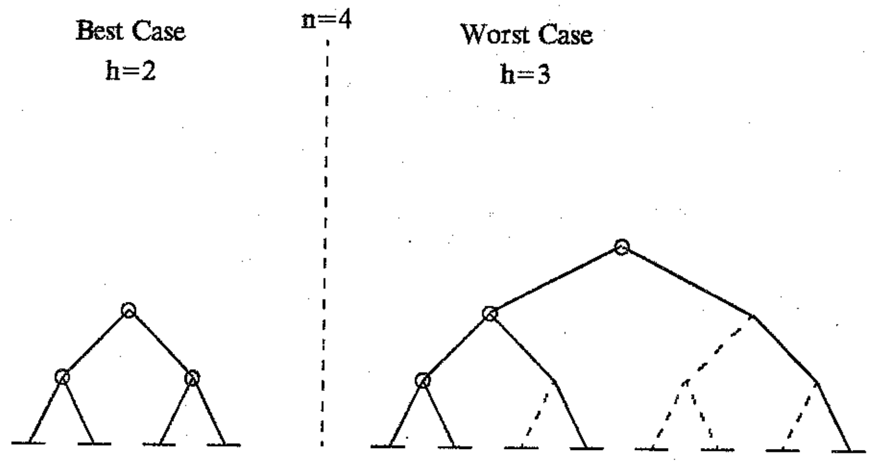


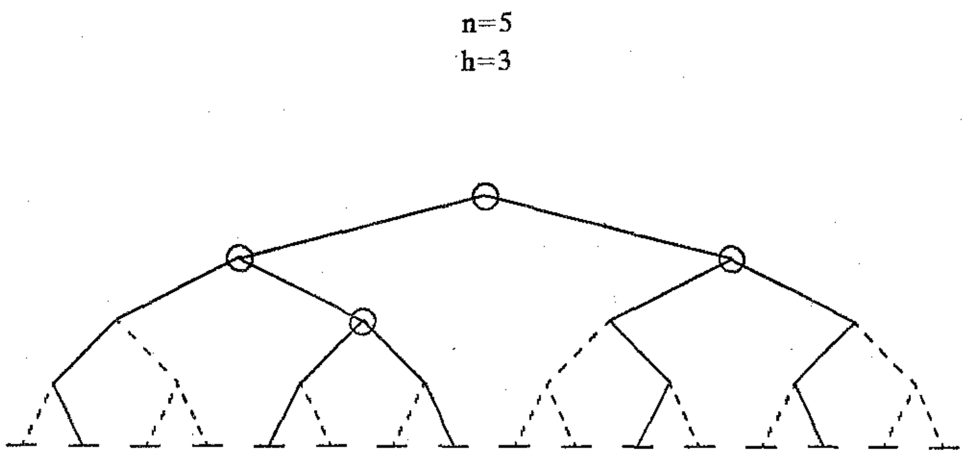
FIGURE 5.3 -- Bounds on Area Height (using H=21)

n	*LB=log(n)*	*UB=min(n,H)-1*
2	*1*	*1*
4	*2*	*3*
8	*3*	*7*
16	*4*	*15*
32	*5*	*20*
64	*6*	*20*
128	*7*	*20*
---	---	*20*

* In the interest of notational brevity within formulas, log(n) will subsequently be understood to denote the integer ceiling of the base 2 logarithm of n. Using this approach, log(3)=2.

As shown in Figure 5.3 (which assumes a machine with a million lcells), the difference between the lower and upper bounds on area height is only important as n grows large, and even for a machine of this size, H is small enough that lower and upper bounds don't diverge greatly. In addition, as shown by the example in Figure 5.4, uniform grouping of RA symbols within area subtrees results in log(n) being a good approximation for h, and the presence of interspersed empty lcells makes no difference to this result. This is because circuit-switched area channels completely avoid portions of the tree containing contiguous groups of empty lcells.*

FIGURE 5.4 -- Uniform Loading in the Presence of Empty Lcells



* An interesting point is that, if desired, we could ***guarantee*** uniform loading by modifying the calculation of the storage management transfer function. This will be discussed in Chapter 6.

5.2.2. Phases of RA Progress

The phases through which an RA progresses form the basis for prediction of its execution time. Each phase is now defined in a way appropriate to a clean division of concerns in the analytic model. While this represents a review of DOT operation as described in Chapter 3, it is important to clarify the exact nature of each phase within the context of the analytic model.*

5.2.2.1. Storage Management Phase – RA Creation

The storage management phase within an lcell involves the actual movement of user contexts within the lcell array, and nothing else. (The time required for calculation of the specification for storage management is included in the execution phase.) Storage management starts in an lcell upon receipt of a specification for storage management, and ends upon completion of the shifting necessary to satisfy the specification.

5.2.2.2. Partitioning Phase – RA Detection

The partitioning phase for an RA includes creation of its embedded tree of processing cells, and all other activities required prior to actually beginning execution within its lcells. Thus, there are two types of partitioning phase: one type associated with *new* RAs (for which the preparatory activities include directory creation, and loading LPL code segments), and the other associated with *old* RAs (which already have their directories and LPL code segments).

5.2.2.3. Execution Phase – RA Execution

This phase starts in an lcell with the beginning of actual LPL program execution. Although execution starts at different times within different lcells of

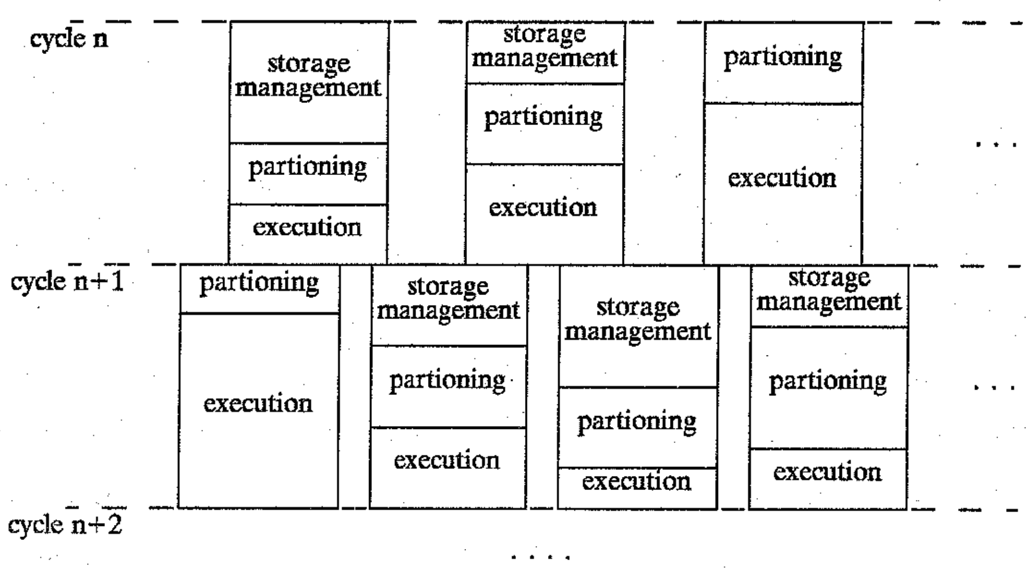
* Chapter 3 gave an informal description of the overall machine cycle. Here we are concerned with an exact analysis of events within the active lcells of an RA.

an RA (because partitioning completes at different times), the lcell that begins execution last is on a critical path for progress of this execution. If LPL messages are sent, this lcell delays turnaround of the first message wave. Thus, for all practical purposes, we may consider the execution phase to begin when *all* the lcells of the RA have been prepared for execution. In order for the execution phase to end, all lcells must send an sm_grant message to the io subsystem. Following this, a stop message arrives at the lcells, area channels are cleared out, and the specification for storage management is computed. The execution phase for an RA ends when the specification for storage management reaches the lcells in which it is contained.

5.3. Formulas for the Duration of RA Phases

We now examine the duration of each phase of RA progress during successive machine cycles. Figure 5.1 showed an example of this progress for a single cycle. The general multi-cycle situation is depicted in Figure 5.5. As shown, the duration of each cycle may be considered separately from the last, with new RAs coming into existence, and old RAs being reborn (if necessary) for each new cycle.

FIGURE 5.5 – RA Progress Through Successive Machine Cycles



5.3.1. RA Storage Management – (SM)

During storage management, lcells shift user contexts in a pipelined fashion. For purposes of simplicity, we assume that all lcells of the RA complete storage management at the same time. Although this is not strictly so, it makes no difference to the ultimate results since the lcell taking the longest time is on a critical path for subsequent execution within the RA. The last lcell to complete storage management within the RA allows subsequent execution, and the storage management time corresponding to this is used by the analytic model.

Every LPL context in an RA begins its execution phase with a *forkn* value of one, and only LPL forking operations can change this value.* The LPL *forkn* context values found within the lcells that create an RA during storage management therefore determine a minimum duration for the storage

* As described in Section 3.2.3.2, the forksize argument of a **fork** statement is loaded into this register when the statement is executed.

management phase of the RA. When storage management takes place, the ***forkn*** variable is used to indicate the number of LPL contexts to be spawned from the containing lcell.

A lower bound for the duration of storage management for the RA is $\max_{RA}(U_i * \mathit{forkn}_i - 1)$, where the maximum is to be taken over all the lcells that create the RA, and U_i is the size of the user context to be located within the i^{th} lcell of the RA.* This is equivalent to assuming that there are enough empty lcells neighboring any given forking context to support its storage requirements. An upper bound for the duration of storage management is arrived at by examining the situation for multiple RAs, and assuming total compaction within the lcell array with the RA at one end. The context in the lcell at the end of the RA must then be shifted through $\Sigma_{\text{machine-lcells}}(\mathit{forkn}_i - 1)$ lcells, where the sum is to be taken over all lcells of the machine. The resulting lower and upper bounds for the duration of storage management for the RA is given in Figure 5.6.

FIGURE 5.6 – Duration of Storage Management for RA

$$LB(SM) = \max_{RA\text{-}lcells}(U_i * forkn_i - 1),\ 1 \le i \le n$$
$$UB(SM) = \Sigma_{machine\text{-}lcells}(U_i * forkn_i - 1),\ 1 \le i \le N$$

5.3.2. RA Partitioning

There are two cases to consider for partitioning. In the first case, the RA has already been executing (during the preceding cycle), so its lcells are ready to continue execution as soon as an active area is constructed for them within the tcells. In the second case, the RA is new and requires a directory and LPL code as well as creation of an active area.

* Recall that for our purposes, this duration is measured in units of τ.

5.3.2.1. Partitioning Old RA – (PO)

Here just creation of the active area is required. This process is pipelined from the lcells to the height required to determine whether the application is innermost, and four values must be unloaded at the top and bottom of the pipe. This gives a lower bound of 2h+8 (assuming that pruning begins at the top of area). During the initial partitioning upsweep, certain partitioning configurations result in delays of more than 1 in nodes of the pipeline. This delay can be up to 3, giving an upper bound of 4H+8 (assuming pruning begins in the io subsystem).* Using the bounds on h (the area height) given in Section 5.2.1, Figure 5.7 gives lower and upper bounds for partitioning an old RA.

FIGURE 5.7 – Duration for Partitioning Old RA

LB(PO)	=	$2*\log(n) + 8$
UB(PO)	=	$4H + 8$

5.3.2.2. Partitioning New RA – (PN)

Here, we must consider building the directory and loading LPL programs as well as the initial creation of an active area. The directory is built using area channels, and is pipelined with four values unloaded at the top and bottom of the pipe. This process therefore always takes 2h + 8. When operator definitions are sent to the lcells, they originate in the io subsystem and are broadcast downward to the lcells where they wait until the directories for all new RAs are ready.** For a lower bound, we assume the LPL program for the RA is first in

* The additional delay is related to the ordering of information that is sent up during partitioning. Treating this aspect of partitioning requires a more detailed analysis than is appropriate here.

** Old RAs require no code segments, so arriving code segments only wait on the time required to compute the directory of the largest new RA. The reason for waiting on the largest new RA to complete building its directory is that the LPL code segments are the stream of operator definitions which must be delivered to the lcells. The lower bound for input time is then the length of the operator definition, denoted by <op>. For an upper bound, assume the operator the definition is sent last. Then, the input time is $\Sigma_{\text{new-RAs}}$<op>, with the sum taken over all new RAs.*

In order to produce useful lower and upper bounds for PN, h', the height of the largest new RA can be treated as follows: for a lower bound, assume h'=h (i.e., n'=n), and for an upper bound, use h'=H-1. Although these estimates yield bounds which are less tight than possible (assuming complete knowledge of the lcell array) they depend only on the RA of interest and are therefore easier to use. With this approach, and use of the above result for PO, we get lower and upper bounds as summarized in Figure 5.8.

FIGURE 5.8 -- Duration for Partitioning New RA

(PN)	*LB*	*UB*
build area	$2*\log(n) + 8$	$4H + 8$
build directory	$2*\log(n) + 8$	$2*(H-1) + 8$
load code	<op>	Σ<op>
total	*LB(PN)*	*UB(PN)*

$$LB(PN) = 4*\log(n) + \langle op\rangle + 16$$
$$UB(PN) = 6H + \Sigma_{new\text{-}RAs}\langle op\rangle + 14$$

5.3.3. RA Execution -- (EX)

The execution phase for the RA includes the time until sm_grant is sent up, the time until the stop message is received (which may include further LPL

broadcast only once, and the lcells of all new RAs must be able to filter code when this is done.

* Actually, when multiple RAs require the same operator definition, the required LPL program is only sent in once, so duplicates need not be counted.

execution), and the time required to compute the specification for storage management.

Let t_s denote the duration of the execution phase until the sm_grant is sent up by the lcells of the RA. The only contributors to t_s are LPL message operations. For each message wave, a prefix packet containing appropriate handling instructions must be sent to the area nodes. The prefix packet contains three bytes, so sending this information out of an lcell accounts for an initial delay of three. The prefix packet is pipelined and followed immediately with the key and message data, which are also pipelined. It therefore takes time h+3 for the first message to reach the top of area, and time h to return to the lcells (since the prefix packet is not returned). For each message that returns to the lcells during a particular message wave, there is an unload time of (3+msize).[*] This gives

$$t_s = \Sigma_{\text{mwaves}}(2h + 3 + ((\text{mreturn\_cnt}_i)*(\text{msize}_i+3))),$$

where for message wave i, mreturn_cnt_i is the number of messages that return to the lcells on that wave, and msize_i is the value coded in the corresponding LPL **send** statement. Using the bounds on area height presented in Section 5.2.1 gives:

FIGURE 5.9 -- Duration of Time to SM_Grant, t_s

$$LB(t_s) = \Sigma_{mwaves}(2*log(n) + 3 + ((mreturn\_cnt_i)*(msize_i+3)))$$
$$UB(t_s) = \Sigma_{mwaves}(2*min(n,H) + 2 + ((mreturn\_cnt_i)*(msize_i+3)))$$

In addition to t_s, there is the stopping interval from t_s until preparation for storage management begins. During this interval, the sm_grant goes up the

[*] The 3, here, represents the byte-count and the two key values that are sent with every message.

tree, and a stop message comes down. Assuming that the RA is executing in the absence of other RAs (i.e., all other lcells have allowed storage management), this takes time 2H. In addition, upon arrival of the stop message at the lcells, a stop packet (as described in Section 3.4.2.1) is sent up and then down the area channels to guarantee they are cleared out. Thus, if the top of area node doesn't see the stop message on its way down the cell manager channels (this can happen in the absence of LPL message activity), it can take an extra 2h+3 time units to guarantee the end of message activity (the stop packet may have to follow a prefix packet up into the first row of tcells).

Therefore, preparation for storage management begins within a minimum of 2H and a maximum of 2H + 2h + 3 time units after t_s. Preparation for storage management is pipelined on the way up, and involves a delay of 2 per tcell on the way down.* Including the time for unloading the pipe then gives a time of 3H + 2 for preparation for storage management.

Of course, in the presence of multiple RAs, the max t_s over all RAs must be used. The resulting bounds for the execution phase are summarized in Figure 5.10, using the established bounds for h and t_s.

* The reason for this delay may be understood from the discussion on preparation for storage management in Section 3.4.4.2. Only after both bc.left_entries and bc.right_departures are received from a parent can a tcell send left_entries and right_departures boundary condition values to its right child.

FIGURE 5.10 – Duration of the Execution Phase

(EX)	LB	UB
sm_grant	$max_{RAs} LB(t_s)$	$max_{RAs} UB(t_s)$
stop message	2H	2H
stop packet	0	2H+1
prep sm	3H+2	3H+2
total	LB(EX)	UB(EX)

$$LB(EX) \quad = \quad max_{RAs} LB(t_s) + 5H + 2$$
$$UB(EX) \quad = \quad max_{RAs} UB(t_s) + 7H + 3$$

5.4. Predicted Execution Time for Single RA

In our analysis for individual RAs, we do not count the storage management phase required to produce an RA in its initial form since this is not information that is available at this level of detail; the RA may have been shifted in as part of a new program, or it may have been created from the execution of previous applications. Later, when we treat complete FFP programs, this information will be available, and will be utilized. Also, we do not consider the cost of a final storage management in the case where an RA completes through the use of the **forkc** statement. This cost is also taken into account when we analyze complete FFP programs, but only when it affects the storage management time for a subsequently formed RA.

We now show how the formulas of Section 5.3 are used to predict the execution times for a variety of RAs. For each example, we first present the analytic model results for the general RA, and then provide a tabular comparison giving a particular simulation and the corresponding analytic model predictions. The basic format for the comparative presentation is shown in Figure 5.11.

The heading indicates the FFP operator whose behavior is being examined, lists the particular RA whose reduction is to be simulated, and gives appropriate parameters for the analytic model. The LPL program headers given in Chapter 2 provide the parameters used for different FFP operators.

Cycle 0 within the table represents past history, and is used to normalize the simulation times with analytic model times. Since we don't consider the time required to initially create the RA at this level of the analytic model, the time actually used in loading the simulation is used as an offset to the analytic model. This allows us to use the actual simulation times (without modification) for comparison with the predictions of the analytic model. The predicted and

FIGURE 5.11 – Analytic/Simulation Model Result Format

FFP OPERATOR
RA to be Simulated
Important Parameters

	ANALYTIC MODEL		*SIMULATION*	
	lower bound	*upper bound*	*observed*	*FFP text (eoc)*
0	*(==)*	*(==)*	*(==)*	*text 0: <>* *(op* *< args*
1	*(SM)* *(PN)* *(EX)* *(++)* *(==)*	*(SM)* *(PN)* *(EX)* *(++)* *(==)*	*(SM)* *(PN)* *(EX)* *(++)* *(==)*	*text 1: <op>=* *(op* *< args*

observed times for the three phases are displayed to the right of their parenthesized respective phase. The total time for an individual cycle is shown to the right of "(++)", and cumulative times are given to the right of "(==)". The rightmost column is used to describe the progress of the FFP-level reductions. The text given in the rightmost column is meant to be roughly indicative (given the limited space available) of the FFP symbols present at the *end* of the cycle. For cycle 0, the FFP segment to be loaded and used for the example is indicated in this space. When a **forkc** is executed, the result achieved upon completion of storage management is shown. The size of the LPL program to be loaded during partitioning, represented as usual by enclosing the operator name in angle brackets, is also given.

5.4.1. Analysis and Simulation of ID

The simplest LPL programs are those that require no forking and no messages. Operators such as CONST, SELECT, HEAD, APNDL, and ID fall into this category. Here, all the information necessary for completing the desired reduction is already present within the LPL environment for each lcell of the RA. These LPL programs therefore complete in one cycle and require no messages.

We now analyze the LPL program for ID given in Chapter 2. The initial storage management takes zero time (as explained previously). Partitioning the new RA then requires time based on the LPL program size (<ID>=29), the height of the active area, and the height of the tree, as shown in Figure 5.8. Since there are no messages, t_s in Figure 5.10 is zero. The predictions of the analytic model, and simulation results are given in Figure 5.12.

FIGURE 5.12 – Analysis and Simulation of ID

$(SM_1) =$
$(LB) = 0$
$(UB) = 0$

$(PN_1) =$
$(LB) = 4*\log(n) + 29 + 16$
$(UB) = 6H + 29 + 14$

$(EX_1) =$
$(LB) = 5H + 2$
$(UB) = 7H + 3$

(ID < a b c d e >) → < a b c d e >
H=4, n=8

	ANALYTIC MODEL		SIMULATION	
	lower bound	upper bound	observed	FFP text (eoc)
0	(==) 61	(==) 61	(==) 61	text 0: <> (id < a b c d e
1	(SM) 0 (PN) 57 (EX) 22 (++) 79 (==) 140	(SM) 0 (PN) 67 (EX) 31 (++) 98 (==) 159	(SM) 0 (PN) 59 (EX) 26 (++) 83 (==) 146	text 1: <id>=29 < a b c d e

5.4.2. Analysis and Simulation of N-ary Add

The LPL program for n-ary addition completes in one cycle, and provides an example of message use. As shown in Chapter 2, n-ary add operates by sending argument values up into the message subsystem, where they are combined using addition. The single result returns to the lcells, where it is accepted and stored as the desired result. It may be analyzed as follows, with values calculated for t_s parenthesised for clarity.

FIGURE 5.13 -- Analysis and Simulation of N-ary Add

$(SM_1) =$
$(LB) = 0$
$(UB) = 0$

$(PN_1) =$
$(LB) = 4*\log(n) + 53 + 16$
$(UB) = 6H + 53 + 14$

$(EX_1) =$
$(LB) = (2*\log(n)+3+4) + 5H + 2$
$(UB) = (2*\min(n,H)+2+4) + 7H + 3$

(+ < 1 2 3 4 5 >) → 15
H=4, n=8

	ANALYTIC MODEL		SIMULATION	
	lower bound	upper bound	observed	FFP text (eoc)
0	(==) 61	(==) 61	(==) 61	text 0: <> (+ < 1 2 3 4 5
1	(SM) 0 (PN) 81 (EX) 35 (++) 116 (==) 177	(SM) 0 (PN) 91 (EX) 45 (++) 136 (==) 197	(SM) 0 (PN) 82 (EX) 35 (++) 117 (==) 178	text 1: <+>=53 15

5.4.3. Analysis and Simulation of SORT

Sort is an example of an LPL program that sends many messages, but still requires no forking. It completes in one cycle, and if there are n numbers to be sorted, requires Θ(n) time. It is analyzed as follows.

FIGURE 5.14 -- Analysis and Simulation of SORT

$(SM_1) =$

$(LB) = 0$

$(UB) = 0$

$(PN_1) =$

$(LB) = 4*\log(n) + 59 + 16$

$(UB) = 6H + 59 + 14$

$(EX_1) =$

$(LB) = (2*\log(n)+3+3(n-3)) + 5H + 2$

$(UB) = (2*\min(n,H)+2+3(n-3)) + 7H + 3$

$$(S < 2\ 3\ 5\ 1\ 4 >) \rightarrow < 1\ 2\ 3\ 4\ 5 >$$
$$H=4,\ n=8$$

	ANALYTIC MODEL		SIMULATION	
	lower bound	upper bound	observed	FFP text (eoc)
0	(==) 61	(==) 61	(==) 61	text 0: <> (S < 2 3 5 1 4
1	(SM) 0 (PN) 87 (EX) 46 (++) 133 (==) 194	(SM) 0 (PN) 97 (EX) 56 (++) 153 (==) 214	(SM) 0 (PN) 88 (EX) 50 (++) 138 (==) 199	text 1: <S> < 1 2 3 4 5

5.4.4. Analysis and Simulation of ROTR

The more general situation for FFP operators is to require forking in conjunction with messages. Such operators require multiple cycles to complete since an intermediate storage management is required. An example of such an operator is ROTR. The rightmost argument element is sent over to occupy the leftmost position after room has been made for it by forking. As indicated by the header for its LPL definition in Chapter 2, ROTR completes in two cycles. With l denoting the number of elements of its argument list, and m denoting the size of the rotated element, its behavior may be summarized as follows:

FIGURE 5.15 -- Analysis and Simulation of ROTR

$(SM_1) =$

$\quad (LB) = 0$

$\quad (UB) = 0$

$(PN_1) =$

$\quad (LB) = 4*\log(n) + 143 + 16$

$\quad (UB) = 6H + 143 + 14$

$(EX_1) =$

$\quad (LB) = (2*\log(n)+3+4l) + 5H + 2$

$\quad (UB) = (2*\min(n,H)+2+4l) + 7H + 3$

$(SM_2) =$

$\quad (LB) = 204*2$

$\quad (UB) = 204*2$

$(PO_2) =$

$\quad (LB) = 2*\log(n) + 8$

$\quad (UB) = 4H + 8$

$(EX_2) =$

$\quad (LB) = (2*\log(n)+3+4m) + 5H + 2$

$\quad (UB) = (2*\min(n,H)+2+4m) + 7H + 3$

$$(rr < a < b > c < d > >) \rightarrow < < d > a < b > c >$$
$$H=5,\ n=9,\ l=4,\ m=2$$

	ANALYTIC MODEL		SIMULATION	
	lower bound	*upper bound*	*observed*	*FFP text (eoc)*
0	(==) 80	(==) 80	(==) 80	text 0: <> (rr < a < b c < d
1	(SM) 0 (PN) 175 (EX) 54 (++) 225 (==) 305	(SM) 0 (PN) 187 (EX) 66 (++) 249 (==) 329	(SM) 0 (PN) 179 (EX) 62 (++) 241 (==) 321	text 1: <op> (rr < a < b c < d
2	(SM) 408 (PO) 16 (EX) 46 (++) 470 (==) 775	(SM) 408 (PO) 28 (EX) 58 (++) 494 (==) 823	(SM) 408 (PO) 20 (EX) 52 (++) 480 (==) 801	text 2: <> << d a < b c

5.4.5. Analysis and Simulation of EE1

Forkc is ideal for supporting operators that require no further execution after an appropriate storage management. Its use in the EE1 functional avoids the cost of an extra machine cycle. Another interesting use of **forkc** is found in the COMP functional. We analyze EE1 here. The summary of analytic model parameters found in its header gives the following result, where m is the number of applications to be created.

FIGURE 5.16 -- Analysis and Simulation of EE1

$$
\begin{aligned}
&(SM_1) = \\
&\qquad (LB) = 0 \\
&\qquad (UB) = 0 \\
&(PN_1) = \\
&\qquad (LB) = 4\log(n) + 126 + 16 \\
&\qquad (UB) = 6H + 126 + 14 \\
&(EX_1) = \\
&\qquad (LB) = (2\log(n)+3+4(m+1)) + 5H + 2 \\
&\qquad (UB) = (2*\min(n,H)+2+4(m+1)) + 7H + 3
\end{aligned}
$$

(<EE1 +><<1 2 3><4 5 6>>) → < (+<1 4>) (+<2 5>) (+<3 6>) >
H=5, n=13, m=3

	ANALYTIC MODEL		SIMULATION	
	lower bound	upper bound	observed	FFP text (eoc)
0	(==) 88	(==) 88	(==) 88	text 0: <> (<ee1 + <<1 2 3 < 4 5 6
1	(SM) 0 (PN) 158 (EX) 54 (++) 212 (==) 300	(SM) 0 (PN) 170 (EX) 66 (++) 236 (==) 324	(SM) 0 (PN) 159 (EX) 60 (++) 219 (==) 307	text 1: <ee1>=126 <(+<1 4 (+<2 5 (+<3 6

5.5. Complete Programs

The above sections have shown how the execution times for individual RAs may be predicted. With appropriate restrictions, this approach may be extended to the analysis of programs for which multiple RAs execute concurrently. A simple example of this is given by a program to calculate the inner product of two vectors. Such an FFP program, one that uses the EE1 functional form, is given in Figure 5.17.

FIGURE 5.17 – A Program for Inner Product of <1 2 3> and <4 5 6>

```
(+ (< EE1 * > < < 1 2 3 > < 4 5 6 > > ))
```

Figure 5.18 shows a graphic representation for execution of this program, in which the creation and progress of individual RAs are depicted. To predict the execution time for this program, the same method used for individual RAs is employed. The progress of each RA through the three execution phases is tracked, with the primary difference being that creation time for RAs may now be taken into account.* Also, since multiple RAs are involved, upper and lower bounds for storage management will in general differ, and the maximum t_s value among the RAs must be used to determine the duration of the execution phase.

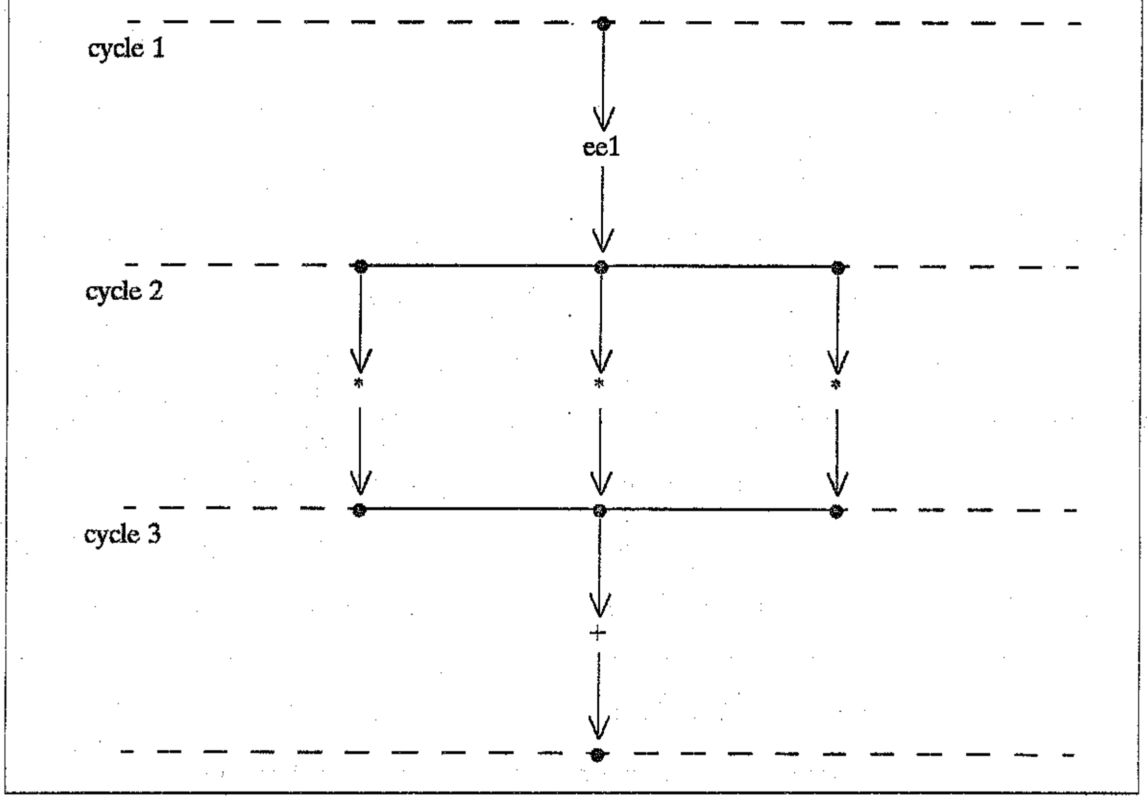
Assuming that H=5, the analysis is as follows. For cycle one, there is one RA for EE1. We know from Figure 5.16 that this first cycle will take between

LB(cycle 1)=212, and UB(cycle 1)=236 time units.

For the second cycle, we must analyze the multiplication operation. Multiplication can be considered an n-ary operation in the same way as addition, and the resulting LPL program mirrors that for addition. Thus the results of Figure 5.13 may be used. But first, we must analyze the time required to create the RAs. At the beginning of cycle 2, as shown in the header for EE1 in Section 2.4.1.8, there will be 3 contexts, each forking off 5 completed contexts. Thus, as required by Figure 5.6, $LB(SM_2)=4*4=16$, and $UB(SM_2)=4*(3*4)=48$. Using H=5

* The only uncertainty now is the time required to initially load the complete program. It seems reasonable to ignore this time in our analysis of execution time, so the first RAs to be detected in a program are assigned zero storage management time.

FIGURE 5.18 -- Execution Trace for Inner Product



and n=5 in the formulas for n-ary add found in Figure 5.13 gives results of $LB(PN_2+EX_2)=81+40=121$, and $UB(PN_2+EX_2)=97+54=151$, therefore

LB(cycle 2)= 137 and UB(cycle 2)= 199.

Since the RAs during cycle 2 are all performing the same parallel computations, their t_s values are all the same. If this were not the case, the maximum t_s among the RAs would be used.

In cycle 3, there is no storage management cost to be paid for creation of the addition RA since the multiplications of the previous cycle require no forking. Thus we have a final n-ary addition with n=6. This gives

LB(cycle 3)= 121 and UB(cycle 3)= 151.

The combined estimates yield LB(program)=470, and UB(program)=586.[*] Figure 5.19 summarizes these results, and presents the results of simulation.

FIGURE 5.19 – Analysis and Simulation of Inner Product

*(+ (< ee1 * > < < 1 2 3 > < 4 5 6 > >)) → → → 32*
H=5

	ANALYTIC MODEL		SIMULATION	
	lower bound	*upper bound*	*observed*	*FFP text (eoc)*
0	(==) 96	(==) 96	(==) 96	text 0: <> (< ee1 * <<123<456
1	(SM) 0 (PN) 158 (EX) 54 (++) 212 (==) 308	(SM) 0 (PN) 170 (EX) 66 (++) 236 (==) 331	(SM) 0 (PN) 162 (EX) 60 (++) 222 (==) 318	text 1: <ee1>=126 < (* < 1 4 (* < 2 5 (* < 3 6
2	(SM) 16 (PN) 81 (EX) 40 (++) 137 (==) 445	(SM) 48 (PN) 97 (EX) 54 (++) 199 (==) 531	(SM) 32 (PN) 90 (EX) 42 (++) 164 (==) 482	text 2: <*>=53 (+ < 4 10 18
3	(SM) 0 (PN) 81 (EX) 40 (++) 121 (==) 566	(SM) 0 (PN) 97 (EX) 54 (++) 151 (==) 682	(SM) 0 (PN) 86 (EX) 44 (++) 130 (==) 612	text 3: <+>=53 32

5.6. Restrictions

The above analysis of the inner product program was easy for a number of reasons. The multiple RAs for cycle two all had the same t_s value. In general, of course, this will not be the case. Also, the program contained no conditional execution paths, and was neither recursive nor iterative.

Koster [Kos79] has dealt with conditional execution and has shown how to use recurrence relations to analyze programs that perform recursion or

[*] Note that these bounds do not include the initial load time of 96, which is used to normalize analytic model predictions for comparison with simulation results. Including this value yields total estimates as shown in Figure 5.19.

iteration. In addition, Mago, et. al. [Mag83] have identified a set of restrictions for parallel RAs that serve to guarantee a determination for $\max_{lcells}(t_s)$ for each cycle.

The basic difficulty in determining $\max_{lcells}(t_s)$ is that parallel execution paths may individually require numerous sequential reductions, and to predict $\max_{lcells}(t_s)$ for each cycle, we must know what RAs are executing. Thus we need be able to construct a graph of the execution paths, similar to that shown in Figure 5.18, which at least parametrically includes this information. If parallel execution paths are allowed different data-dependent behavior (perhaps one path involves sorting, and another parallel path requires a conditional matrix transposition) then such a graph cannot be constructed. A useful set of restrictions, suggested by Mago, et. al. [Mag83], are given in Figure 5.20.

FIGURE 5.20 – Restrictions on Parallel RAs for Analyzability

1) The number of parallel execution paths for a program is known, at least parametrically

2) One of the following holds for parallel execution paths:
- *no RA requires messages;*
- *the RAs along each path are identical; or*
- *in each path, only the last RA is allowed to send messages whose number is known only at run-time.*

By constraining the dissimilarity of parallel execution paths, these restrictions define a class of FFP programs for which lower and upper bounds on execution time are easy to derive. The situation is similar to that for von Neumann programs; they are generally not analyzed unless they are suitably structured and the data characteristics are sufficiently predictable.

5.7. Summary

This concludes our discussion of the analytic model. In our initial approach to designing DOT, many aspects have been simplified in the interest of furthering insight into (and identification of) the important problems facing an efficient implementation. The analytic model presented is a great help in this respect, since it is based on the design representation and gives useful predictions for performance. The resulting insights aid investigation of ways to improve the design, as the following chapter on design alternatives will show.

In the discussion of other reduction machines given in Chapter 1, we pointed out the importance of limiting process interference -- both for reasons of performance, and predictability. We can now characterize the degree to which we have been successful in this.

In the context of DOT, the progress of an individual RA through the phases of the machine cycle may be viewed as a process, and it is therefore interference between parallel RAs that must be examined. As shown by the analytic model we have presented, execution of parallel RAs generally proceeds with very little interprocess interference. As much as possible, we have tried to decouple the processing cells of DOT so that the progress of any RA through the three phases of the machine cycle is relatively independent of other RAs. This is the primary reason why a useful analytic model of program execution on DOT can be developed.

There are two ways that RAs may still interfere with each other. During every machine cycle, each RA determines a local t_s and a corresponding lower bound for $\max_{lcells}(t_s)$. The greatest such lower bound, however, determines the actual duration of the execution phase for all RAs. The penalty for this interference is that RAs that complete without messages may have to wait on

RAs that do require messages. As shown in the above examples, however, messages are handled efficiently by DOT. In addition, Chapter 4 on simulation suggested that the penalty for being "fair" (i.e., sending only a limited number of messages per cycle) is not serious.*

The other possibility for process interference is during storage management. As shown by the analytic model, the situation here is more serious in terms of its possible impact on performance. During storage management, RAs and entering new programs must compete for space within the lcell array. Space requirements for all RAs may be satisfied, but at the cost of shifting some RAs a great distance through the lcell array. Storage management is performed in such a way as to limit this kind of interference, and simulation results confirm that this is generally successful. Nevertheless, the cost of shifting complete LPL program contexts within the lcell array is the main performance bottleneck of DOT. This cost is expected; it is the price to be paid for the benefits of string reduction enjoyed throughout the rest of the machine cycle. Because of this cost, however, the greatest improvements in performance will most likely result from reducing the amount of information shifted during storage management.

Possible approaches include modifications to LPL that enable an increase in the efficiency of storage management, and modifications to DOT that allow further de-coupling of the machine cycle phases within separate RAs. As an example of the first category, the **forkc** statement drastically reduces the size of contexts that are forked (in addition to saving an execution cycle) from 204 bytes to 4 bytes. Approaches in both categories are considered in the following

* Also, our approach for determining the duration of the execution phase can be easily changed to remove this interdependency between RAs. The next chapter will discuss various alternatives.

chapter on design alternatives.

CHAPTER 6

Design Alternatives and Extensions

Numerous alternatives are possible within the design space of the programming system we have described. By discussing these alternatives now, we clarify many of the dimensions of the design space, identify tradeoffs, and examine the flexibility and potential of the design we have constructed. Many of the tradeoffs do not lend themselves to a formal analysis, so an important use of the simulation will be to portray the behavior of the programming system under the influence of alternative approaches.

Clearly, depending on how pervasive a particular design decision is, modification of DOT to reflect an alternative approach will require changes of varying scope within the simulation. For each alternative identified, we will therefore be concerned with this practical issue as well as the possible benefits to be realized by making changes to the design.

In addition to design alternatives, this chapter also discusses design extensions. Both involve changes or modifications to the present design, but extensions do not involve tradeoffs in the same sense as the alternatives; extensions may be viewed as holding clear-cut benefits for the programming system. They represent our suggestions for work that definitely should be done in order to further improve the desirability of the programming system.

An overview of the design alternatives and extensions that we will discuss is shown in Figure 6.1. Each possible change is given under the topmost system level affected, with the understanding that lower levels may also be affected.

FIGURE 6.1 – Possible Design Modifications

1) ALTERNATIVES
- *A. FFP Level*
  - *Text Representation*
- *B. LPL Level*
  - *Message Routing*
  - *Non-blocking Fork*
  - *Synchronization of Segment Completion*
- *C. DOT Level*
  - *Duration of Execution Phase*
  - *Shifting vs. Reloading LPL Code*
  - *Storage Management Transfer Function*

2) EXTENSIONS
- *A. FFP Level*
  - *JCL for User Programs*
  - *Temporary Storage (PUSH, POP Operators)*
  - *Visual Tracing*
- *B. LPL Level*
  - *Event Indicator for Storage Management*
- *C. DOT Level*
  - *Variable Context Sizes*
  - *Increased Phase Independence for RAs*
  - *Multiple LPL Program Input Ports*

6.1. Design Alternatives

First, we discuss possible alternatives to the current design.

6.1.1. FFP-level Text Representation

In Chapter 2, we presented an FFP-level representation for user programs based on the use of nesting level numbers. This representation was suggested by Mago in his original description of the tree machine [Mag79]. From this, we derived an LPL representation that includes information required for efficient use of the LPL multiprocessor architecture.

Since the DOT implementation is driven by these two architecture levels, the initial choice of the FFP-level text representation is clearly of central importance to the whole programming system. Modification of the design to support a different FFP-level representation within the machine would require pervasive changes. The detailed nature of the DOT simulation (which mirrors and represents the design) therefore precludes a straightforward investigation of possible alternatives at this level.

This category of alternative is mentioned here to underscore its primary importance to the overall multiprocessor design, rather than to recommend its investigation through use of the current simulation. By performing another design in parallel with the one we have described, David Middleton, here at UNC, is investigating other FFP-level representations, including the *PC Representation* (for *Potentially Compact*) originally used by Tolle [Tol81].

6.1.2. LPL Message Routing

The LPL architecture does not include the tree structure that is used to implement it. The primary reason for this is our desire for simplicity. As we have made clear, DOT is quite complex in its operations, and simplicity is thus a distinct virtue wherever possible. But, when simplicity is bought at the ultimate cost of efficient performance, alternatives should be at least identified for investigation by future workers.

As we have shown, simple combining and sorting operations are handled efficiently in the current DOT implementation -- without the need for explicit incorporation of message routing into the LPL architecture. However, some operations might benefit by allowing explicit LPL control over the routing of messages among the tcells of an active area. For instance, Pargas [Par82] has shown how generalized routing may be used for efficient solution of partial differential equations. In addition, Presnell and Pargas [Pre81] have examined the use of shortest path routings in tree machines.

Presnell has suggested a simple generalization of the LPL message scheme that allows shortest path routing. In this approach, the message prefix could differ between the messages of a given message wave, allowing area nodes to route messages in different ways depending on prefix instructions and message data. Messages would be routed down to a left or right child, or up to a parent as done presently. Downward-moving messages would be handled via broadcast as in the current implementation.

The main performance penalty to be paid for this approach is the use of multiple prefix packets where, before, a single pipelined prefix sufficed. Also, since each message must be handled separately, messages can no longer be pipelined. In cases where an $\Theta(n)$ dependence on the size of an RA may be avoided, however, this would be a small price to pay. The increased overhead for message processing could be made up for by reduced traffic through the top of area -- enabling more balanced communication loads and increased utilization of the area nodes.

While the ramifications of the above message protocol require further investigation, implementation within the current design context appears feasible. This would allow simulation to aid analysis of the tradeoffs involved. Synchronization will be required within area nodes to handle non-deterministic arrival of messages from above while locally routing messages down to a child. In its present form, DOT requires no such synchronization within the tcells.

6.1.3. Non-blocking Fork

Since storage management potentially represents the most expensive phase of machine execution, it is important to consider design alternatives that reduce the need for shifting within the lcell array. One such possibility involves virtualizing the lcells of the LPL architecture, so that a single DOT lcell may support a contiguous segment of forked LPL-level (virtual) lcells.

To see why this could be valuable, consider the following. At present, FFP reductions can require an expression to grow temporarily within the lcell array even though the final reduced result is no larger than the original application. Operators that merely restructure a list (e.g., ROTR) provide examples of this phenomenon; forking is used to create room to receive symbols whose position is to be changed, and whose original containing lcells are released when the reduction completes. In such cases, there is no net increase in expression size, and virtual lcells could be used to temporarily contain (within a single DOT lcell) multiple forked LPL contexts during the process of a reduction, so that no intermediate shifting would be required.

This would minimize the degree to which separate RAs interfere with each other during their execution, and allow most FFP primitives to be implemented in a single cycle machine since the LPL fork operations could proceed without storage management within the lcell array. Shifting would be required only upon completion of a reduction -- to create the one-to-one correspondence between FFP-level symbols and DOT lcells required for new partitionings.

The basic concept is thus similar to multiprogramming on traditional architectures, with the exception that only a single LPL code segment would be required. The tradeoff to be examined is the necessary increase in the size and complexity of DOT lcells -- needed to allow an lcell to contain, schedule, and execute multiple user contexts -- versus increased independence between RAs and increased execution efficiency for many FFP operators.

Implementing non-blocking LPL fork operations appears feasible within the DOT model. Message reception would be straightforward; copies of the message would be placed in each LPL context, and the appropriate filter would be executed once for each context. Allowing **send** statements would be more difficult, since combining or sorting messages would have to be done by the containing DOT lcell. The restriction that no **send** statements be executed following a forking operation seems reasonable, however, and takes care of this problem. Support for fork operations that create more LPL contexts than can be held by a single (limited size) DOT lcell must also be addressed. In such cases, storage management will be required, but the implementation should be able to handle this in a manner that is transparent to the LPL programmer (just as virtual memory or multiprogramming is transparent to a programmer).

6.1.4. Completion Synchronization

Because of our process-oriented design methodology, and the desire for simple, asynchronous and free-running processes wherever possible, each segment of an LPL program was originally allowed to simply perform its own local duties and then complete (by executing an **endsegment** statement). Although messages and forking might require multiple cycles for some of the segments of an LPL program, other segments requiring fewer cycles were allowed to complete without concern for the longer-running segments. This was a convenience provided by DOT for the LPL programmer since, in reality, the lcells of an RA must all be stepped forward together at the end of the same cycle to prevent partitioning anomalies.

To insure that RA lcells were not stepped forward too soon, DOT lcells within an RA originally sent a *state packet* up into the area channels upon receiving the stop message from the io subsystem. This packet served the same function as the LPL stop packet now does, by clearing out area channels, but also included the execution state (completed or not) of the sending lcell. The state information was combined by area nodes using logical multiplication on the way up, and upon return to the lcells indicated whether all lcells of the RA had completed and the RA should be stepped forward.

The original method had the advantage of insulating the LPL programmer from details required for synchronizing completion of an RA, but was less efficient than necessary. If LPL segments are required to synchronize their own completion (which they can easily do by keeping track of messages, or forks when necessary) then the state message is not required, and preparation for storage management can begin earlier, thus shortening the execution cycle.

The approach we therefore took, as described in Chapters 2 and 3, was to require LPL programs to perform their own completion synchronization. With this done, a top of area node can detect the stop message on its way down the tree, and immediately insert a corresponding stop packet into its down-going area channels (following the LPL message currently in transit, if any) without waiting for the stop message to reach the lcells and the state packet to then be sent up and return to the top of area. Doing this therefore avoids an additional 2h delay required by the state packet approach.

Our experiences with the improved shutdown mechanism when implemented in the simulation were surprising. In some cases, the "improved" design actually ran slightly slower. The reason for this was the additional LPL code required to synchronize completion. When h was small, the increased code loading time during partitioning (due to increased code size) was not offset by the 2h saving during shutdown. With larger areas, the desired effect is achieved and synchronized completion results in the best performance. Nevertheless the tradeoff remains interesting. LPL programs are considerably harder to write with a synchronization constraint between segments. Many of the LPL programs rewritten for the new design initially deadlocked, and new LPL programs take longer to develop.

In the current design, we nevertheless decided to use the approach that is potentially more efficient at the cost of ease in writing LPL programs. This approach can be supported by arguing that LPL programs are essentially microprograms, and therefore will be written infrequently. On the other hand, it is reasonable to note that even on a large machine (say H=21) h will never be prohibitively large, so the extra delay required to send a state packet may not be terribly important when compared to realistic storage management times.

Looking at the benefits to be derived through the use of a state packet (in spite of the execution overhead), the resulting decrease in LPL code size is not a major factor; the increase in understandability is. Both factors are most noticeable when the LPL program in question requires multiple cycles and numerous code segments.

6.1.5. Duration of Execution Phase

Since the stop message which originates in the io subsystem initiates the required course of events for terminating the execution phase, regulating the duration of the execution phase reduces to deciding how soon (following delivery of required LPL code segments) the stop message should be sent. It is useful to note that this decision can be made without concern for correct operation of the machine. This fact is a natural result of a process oriented design methodology. Time is *never* a factor in the correctness of the design -- only the partial orderings of events made explicit in the process descriptions. Because of this, we are free to use whatever means we wish to determine an appropriate duration for the execution phase.

A variety of alternatives for regulating the duration of the execution phase are possible. These include fixed durations, and heuristically varied durations. The approach we have taken for DOT allows RAs to control the cycle time, thus providing a degree of sensitivity to dynamically changing execution requirements, while still allowing an analytic model to predict execution times. This method can result in the shortest possible cycle times and the best possible execution efficiency when all RAs receive few (or no) messages before completing or forking.*

If $\Theta(n)$ messages must be received by the lcells of an RA in order to complete a reduction, however, allowing the RA to complete in a single execution phase could seriously reduce the efficiency of other reductions within the machine. It seems best to interrupt such an RA temporarily, so that other reductions that have completed can be stepped forward, and then continue the RA during the following machine cycle. In the present design, LPL programs that allow this, by executing **smanage** at appropriate intervals, are called *fair operators*.

As shown by simulation, the cost of being fair may not be serious. This is due to two factors: the lower bound on storage management for an RA interrupted for this reason is generally zero (messages are being received, and local forks have not been executed), and the subsequent cost for repartitioning the old RA is low (code and directories are already loaded).

As an alternative to the current approach requiring fair operators, we might consider using a fixed duration for the execution phase. This would also prevent a single RA from monopolizing the machine cycle. The analytic model of RA execution time can be used for guidance in choosing an appropriate duration.

With the help of the formulas derived in Chapter 5, we can investigate the normalized efficiency, ξ (defined as the time required to reduce an RA

* As shown by the analytic model, only the number of received messages is important to execution time; any number may be sent, but if combined and pipelined the delay is the same as the transit time for a single message.

completely within one cycle divided by the cost of reducing the RA over a period of k additional cycles), for a large $\Theta(n)$ communication-bound reduction. The primary assumptions required are that the lower bound on storage management time for the RA is zero (i.e., no forks are performed), and that other RAs don't interfere with the communication-bound RA, so that the actual storage management time is zero as well. As mentioned above, these assumptions appear reasonable in view of early simulation results.

Figure 6.2 summarizes the analysis when n messages of msize=1 are received by an RA of size n.

FIGURE 6.2 – Efficiency of Fair Operators

ξ = *best-time / actual-time*

message-cost	$\simeq 4n$,
partitioning-cost	$\leq 4H$,
total-cost	$\leq 4kH + 4n$, *thus*

$$\xi \geq 4n \,/\, (4n + 4kH)$$
$$\geq 1 \,/\, (1 + kH/n).$$

If the constant cycle time is equivalent to m messages/cycle, $k = n/m$, *so*

$$\xi \geq 1 \,/\, (1 + H/m).$$

Alternatively, to achieve a given efficiency, ξ, *let*

$$m \geq \xi H \,/\, (1-\xi).$$

From the results of Figure 6.2, to achieve 50% efficiency in communication-bound RAs, the machine cycle should be set so at least H messages may be sent each cycle. For a machine of height H=20, to achieve 50% efficiency the duration of the execution phase should be at least (2H + 4H + 3) = 123τ. If τ is

100 nsec, then, the duration of the execution phase should be $\simeq 13\mu$sec.*

Note that this value is fairly small in comparison with the times required to shift contexts within the lcell array. Presently, the time required to receive 20 LPL messages is less than the cost to shift an executing user context a distance of one lcell. In view of this result, decreased efficiency for RAs that don't require the whole execution phase for completion may not be an important factor. This is because the newly-formed expressions created by rapidly reduced RAs (during the ongoing execution of a large communication-bound RA) will often require storage management shifting due to forks that have been executed, and then storage management costs should predominate.

Although simulation will be required to judge whether the assumptions of this reasoning are born out by experience, the above discussion shows one direction that a search for a constant execution phase could take. With such an approach, execution times for complete programs could still be predicted in a manner similar to that described in Chapter 5.

With a heuristic cycle time, it is no longer possible (in general) to predict execution times. Nevertheless, a heuristic approach might still be indicated if the observed results were good. Figure 6.3 lists some of the approaches that could be considered.

FIGURE 6.3 -- Possible Variable Cycle Time Heuristics

1) less than 100% sm_grants required from lcells (e.g. 80%)
2) set cycle times based on RA operators
3) set cycle times based on storage management shift requirements

* Alternatively, to achieve 90% efficiency at least 9H messages should be received each cycle. With H=20, this would result in an execution phase duration of (2H + 36H + 3) = 763τ, or $\simeq 77\ \mu$sec.

In the first approach, a mechanism similar to the present use of sm_grant messages would be appropriate. Instead of waiting for all lcells to allow storage management, however, a particular threshold would be used (e.g., the stop message might be sent down after 80% of the lcells allow storage management). The threshold could be chosen dynamically each cycle, based on lcell contents, or fixed arbitrarily.

In the second approach, the particular LPL programs active within the lcell array (and possibly their corresponding RA sizes) would be used to determine an appropriate duration of the execution phase. Such information could easily be made available to the io subsystem during partitioning. Determining a satisfactory approach toward such an heuristic, by weighting various LPL program characteristics and RA sizes, would be very interesting.

In the third approach, information concerning the extent and distribution of storage management shift requirements would be used. When large amounts of shifting are required to create an RA, a longer execution phase would allow the shifting activity to complete and some useful work to be performed before termination of the execution phase.

Of course, the essence of an heuristic is that it attempts to balance complex and conflicting forces through simple means. The fact that the above examples are so different merely indicates the variety of factors that influence execution efficiency within the current design. In addition, the results of Figure 6.2 indicate that duration of the execution phase, although important, is not the predominant factor influencing execution efficiency -- rather, the duration of the storage management phase appears to be the most crucial. This is a useful result, since it indicates that a simple fixed duration for the execution phase may turn out to be generally satisfactory.

6.1.6. Shifting vs. Reloading LPL Code

At present, the LPL code segment is part of the user context that is shifted during storage management. This means that once a code segment is loaded, successive partitionings complete much more rapidly, and this allows good efficiency for fair operators. However, loading LPL code segments during each partitioning would reduce the time presently required for storage management by a factor of three.* This is clearly an important tradeoff, and it involves the number of FFP operators that are active during each cycle, and the number of machine cycles required for their reductions.

It is likely that a small number of commonly used LPL programs could be stored in ROM within the lcells. This would shift the balance toward loading necessary LPL programs during each partitioning, as opposed to shifting them with active environments during storage management. If non-blocking fork operations are feasible, most reductions would complete in one cycle anyway, so the frequency of reloading LPL code would be reduced even further.

Investigation of this tradeoff will require only moderate changes to the current design.

6.1.7. Storage Management Transfer Function

Calculation of the storage management transfer function is relatively unconstrained as long as it results in a feasible solution as described in Chapter 3. In addition, storage management is the most important phase to handle efficiently because of its great potential effect on execution efficiency. A variety of alternatives should therefore be identified and investigated. An important consideration for any method is that it should pipeline effectively.

* Recall that the LPL context size is 204 bytes, of which 150 bytes are used to hold the LPL code segment.

Although we use the method originally suggested by Mago that minimizes movement between subtrees rooted at higher levels of the machine [Mag79], Stanat and Mago have shown how to optimize the overall context movement by minimizing the maximum distance traveled by any context during storage management [Sta81a].

Other possibilities include purposely distributing available empty lcells among user contexts. This could serve to insulate separate RAs from each others' storage requirements, resulting in execution times generally close to the theoretical lower bounds predicted by the analytic model. If effective, such an approach would shift symbols farther than necessary in order to produce interspersed empty lcells and reduce the need for shifting later. To lower the cost of shifting symbols further than necessary, this activity might be restricted to non-active contexts only.

In the absence of advance knowledge concerning future storage demands of executing FFP programs, however, the method we presently use is probably close to optimal; it pipelines efficiently, and limits shifting during storage management effectively. In order to provide better overall performance during the execution of complete programs, guidance concerning beneficial placement of interspersed empty lcells from the FFP programmer might be useful. Automated analysis of FFP program text might also provide information useful to effective management of the lcell array during execution.

6.2. Design Extensions

We now give recommendations for design extensions.

6.2.1. Job Control Language

At present, the *mkusr* program accepts user-supplied FFP programs and creates a batch of user programs organized as required for loading and execution on the DOT machine. The FFP programs are presently written using the aln FFP-level representation, and the primary job of the *mkusr* program is to create a file in which user program symbols and alns occur in pairs in reverse (right-to-left) order as required for loading. This file is accessed during a simulation run by the vm subsystem, which enters the batched programs into the lcell array when indicated by the top-level storage management transfer function.

While this approach is quite satisfactory in its support for testing the DOT design and simulating the execution of FFP programs for evaluation of tradeoffs, a more realistic user interface will ultimately be required.

An interface between the outside world and the DOT machine that allows entry of jobs concurrently with machine operation is necessary. This should be fairly easy to develop within the present simulation. More important than this, however, is development of a smarter mkusr program. Facilities that mkusr could ultimately include are given in Figure 6.4.

FIGURE 6.4 – User Interface Facilities

- *Translate user FFP to the machine's FFP-level representation*
- *Translate operator names to the appropriate machine op-codes*
- *Translate FP to FFP*
- *Support user-defined operators*
- *Support run-time data entry*

The first three items simply require the development of a more sophisticated translation mechanism than now employed. The last two, however, will in addition require development of a Job Control Language to allow the user to define and use logical identifiers for reference to desired user-defined operators and read-only data sets. Corresponding to this JCL must be a runtime system to allocate unique identifiers for binding to logical identifiers during program execution.

When a program is run, its user-defined operators and read-only data sets will be associated with appropriate unique identifiers and placed (at least logically) within the LPL program library. When the program has completed, the associated user-defined operator definitions will be removed, and their op-code identifiers freed to allow allocation to the user-defined operators of new programs. This facility will allow general programs to be written in a structured fashion (data and program may be kept separate, and user-defined operators are similar to procedures).

JCL support for user-defined operators and read-only data sets will generate LPL programs (with temporary op-codes assigned as described above) that create the desired FFP text within the lcell array when encountered as the operator of an innermost reduction. Automatic generation of this restricted type of LPL program should be straightforward. Figure 6.5 shows a user-defined operator and gives the corresponding LPL program. The operator calculates the Euclidean distance from the origin to a point. Although FFP function names are used in the LPL program for clarity, the appropriate op-codes would actually be used. The LPL program source given in Figure 6.5 is only an intermediate step on the way to the corresponding object code. In practice, the required object code would be generated directly from the user-level FFP operator definition.

FIGURE 6.5 – A User-defined Operators and its LPL Program

Defuserop
```
DIST ≡
        <COMP + <CONS
                <COMP * <CONS <SELECT 1> <SELECT 1>>
                <COMP * <CONS <SELECT 2> <SELECT 2>>
                >
        >
```

```
program DIST /* called in when innermost (DIST arg) is encountered
   destination 1 0 0 0 /* replace DIST with its definition
     cselect "< : 0 "COMP : 1 "+ : 1 "< : 2 "CONS : 3
             "< : 3 "COMP : 4 "* : 4 "< : 5 "CONS : 6
             "< : 6 "SELECT : 7 #1 : 7 "< : 6 "SELECT : 7 #1 : 7
             "< : 3 "COMP : 4 "* : 4 "< : 5 "CONS : 6
             "< : 6 "SELECT : 7 #2 : 7 "< : 6 "SELECT : 7 #2 : 7
             .
     forkc #27
     endsegment
   destination 0* 0* 0* 0*      /* all other symbols unchanged
     keep
     endsegment
   endprogram
```

6.2.2. Pushdown Storage for Lcells

From the nature of the DOT design, it is clear that argument copying should be avoided by LPL programs whenever possible because of the corresponding necessity for forking and increased storage management time. Some LPL programs cannot avoid this. DBL and ROTR, for instance, must copy all or parts of their argument to produce the desired result. COND, the first phase of the FFP conditional operator, also needs to copy its argument, but not because of the result it produces. Rather, a temporary copy of the argument for COND is required so that the predicate can be evaluated -- off to the side, as it were -- in order to apply the correct function. To illustrate this, Figure 6.6 gives an example reduction for an FFP text segment that uses COND to return 1 if the argument length is less than 10, and 2 otherwise.

FIGURE 6.6 -- COND Copies its Argument

```
(<COND <CONST 1> <CONST 2> <COMP <LT 10> LENGTH>> arg)
        the argument is copied, and an inner evaluation of the predicate
        is begun
(<COND2 <CONST 1> <CONST 2>> <(<COMP <GT 10> LENGTH> arg) arg>)
        within in predicate evaluation, composition results in
(<COND2 <CONST 1> <CONST 2>> <(<GT 10> (LENGTH arg)) arg>)
        which ultimately reduces to, say,
(<COND2 <CONST 1> <CONST 2>> <T arg>)
        cond2 sees that the first function should be applied to arg, so
        it creates the appropriate reduction
(<CONST 1> arg)
        which reduces to
1
```

Note that in the first step, the size of the RA may double. This overhead of argument copying required for support of COND is unfortunate since conditional execution is generally necessary in realistic programs. Mago has suggested a method to avoid argument copying in such cases [Mag82]. The mechanism is interesting since its implementation requires changes to both DOT and the LPL architecture.

The basic idea is that lcells are given a pushdown register capable of saving the FFP-level representation for a text symbol. A *push* operation copies a symbol into this register, which is not affected by following reductions. A subsequent *pop* then brings the symbol back into the lcell array to again participate in reductions. For COND, then, we push the argument down, and evaluate the predicate, destroying the original argument. We then pop the earlier-pushed copy of the argument back up for use by the appropriate function.

The push operation may be implemented within FFP or LPL. Since efficiency is of concern, LPL is the best place for it. An extra cycle would be required if pushing were done at the FFP level. In addition, including a push operation in FFP would require modification of the FFP architecture -- something of concern in a language-based design.*

The pop operation, on the other hand, cannot be easily implemented within LPL. This is because only RAs execute LPL, and the pushed symbols are not allowed to affect partitioning and the creation of RAs. Thus, there is no easy way to guarantee that lcells holding pushed symbols will be part of an RA whose lcells might execute a pop instruction. Unfortunately, placing the pop operation in FFP is also unfeasible -- for the same reasons as given above for the push operation.

A compromise approach is to place pop in an intermediate position between the LPL and FFP levels. Pop can then be considered a "pseudo-operator" possibly found at the FFP-level during partitioning, but always pruned out of the area before reduction begins, and only placed in the lcell array by LPL code as a special non-FFP reserved symbol. Figure 6.7 shows the above example as it might appear during successive execution cycles with use of the LPL push statement and the pop pseudo-operator. The pushed symbols of the argument are represented in curly brackets. Pop is represented by ↑.

* Currently, the FFP architecture is based on linear (one dimensional) expressions. The push operation requires that a second dimension be visualized.

FIGURE 6.7 – COND with Push and Pop

```
(<COND <CONST 1> <CONST 2> <COMP <LT 10> LENGTH>> arg)
        The COND LPL program pushes the argument, and creates an
        innermost RA for evaluation of the predicate
(<COND2 <CONST 1> <CONST 2>> <(<COMP <LT 10> LENGTH> arg {arg})>)
        The predicate is ultimately reduced, yielding
(<COND2 <CONST 1> <CONST 2>> <T {arg}>)
        COND2 checks the result and then creates
(↑ <CONST 1> {arg} )
        During partitioning, the pop pseudo-op is detected. Only in
        this case are pushed symbols included in the RA, and they appear
        in their popped up form. The pseudo-op is pruned from the
        active area during the downsweep, so that the RA seen by CONST is
(<CONST 1> arg)
        which reduces as desired.
1
```

As can be seen, the above approach requires no argument copying, no extra cycles, and no changes to user-level FFP. It is important that the pop pseudo-op be pruned from the area so the RA will appear as expected by the function selected by the predicate. In the form described, push operations may not be nested. With further extension of the push-down storage in the Icell, nesting to a fixed depth would be possible.

Although a variety of modifications of DOT are required to support push and pop, these changes are not complex. Storage management should never shift contexts "over" pushed symbols. With this restriction, pushed symbols will always be available for inclusion in an RA created by a subsequent pop operation. As indicated above, partitioning must be changed so pushed but empty cells are treated as empty unless a pop pseudo-op is detected. In this case, the pushed symbols are popped and included as usual FFP-level symbols in that RA, while the pop pseudo-operator is pruned and the containing Icell made empty.

6.2.3. Visual Tracing

At present, all simulation output is oriented towards a terminal or line printer. Since detailed information concerning computational activities and data transfers is available from simulation tracing, however, the possibility of visually oriented simulation output is raised.

In addition to making visible the flow of information and the progress of FFP reductions within the DOT machine, such a trace package could also provide statistical assistance, and allow run-time factors such as channel utilization to be graphically displayed. Simulation results of interest in evaluating design alternatives could be accumulated and presented at the user's request.

Figure 6.8 shows the steps necessary to provide such a facility.

FIGURE 6.8 -- Steps to Provide Visual Tracing

- *develop a graphical machine model*
- *modify the simulation to produce appropriate data*
- *connect simulation output to the visual model*
- *provide user interaction*

An appropriate visual model might employ graphical representations similar to those used within this dissertation to depict a simulated machine, but flexible windowing operations would be required. The largest machine depicted within this dissertation contained only 16 lcells, while the machines required for realistic simulations will be much larger. Because of this, the ability to handle different levels of abstraction in the visual model, by using different representations of the cells and communication lines, would be desirable. At a top level, visual access to windowed segments of the lcell array would allow storage management and the ongoing progress of FFP text reductions to be examined. At a lower level, a more detailed visual model might include the tcells as well, highlighting active areas, and portraying the flow of information between cells during execution.

At present, simulation output is keyed to process ids that are maintained by the ClassC scheduling mechanism. The correspondence between process ids and cell locations must be made more explicit if trace information is to be displayed graphically. The most direct approach would be for the geometric location of each cell to be encoded into a variable globally available to members of each cell during initialization, and for every trace message to include this information. Making such a modification to the current simulation will be easy, since this facility was envisioned during the initial design and the appropriate hooks are in place.

Although it would be possible to pipe simulation output directly to a visual trace package, it seems better to batch the simulation as done presently, and use the simulation output later -- filtering it as appropriate for the desired visual trace. One reason for this is that simulation time can progress quite slowly. Program execution can require hours of wall time when large machines are simulated. By decoupling simulation from visual tracing, reasonable viewing times are made possible.

Performing visual tracing separately from simulation also expands the possibilities for user interaction. The progress of simulation time might be a variable to be selected by the user during tracing. Interactive windowing of the visual model would also be very useful. The ability to change levels of abstraction interactively is another possibility.

In summary, visual tracing represents a useful and relatively straightforward improvement to the current simulation environment, and it is highly recommended.

6.2.4. LPL Storage Management Event Indicator

A useful (and easily accomplished) extension for LPL would be to provide a read-and-reset register that indicates whether or not storage management has occurred since the last time the register was examined. This facility would be oriented towards support for fair FFP operators, and would allow an exact count of the number of messages processed since the last storage management.

At present, fair operators incorrectly assume that no messages are received following an **smanage**. This assumption is made because storage management is completely invisible to LPL code, thus a count of messages received in the interim between the **smanage** and the end of the execution phase is not possible. A read-and-reset boolean register to reflect the occurrence of storage management would allow the correct message count to be maintained.

6.2.5. Storage Management with Variable Context Sizes

At present, two different sizes of user contexts are shifted during storage management: active (204 bytes) and non-active (4 bytes). Besides the 54 bytes of LPL environment included in an active context, a fixed size code area of 150 bytes is shifted. Since code segments are often quite small (many contain only a one byte long endsegment op-code), a more flexible approach is indicated. Storage management should only shift the actual amount of code that is resident -- not the complete code area. This could result in greatly improved performance.

6.2.6. Increasing Phase Independence

The efficiency of the DOT implementation arises from decoupling as much as possible the phases of the machine cycle in separate RAs. Unfortunately, coupling still occurs during storage management, when large user contexts are shifted in pipelined fashion in the lcell array. We might ask if we can further decouple the activities of different RAs during storage management as well. Luckily, we can.

The key is to recognize that partitioning requires only the ***symbol***, ***aln***, *state*, and ***forkid*** context information. This could be shifted first, during an initial storage management phase, allowing partitioning to begin as soon as possible and proceed while the shifting of all other context information (required for execution only) takes place. The storage management phase would therefore take place in two pipelined shift operations. First, the information required for partitioning would be shifted, and then, after this was finished, all remaining information would be moved. For RAs that are already active, partitioning would be effectively free. For RAs that are new, the upper bound for storage management would involve the non-active context size only. Both these results would be very important for execution efficiency.

Implementation should be straightforward. A new process will be required to complete storage management by shifting the balance of executing contexts within the lcell array while the lcell manager proceeds with partitioning. In addition, execution must wait until the new process signals that the complete context required for execution has arrived. This extension can be made in conjunction with modifications to support shifting variable context sizes.

6.2.7. Multiple Input Ports

Although we have used a single port located above the tree root for LPL program input, there is no reason why multiple ports distributed throughout the tree might not be used in a similar fashion.

As is done now, top of area node managers that wish to request an LPL program would have their requests sent up by tcell managers, which merge these requests with those from underneath. Such requests could be served by the first tcell they reach that includes an io port. Requests for LPL programs would only reach the top of the tree if no io port exists between the top of the tree and the node in which pruning for a newly-discovered RA starts. An io port would act as a filter for input broadcast from above; only LPL programs not broadcast from the local port need be passed down.

Questions concerning the optimal placement of io ports should be investigated. They should be located high enough to catch a reasonable number of requests (hence, locating them at height 3 is probably not a good idea), and to provide balanced operation. Simulation seems a good way to initially investigate this problem, and the current design would require few changes to implement the approach suggested above.

It would be also possible to provide more generalized routing through the tree, so an io port could serve areas whose top of area is located higher than the port. This would require developmental work to determine an appropriate routing protocol.

CHAPTER 7

Conclusion

Our main goal in this dissertation has been the presentation of the LPL architecture and the DOT implementation model. These are the essential components of an efficient and highly parallel programming system designed to execute FFP languages. A complete programming system was presented, including a variety of LPL definitions for useful FFP operators.

To express the DOT design, we used a concurrent programming language with support for process-oriented simulation. The result is a model of the top-level implementation characteristics of a multiprocessor capable of efficiently supporting LPL and FFP. Aspects related to the simulation approach for multiprocessor design in general, and for DOT in particular, were discussed and initial results of simulation were given.

An analytic model for the progress of reductions on the programming system was derived. This model is based on the actual data manipulation and message transfer protocols embodied in the DOT design, and it accurately predicts upper and lower bounds for RAs.

Alternative approaches to the design were discussed, and in some cases these were analyzed with the help of the analytic model. In general, comparison of alternatives will require use of different simulation models, and we have indicated fruitful directions of approach within the context of the current DOT model. Important extensions and directions for future development work have also been indicated.

A natural question to ask at this point is whether the DOT design representation of the desired multiprocessor implementation can be directly mapped into VLSI. For example, could the DOT implementation of the lcell and tcell classes be given to a silicon compiler? Unfortunately, the answer is no. For such an approach to be effective, a lower level of implementation should probably be used, such as the circuit level or the register transfer level. Attempting to map DOT processes directly into dedicated hardware is likely to be wasteful of circuit logic.

To produce efficient realizations for the lcells and tcells, the behavior of their processes in time must be carefully examined. The first thing to be noticed is that DOT processes spend most of their time waiting for the arrival of information that is required for a subsequent operation. The tcell manager, for instance, is essentially idle throughout the entire execution phase of the machine cycle, waiting for the stop message which originates in the io subsystem. Because of this, dedicating an entire block of VLSI circuitry to the tcell manager, including logic for all required arithmetic manipulation, would be wasteful. Logic required for arithmetic operations should probably be shared between the node manager and the tcell manager. The same considerations will apply to the lcell as well; the LPL interpreter process should share an arithmetic unit with the lcell manager.

In addition to examining the characteristics of the cell processes with the aim of efficient processor utilization, usage of logical communication channels over time should be investigated carefully in order to achieve efficient utilization of physical intercell communication lines. An intercell communication line might be time-division multiplexed to provide shared use of a single line by multiple logical channels, but, if so, it is likely that the appropriate balance of communication scheduling will differ dynamically, as cells move through the execution cycle. For instance, the communication bandwidth available on the cell manager channels during the execution phase could be minimal.

In our design approach, we were motivated by the desire to borrow techniques useful in the design of parallel software, and apply them within the larger context of multiprocessor design. The results have been successful. DOT is the first complete implementation model based on Mago's design goals. Although the overall system is quite complex in its operation, the use of small, simple, sequential processes for its description produced an intellectually manageable design.

To make the transition from the DOT implementation model to actual realization, however, requires that processes and logical channels which were carefully separated in our high-level design, for the purpose of clarity, be efficiently integrated into their respective cellular components. Our design does not provide guidance for this; the tasks to be performed within the tcells and lcells have been identified, but the allocation of tasks to specific hardware and firmware remains an open issue.

These low-level considerations were outside the scope of this dissertation, but they ultimately need to be addressed in order to produce a hardware realization. As indicated in Chapter 6, many design alternatives still need to be investigated, and many improvements to the present design are possible. Although a final decision on realization must wait for further resolution of these higher-level alternatives, general investigation of important issues related to realization, such as those mentioned above, is indicated. In addition to providing advance insight into problems to be expected in a realization, such research may indicate a more fruitful high-level design methodology than the one we have used, when the desire for straightforward realization is taken into account.

REFERENCES

[Abr70] P. Abrams, "An APL Machine," SLAC Report No. 114, Stanford Linear Accelerator Center, Ph.D. dissertation, Stanford University, 1970.

[Ack82] W. Ackerman, "Data Flow Languages," Computer, Vol 15, #2, pp. 15-25, February 1982.

[Agg82] S. Aggarwal, "Flexibility of Computer Network Simulation Using the Hierarchical Class Concept," Proceedings of the Tenth IMACS World Congress on System Simulation and Scientific Computation, pp. 207-209, 1982.

[And81] G. Andrews, "Synchronizing Resources," ACM Transactions on Programming Languages, Vol. 3, #4, pp. 405-430, October 1981.

[Bac72] J. Backus, "Reduction Languages and Variable-free Programming," IBM Research Report RJ1010, Yorktown Heights, NY, April 1972.

[Bac73] J. Backus, "Programming Language Semantics and Closed Applicative Languages," IBM Research Report RJ1245, Yorktown Heights, New York, July 1973.

[Bac78] J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," CACM Vol. 21, #8, pp. 613-641, August 1978.

[Bac81] J. Backus, "Is Computer Science Based on the Wrong Fundamental Concept of Program? An Extended Concept," in Algorithmic Languages, Bakker/Vliet (eds.), IFIP, North-Holland Publishing Co., pp. 133-165, 1981.

[Bac82] J. Backus, "Function Level Computing," IEEE Spectrum, Vol. 19, #8, pp. 22-27, 1982.

[Ber75] K. Berkling, "Reduction Languages for Reduction Machines," Second Annual Symposium on Computer Architecture, pp. 133-138, 1975.

[Bla83] G. Blaauw and F. Brooks, Computer Architecture, in preparation, 1983 draft.

[Bri77] P. Brinch-Hansen, The Architecture of Concurrent Programs, Prentice Hall, 1977.

[Bur78] H. Burkle, A. Frick and Ch. Schlier, "High Level Language Oriented Hardware and the Post-Von Neumann Era," Fifth Annual Symposium on Computer Architecture, ACM-SIGARCH Newsletter Vol. 6, #7, pp. 60-65, April 1978.

[Car81] W. Carlson, "The Pascal Microengine", Workshop on High-Level Language Computer Architecture, Los Angeles, Ca., 1981.

[Chu81] Y. Chu, "Design of Jovial Direct-execution Architectures," Workshop on High-Level Language Computer Architecture, Los Angeles, Ca., 1981.

[Cla80] T. Clarke, et. al, "SKIM -- The S, K, and I Reduction Machine," Proceedings of the LISP-80 Conference, pp. 128-135, August, 1980.

[Dav75] A. Davis, "The Architecture and System Method of DDM-1: A Recursively-structured Data Driven Machine," Proceedings of the Fifth Annual Symposium on Computer Architecture, 1978.

[Del83] P. Delesalle, "Computer Architecture Taxonomy," Tenth Annual Symposium on Computer Architecture, 1983.

[Den79] J. Dennis, "The Varieties of Data Flow Computers," Proceedings of the First International Conference on Distributed Computing Systems, pp. 430-439, October 1979.

[Des78] A. Despain, "X-Tree: A Tree Structured Multiprocessor Architecture," Fifth Annual IEEE Symposium on Computer Architecture, 1978.

[Fra77] W. Franta, <u>The Process View of Simulation,</u> North Holland, 1977.

[Fri76] D. Friedman and D. Wise, "CONS should not evaluate its arguments," in Michaelson and Milner (eds.), *Automata, Languages, and Programming*, Edinburgh Press, pp. 257-284, 1976.

[Fuc82] H. Fuchs, J. Poulton, A. Paeth, and A. Bell, "Developing Pixels-Planes, A Smart Memory-Based Raster Graphics System," 1982 Conference on Advanced Research in VLSI, MIT, pp. 137-146, January 1982.

[Hoa78] C. Hoare, "Communicating Sequential Processes," Communications of the ACM, pp. 666-677, August, 1978.

[Ich79] J. Ichbiah, et. al, "Ada Reference Manual," ACM SIGPLAN Notices, Vol 14, #6, June, 1979.

[Keh78] D. Kehs, "A Routing Network for a Machine to Execute Reduction Languages," Ph.D. dissertation, University of North Carolina at Chapel Hill, 1978.

[Kel79] R. Keller, G. Lindstrom, and S. Patil, "A Loosely-coupled Applicative Multi-processing System," AFIPS Conference Proceedings Vol. 48, pp. 613-622, 1979.

[Ker78] B. Kernighan, and D. Ritchie, The C Programming Language, Prentice-Hall, 1978.

[Klu82] W. Kluge, "A Concept for Cooperating Reduction Machines," Proceedings of the International Workshop on High-level Language Computer Architecture, pp.170-180, 1982.

[Kos77] A. Koster, "Execution Time and Storage Requirements of Reduction Language Programs on a Reduction Machine," Ph.D. dissertation, University of North Carolina at Chapel Hill, 1977.

[Leu76] C. Leung, D. Misunas, A. Neczwid and J. Dennis, "A Computer Simulation Facility for Packet Communication Architecture," Proceedings of the Third Annual Symposium on Computer Architecture, pp. 58-63, January, 1976.

[Mag79] G. Mago, "A Network of Microprocessors to Execute Reduction Languages," International Journal of Computer and Information Science, Vol. 8, Nos. 5,6, pp. 349-385 and 435-471, 1979.

[Mag80] G. Mago, "A Cellular Computer Architecture for Functional Programming," Digest of papers, IEEE Computer Society COMPCON, pp. 179-187, Spring 1980.

[Mag83] G. Mago, D. Stanat, and A. Koster, "Program Execution in a Cellular Computer: Some Matrix Algorithms," in preparation.

[Man74] Z. Manna, Mathematical Theory of Computation, McGraw Hill, 1974.

[Mar80] F. Maryanski, Digital Computer Simulation, Hayden, 1980.

[Par82] R. Pargas, "Parallel Solution of Elliptic Partial Differential Equations on a Tree Machine," Ph.D. dissertation, University of North Carolina at Chapel Hill, 1982.

[Pre81] H. Presnell, and R. Pargas, "Communication along Shortest Paths in a Tree Machine," Proceedings 1981 Conference on Functional Languages and Computer Architecture, pp. 107-114, October 1981.

[Ric71] R. Rice and W. Smith, "Symbol -- A Major Departure from Classic Software Dominated von Neumann Computing Systems," AFIPS Conference Proceedings, Vol. 38, SJCC, pp. 575, 1971.

[Sha75] R. Shannon, Systems Simulation -- the art and science, Prentice Hall, 1975.

[Sha82] D. Shaw, "The NON-VON Supercomputer," Technical Report, Department of Computer Science, Columbia University, 1982.

[Shu83] M. Shute, "The Role of Simulation in the Study of Multiprocessor, Control Flow, and Data Flow Systems," Ph.D. dissertation, Westfield College of the University of London, 1983.

[Sid83] W. Siddall, "Virtual Memory Algorithms for Tree-Structured Processors," Ph.D. dissertation in preparation, University of North Carolina at Chapel Hill.

[Sta79] D. Stanat and G. Mago, "Minimizing Maximum Flows in Linear Graphs," Networks, Volume 9, #4, pp. 333-361, 1979.

[Sta81] D. Stanat and E. Williams, "Optimal Associative Searching on a Cellular Computer," Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, pp. 99-106, October, 1981.

[Sta81a] D. Stanat and G. Mago, "Optimal Storage Management in a Cellular Computer," Technical Report 81-006, Department of Computer Science, University of North Carolina at Chapel Hill, 1981.

[Ste81] G. Steele and G. Sussman, "Design of a LISP-based microprocessor," CACM Vol. 23, #11, pp. 628-645, 1981.

[Sto83] S. Stolfo, "Architecture and Applications of DADO: A Large-scale Parallel Computer for Artificial Intelligence," Technical Report, Department of Computer Science, Columbia University, 1983.

[Str82] B. Stroustrup, "Adding Classes to the C Language: An Exercise in Language Evolution," to appear in Software: Practice and Experience.

[Ten82] A. Tanenbaum, Computer Networks, Prentice Hall, 1981.

[Tol81] D. Tolle, "Coordination of Computation in a Binary Tree of Processors: An Architectural Proposal," Ph.D. dissertation, University of North Carolina at Chapel Hill, 1981.

[Tre80] P. Treleaven and G. Mole, "A Multi-processor Reduction Machine for User-defined Reduction Languages," Proceedings Seventh Annual Symposium on Computer Architecture, pp. 121-130, May 1980.

[Tre82] P. Treleaven, D. Brownbridge and R. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," ACM Computing Surveys, Vol. 14, #1, pp. 93-143, March 1982.

[Tur79] D. Turner, "A New Implementation Technique for Applicative Languages," Software Practice and Experience, Vol. 9, pp. 31-49, September, 1979.

[Wil81] E. Williams, "Analysis of FFP Algorithms for Associative Searching," Ph.D. dissertation, University of North Carolina at Chapel Hill, 1981.