Alternative Program Representations in the FFP Machine

TR86-027

1986

David J. Middleton

The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27599-3175

Alternative program representations in the FFP machine

by

David John Middleton

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, 1986

Approved by:

Gyula Magó

Advisor: Gyula Magó

[illegible]

Reader: Frederick P. Brooks, Jr.

[illegible]

Reader: Richard Snodgrass

DAVID JOHN MIDDLETON
Alternative program representations in the FFP machine.
(Under the direction of GYULA A. MAGÓ.)

Abstract

Program representation seems to be an important factor in enabling parallel computers to provide the high performance they promise. Program representation, from that of the initial algorithm to that of the run-time machine code, must provide enough information that parallelism can be detected without imposing further constraints that prevent this parallelism from being exploited. The thesis of this research is that the efficiency of the FFP machine is significantly affected by the choice for program representation. The results support the conjecture above and suggest that greater emphasis should be placed on program representation in the design of parallel computers.

This research examines run-time program representation for the FFP machine, a fine-grained language-directed parallel computer which uses a string-reduction model of computation. Several alternative program representations are proposed, and the resulting machine variants are compared.

Four characteristics are then developed which accurately predict the advantages and problems that would arise with other forms of program representation. These characteristics can be derived from the basic nature of the FFP machine and its model of computation, so these characteristics may well be important in the design of other parallel computers that share the basic character of the FFP machine.

The results are organised into a procedure for choosing a representation, based on the expected use of a particular instance of the FFP machine. A definition of processor granularity is proposed based on software function instead of hardware costs.

Acknowledgements

I joyfully acknowledge the love, dedication and effort that my Father and Mother spent in raising our family. I dedicate this dissertation to them. I want to thank my wife, Michele, for her support and encouragement through the difficult times. I am also grateful to the members of my Honours year, who made the study of Computer Science especially enjoyable: Debbie Blatch, Frank Carnovale, Paul Lorger, David Powers, John Reeves, Ros Riley, Michael Selig and Kerry Stewart.

I have been extremely fortunate to work with Gyula Magó. He has demonstrated a degree of scholarship which I hope to emulate and is very considerate to the students under his direction. Frederick Brooks has taught me a great deal about computer architecture and the framework underlying scientific exposition. Richard Snodgrass gave generously of his time and ideas in helping me to express the results of this research more comprehensibly. I would also like to thank Don Stanat and Dean Brock for many interesting discussions and classes.

I would like to thank John Zimmerman for many enjoyable discussions on the nature of Science in general, and Computer Science in particular. I am grateful to Bruce Smith for the opportunity to work on a wider range of projects. I would also like to thank the newcomers to the project, Ed Biagioni, Lakshmi Dasari, Bill Gibson and Will Partain, who have took part in several interesting discussions and pointed out at least one unnoticed error in this research.

I am grateful for the availability of the tools, TEX and MacDraw, which, as well as formatting this document, provided many challenging hours of thought on the nature of text processing.

Table Of Contents

Table of Figures

Glossary

absolute level number, ALN	50
application parentheses	20
areas	29
attachable	64, 105
braces	47
capability	113
clones	25
compact layout	69
compressed representation	69
cumulative message operation	88
denotation	49
directory	21
Empties	98
EndOfStream	28
execution phase	32
expression tree	20
filter	25
Fine granularity	114
fine-grained	66
fingers	12
firstL	21
fork	25
graph reduction	3
idioms	35
index	21
L array	29
L cells	6, 29
lastL	21
layout	68
Leftscraps	98
manipulation	49
mergeleft, mergeright	99
message processors	21
message wave	25

Chapter 1

Program representation in parallel computers

1.1 The importance of program representation

Parallel computers are intended to perform computations more quickly by using many processors at the same time, each one applied to a different part of the computation. However, several issues may arise to prevent parallel computers from fulfilling this promise. One issue concerns the ways applications can use the parallel computer once the hardware has been constructed. The complexities of operations such as program decomposition, task allocation and task scheduling, create problems by making the process of programming parallel computers difficult and very sensitive to minor aspects of a particular machine design. As a response to this difficulty, language-directed design is appearing with increasing frequency as a principal foundation of general-purpose parallel computers. Such computers have been designed around high level languages that are based on abstract models rather than concerns about implementation on traditional computers [Clarke *et al.* 80, Glaser *et al.* 84, Kluge 83, Agerwala *et al.* 82]. These systems require much less transformation of the programs before execution because the hardware closely follows the model of computation of the chosen language.

A second issue in the design of general-purpose parallel computers is how programs should be represented. According to Flynn and Hennessy, "The initial representation profoundly influences the ability [of sequential computers] to obtain a good representation at other levels [of execution]. . . . The representation problem is worsened with the advent of distributed computing" [Flynn and Hennessy 79]. They further suggest that the choice for program representation is important in all general-purpose parallel computers, in order that available parallelism can be detected, without constraining machine operation to the point where it cannot exploit that parallelism. It appears that general-purpose parallel computers may be particularly sensitive to design choices related to program representation (both in comparison with other factors in the design of general-purpose parallel computers and with the sensitivity of general-purpose sequential computer design to program representation).

As a first step in examining this conjecture, the thesis of this research is that the design of the FFP machine is very sensitive to decisions about the run-time representation of programs. That the FFP machine is a language-directed design is important for examining the conjecture because design decisions about program representations in such computers are made explicitly, and earlier than, other decisions such as the network topology or operations which might constrain them in unexpected ways. Several variants of the FFP machine are studied that differ in the way programs are represented. That minor variations in program representation cause large differences in performance for the variant FFP machines is evidence supporting the conjecture [Middleton 85]. The causes of these differences are explained in terms of characteristics derived from the fundamental philosophy of the FFP machine design. To the extent that a different parallel computer design shares this philosophy, that design may be expected to be sensitive to program representation in similar ways.

1.2 Representing FFP programs in the FFP machine

The FFP machine [Magó 79, Magó 80], is specifically designed to execute Backus's Functional Programming languages [Backus 78, Backus 73]. (Formal Functional Programming languages, or FFPs, differ from the Functional Programming languages, or FPs, in providing more powerful facilities for manipulating program expressions, but the important difference for the FFP machine is that FFP provides a uniform syntax which simplifies automatic analysis). FP languages have a number of properties which are attractive for parallel computer design. They lack a global state, that is variables that can be accessed from any part of the program. Since there is no need to support such a state, the FFP machine need not provide multiple access paths from its many processors to the entire memory (either global or distributed). FP languages use an evaluation rule that is simple to support; the definition of available computations (ones which are not waiting for other computations to complete) allows the machine to find and begin evaluating them rapidly. In comparison with the basic operations in standard imperative languages, those of FP provide a high level of discourse and a strong modularity because of the lack of variables.

The FFP machine evaluates FFP programs using a string reduction model of computation. Descriptions of different models of computation can be found in surveys by Treleaven, Hopkins and Brownbridge [Treleaven *et al.* 82], and by Vegdahl [Vegdahl 84]. In *string reduction*, a program consisting of both function and data is represented as a string of symbols. Execution involves repeatedly replacing substrings of the program with other strings that have the same meaning. The value of each program subexpression is independent of other program subexpressions (in most such models). In contrast, ***graph reduction*** allows subexpressions in the program to be shared (via variables). This sharing avoids the need for recomputation but requires the hardware to provide more general memory access. The FFP machine holds the symbols of program strings in a large linearly-ordered array of fine-grained processors, groups of which cooperate in replacing appropriate substrings.

One unique aspect of the FFP machine design is that it dynamically maps the hardware onto running software. Not only is the hardware formulated at design time to reflect the properties of a language (that is, the set of all possible programs), but the hardware is also configured during execution to match the properties of individual programs. A separate dedicated set of processors is assigned dynamically during execution to each available computation in the FFP machine. In this way, the problem of program decomposition has been replaced with the problem of hardware decomposition, that is, dividing the hardware resources into groups that match programs' characteristics, rather than dividing programs into tasks that match processor characteristics. Under this philosophy of mapping hardware onto software, decisions about program representation, while still very important for the FFP machine, may differ from those for other language-directed computer designs.

1.3 Organisation of this dissertation

This dissertation is a systematic study of program representations that could be used in the FFP machine. This research investigated how the representation of programs in the FFP machine affects the requirements upon, and the abilities of, the hardware to support the rapid evaluation of FFP programs. That is, the program representation dictates how the hardware must represent intermediate expressions during execution and what information is available to the processors to direct their operation. The different representations are created by altering the L cell's set of symbol registers, those registers that hold the part of an FFP expression residing in a given L cell.

The results are organised as a procedure for evaluating the consequences of various choices for program representation, rather than as a single choice of the best representation. The design of the FFP machine must try to satisfy several criteria and the merits of each representation vary according to the relative importance of the criteria. Selecting the best representation requires a knowledge of the relative effects of these criteria on the final speed and complexity of the machine, a knowledge which is not yet sufficiently available. More importantly, as the FFP machine is extended to support other languages and facilities (such as logic programming or outermost evaluation), the factors affecting the choices will change. A catalog of consequences encompassing these different goals will better serve this development of future FFP machines. Furthermore, such a catalog is less sensitive to the problem of representations being discarded because possible methods have not yet been discovered to mitigate their disadvantages (such as increasing the information or mechanisms available within the L cells). A side product of this research has been insight into the differences between fine and coarse granularity in parallel processors, from the point of view of aspects of computation rather than hardware.

The considerations for program representation made in other parallel computer designs are examined in Chapter 2. It is apparent that questions of program representation have had little impact on the design decisions in most of these machines.

A generic FFP machine presented in Chapter 3 demonstrates the model of computation and provides a standard for comparing the various FFP machines that result from different choices for program representation.

A benchmark is developed in Chapter 4 which will measure the usefulness of a representation in terms of how easily the corresponding FFP machine supports operations necessary to the model of computation. Especially since the FFP machine is a language-directed design, the benchmark is related to aspects of implementing the FFP language (instead of, for example, implementing specific applications). Each representation is evaluated with respect to its effects on the time, space and hardware complexity costs incurred by the different elements of the benchmark.

The next step is to generate candidate representations for evaluation. Several are presented in Chapter 5 that exemplify the range of consequences discovered during this research. This research on alternative program representations was motivated by the need for a program representation that expressed closing sequence brackets and application parentheses explicitly during execution. The consequent increase in the number of L cells needed to hold programs led to the search for representations that use fewer L cells by storing more FFP symbols in each one. From the behavior of the example templates, four constraints are developed as being necessary for a useful candidate representation. These constraints can be derived from properties of the FFP machine which are considered essential: that a program should be directly expressed as a string of symbols with each processor holding a small part of the complete expression. These constraints and motivation delimit and fill in the set of candidate representations to be examined.

Some consequences of the representations considered here are examined in detail. Chapter 6 describes the consequences of the fact that the representations of FFP programs may no longer be unique. Chapter 7 examines the effects of changes in program representation on the low-level hardware operations, those that belong more to the machine itself than the FFP language.

The results of the research are collected in Chapter 8. The steps are described by which a particular program representation would be chosen and the consequences of such choices are listed. A catalog containing some specific templates demonstrates the interaction of various design choices concerning program representation and shows the relative effects of these choices on the different measures in the benchmark. The balance between fine and course granularity in processing elements is discussed with regard to the needs of implementing functions, instead of hardware factors, as is more common.

Fonts are used to emphasize specific meaning in terms: the *italic font* introduces definitions which are listed in the glossary; the *slanted font* is used for FFP functions and for system operations and variables used in the FFP machine; and the **bold font** indicates FFP expressions as they are held in the FFP machine during execution.

Chapter 2

Related work: program representation in other parallel computers

One aspect of designing any computer involves choosing the form in which abstract programs are to be represented internally for the hardware to interpret. Because of the severe difficulties with programming most parallel computers that have been built so far, this choice seems especially important in such machines. Designing parallel computers involves more numerous and diverse design choices than designing sequential computers. Therefore, program representations that may be acceptable for sequential machines may not satisfy these more stringent demands. This is borne out by problems such as synchronising and scheduling tasks which arise when sequential languages are adapted to parallel computers.

This review will examine choices for program representation in parallel computers. We begin with previous versions of the FFP machine, and then examine other parallel computer designs that share some of the FFP machine's characteristics, those being language-directed or fine-grained parallel computers.

2.1 Previous versions of the FFP machine

A generic FFP machine, which is described in Chapter 3, is summarised here in order to contrast different versions of the FFP machine. The FFP machine is a tree structured network in which the leaves are simple programmable processors [Magó *et al.* 84]. The internal nodes contain communication processors which can perform a small amount of processing upon messages that pass through them. An FFP program consists of a function and an operand and is represented as a string of symbols stored in the linear array of leaf processors, called *L cells*. For parallel languages in general, an available computation is one that does not need results from other computations; it may be evaluated immediately providing the parallel computer has an idle processor. Each available computation in an FFP program is completely expressed by a contiguous substring of symbols called a ***reducible application***, or ***RA***. The L cells that hold an RA, together with some of the communication resources inside the tree network, are configured as an isolated subnetwork dedicated to evaluating that RA. The locality of the RA, that is, the confinement of all the information needed by an available computation to a contiguous region of the FFP machine, enables the FFP machine to prevent contention for communication resources arising between different subnetworks. These subnetworks are also tree structures and, as such, do suffer from a communication bottleneck at their own roots when their RA requires many messages to be sent among their own L cells. An RA is reduced to its result by the combination of the actions of the L cells in a subnetwork holding the RA, each working independently with its own local information.

2.1.1 The initial proposal

In Magó's proposal, the original design from which the other versions have evolved, L cells can hold one symbol of the FFP expression [Magó 79, Magó 80]. His representation differs from the natural string representation of an abstract FFP program in that some syntactic symbols, the closing sequence brackets and application parentheses, are removed from the FFP program's run-time representation in order to save space. In order to compensate for their absence, extra information is stored in the L cells which describes the program structure in a different way.

2.1.2 A virtual computer surrounding the FFP machine

Frank [Frank 79] and Siddall [Frank, Siddall and Stanat 84] have investigated schemes for embedding the FFP machine in a virtual computer that can execute programs larger than the FFP machine can hold. Many of these alternatives involved moving subexpressions of the program to secondary storage and leaving pointers in their place.

The only change to the representation of FFP programs is the addition of these pointers; otherwise the FFP machine design is the same. The pointers add extra testing to the duties of the L cells but since they are programmable, this is a negligible change. The pointers also require communication paths from all of the L cells to the secondary storage, a facility which interferes with the independence that the individual networks had enjoyed.

2.1.3 Different communication networks

Kellman proposed an FFP machine in which the binary tree of communication nodes is replaced with a sorting network [Kellman 83]. This scheme does not exploit the opportunity for locality which RAs provide: an RA no longer has a dedicated network isolated from those of the other RAs. Instead, all the symbols of the FFP program reside in a single collection of processors and the sorting network repeatedly shuffles the L cells to allow them communicate through nearest neighbor connections. Kellman's machine can accomplish storage allocation and the permutation of symbols within an RA for FFP operations such as *transpose* in asymptotically logarithmic time, in contrast to the asymptotically linear time that the FFP machine with a tree network requires. The communication network of the FFP machine performs simple message processing which is expensive, however, if performed by the L cells exchanging messages. Such message processing is not performed by the sorting network that Kellman proposed, but rather by moving the L cells around so that they can communicate via the neighbor to neighbor connections. While still asymptotically logarithmic in cost, this mechanism is significantly slower than the FFP machine's use of dedicated, circuit switched, hardware resources. Kellman estimated that, in comparison with Magó's original FFP machine [Magó 79], the crossover point where the asymptotic behavior of his design overcomes its initial complexity should occur in RAs containing operands of around one thousand elements. More recent versions of the FFP machine are significantly simpler and faster [Magó *et al.* 84], and this increases the size of the operand necessary to make Kellman's machine the more effective one.

Plaisted has examined the use of two other sorting networks in supporting certain particular mechanisms of the FFP machine: a Benes network and a delta network [Plaisted 84a, Plaisted 84b]. The rapid insertion and copying of data provided by these richer networks are particularly necessary in term-rewriting systems, which emphasize copying expressions more heavily than does evaluating general FFP programs.

2.1.4 Tolle's version of the FFP machine

Tolle proposed a machine in which both types of nodes in the tree network are more complex and where the majority of computation has shifted to the internal nodes leaving the leaf nodes more like smart memory cells [Tolle 81]. His L cells are able to hold a number of symbols of the FFP program within certain constraints and the closing syntactic markers are not deleted from his representation [Tolle 81, pp. 18-22]. His model for the evaluation of RAs centers on a tree representation rather than a string representation. Significant complexity results because these trees representing the RAs are independent of the communication trees in the FFP machine; the tree expression for an RA must be augmented with auxiliary nodes in order that it can be mapped into the hardware binary tree. The FFP machines presented in this research are similar to Tolle's design in placing several FFP symbols in each L cell. However, they differ quite markedly in using the string representation as opposed to the tree representation during execution; the cells of these FFP machines remain significantly simpler than Tolle's.

2.2 Other language-directed parallel computers

Language-directed parallel computers can be classified as reduction machines, dataflow machines or control-flow machines [Treleaven *et al.* 82, Vegdahl 84]. Reduction machines, which include the FFP machine, take some representation for both the program and the data, and repeatedly replace pieces of this expression with equivalent pieces that are closer to the final result. Graph reduction represents a program as a graph of values and operations and evaluation consists of replacing suitable subgraphs. Graph reduction allows for the value of an expression to be used by several different parts of the program. String reduction represents the program as a string of symbols and evaluates it by continually replacing substrings. The result of a computation may not be shared without explicit copying. (The FFP machine nevertheless supports such mechanisms for sharing anyway [Magó 82]).

Dataflow computers model computation as a graph in which the nodes are operations. Execution involves transferring values, called tokens, along the directed arcs connecting the nodes of the graph. The dataflow graph remains basically unchanged during execution, which distinguishes the dataflow model of computation from that of graph reduction, where execution is expressed in terms of modifications to the graph.

Control-flow parallel computers execute conventional sequential programs in individual nodes, and achieve parallelism by using many such nodes which communicate by passing messages, often through a globally shared memory.

2.2.1 SECD machines

The SECD machine is an abstract model proposed by Landin for describing the reduction of lambda expressions [Landin 64]. The following description has been taken from that of Glaser, Hankin and Till who present a version of the machine that supports lazy evaluation [Glaser, Hankin and Till 84]. Lazy evaluation allows expressions to be evaluated only partially before being used elsewhere in a computation; the expressions are only fully evaluated as specific values become required.

The SECD machine is named for the four major data structures supporting execution: the Stack, the Environment, the Control and the Dump. The Stack holds partial results while an expression is being evaluated, the Environment contains the free variables and their values and the Control holds the program expression. When a nested subexpression is to be evaluated, the Stack, Environment and Control structures are saved in the Dump, to be retrieved after the subcomputation has finished.

The SECD-M machine adds multiprogramming facilities to the lazy SECD machine with the intent of being able to build operating systems in a functional language [Abramsky and Sykes 85]. The extensions consist of a fair merge operation for combining lists and a fair scheduler for executing different processes concurrently. The main change to the machine is a new data structure to hold circular lists of processes. Processes can be created dynamically with a SPLIT instruction and a process may depend on the results from a number of other processes, these results being combined using the fair merge operation. Abramsky and Sykes concentrate on the manipulations to the RQ, the ready queue of processes, necessary to achieve multiprocessing with cooperating processes and to model the behavior of different types of I/O devices. An interesting characteristic of the compiled code is that there are no critical sections to be protected from the scheduler's interruption and it appears that this system should be able to exploit a computer which contained more than one SECD machine. It may happen, however, that extensive sharing of the S, E, C, D and RQ structures seriously inhibits the potential for parallelism. Like the SECD machine, the SECD-M machine shows a strong similarity to the traditional sequential computer and operating system; it has, in fact, been implemented on the Pascal virtual machine microprogrammed on the Perq computer.

2.2.2 The GMD machines

Berkling designed a sequential reduction machine based on the lambda calculus [Berkling 75, Berkling 78]. It has been used as the basis node of a parallel reduction machine proposed by Kluge [Kluge 83].

Berkling's proposal represents the abstract program, a hierarchical expression, as a string of characters representing the preorder scan of the expression tree. These programs are manipulated while moving backwards and forwards across three stacks. (These stacks differ from the S, E, C and D structures of Landin in being much more purely LIFO structures and the machine is more purely reduction based). Kluge develops a model of parallelism in these lambda expressions which involves evaluating subexpressions in a demand driven, or top-down, fashion. Kluge implements this model of computation with a network of Berkling's machines. The network topology is not a fundamental design choice: it is not required to match the hierarchy of the expression structure. Kluge concentrates on questions of task allocation using localised decision processes that must avoid deadlock or starvation while achieving reasonable utilisation of the reduction machines. Little distinction is made between an abstract lambda expression and its concrete representation in the machine; evaluating a primitive application in a reduction machine node of his multiprocessor is an atomic operation that does not use any parallelism.

2.2.3 Combinator machines

Turner describes the use of the combinatory calculus as a model of computation and presents a valuable overview of combinator-based machines [Turner 84]. Applicative languages can be compiled to combinators which form the machine code to be executed. Such a combinator expression will have had all variables replaced by references to the input arguments which are constants. While the use of combinators is most naturally described in terms of string reduction, parallel computer designs based on combinators typically use graph reduction in order that subexpressions, which would be copied frequently in the abstract model, might instead be shared, in order to reduce communication costs.

The SKIM machine was an early implementation of a computer based on a combinator model of computation [Clarke *et al.* 80]. The philosophy of the design is that of a machine, implemented in microcode, which interprets a tree structured combinator expression. The specialisation of the microcode for this machine involves dividing memory into a Head bank and a Tail bank. Parallel versions of the SKIM machine are being investigated [Clark 82, Stoye *et al.* 84].

Hudak and Goldberg introduced the concept of serial combinators [Hudak and Goldberg 85, Goldberg and Hudak 85]. Serial combinators are derived from a particular program as it is being translated into the combinator expression to be reduced. They are the largest combinators available that result in fully lazy evaluation and that have no concurrent substructure. (The combinator definitions are represented as conventional compiled straight-line code). They describe a system that uses heuristics to choose when to distribute expressions to remote processors. Their approach is reminiscent of Kluge's; the heuristics will estimate the comparative costs of distributing the operation as opposed to performing it locally and of accessing a shared value over the communication network as opposed to recomputing it locally. Processing elements are associated with one segment of the global address space and parallelism arises from the available reductions being spread through that memory. A diffusion scheduler distributes expanding parts of the combinator graph evenly to the separate processors.

2.2.4 Wilner's recursive machine

Wilner proposed Recursive Machines based on recursively constructing complex computing systems from simpler ones [Wilner 78, Wilner 80]. A Recursive Machine consists of a single processing element or a network of Recursive Machines connected in a way that provides the same interface. The memory of a Recursive Machine is organised to accommodate the dynamic variation of recursive data structures. Information is stored in fields where a field is either a character or a bracketed sequence of fields. Inserting new information into this memory organisation is simplified by the absence of fixed addresses associated with elements of the data structure. Instead, logical addresses are used which describe fields with respect to the structure given by the punctuation brackets. The memory of a Recursive Machine is distributed through the processing elements as a linear string of storage locations which hold the fields as a string. Each processing element contains pointers called *fingers* that refer to subfields contained in that processor. There are instructions to move the fingers to an adjacent field, to a subfield, or to an outer, enclosing, field. The processing elements can perform serial computations on fields as their fingers scan over them. According to Wilner, "With respect to execution, a Recursive Machine can be thought of as a rewriting system", that is, some fields may become active during execution, and be overwritten with the result of the computation they describe.

There are many similarities between the FFP machine and Recursive Machines, despite the design of the Recursive Machines not having been derived from language considerations. The FFP machine has a similar hierarchical structure, differing in that a second type of node, the internal communication node, has been created for connecting sub-machines. In both machines, a microcode language in the processing elements is used to implement higher level languages. In the FFP machine, the processing elements are smaller, containing less than a kilobyte of memory, principally to store microcode, in contrast to the eight kilobytes of data storage in a Recursive Machine element, and FFP functions are implemented in the FFP machine by several processors running concurrent microprograms. The RAs of the FFP machine exactly correspond to the active fields of the Recursive Machines. Whereas the essence of computation in the Recursive Machines depends on the placement and manipulation of fingers to allow sequential access to fields, addressing in the FFP machine allows random access within the top few levels of the structure of an RA.

2.2.5 The ALICE graph reduction machine

The ALICE machine uses a graph reduction model of computation to support applicative languages [Darlington and Reeve 81]. Each node in a program graph is a packet containing the name of the function to be applied, reference to the arguments and flags to indicate lazy or eager evaluation and the status of the process. Execution involves selecting a packet, determining whether it is reducible and if so, replacing it with other packets according to the rewrite rules of its definition. The hardware implementation involves small groups of agents which can rewrite the reducible packets which are stored in a shared memory; higher performance machines might be constructed by connecting thousands of such groups by a delta network.

2.2.6 AMPS and Rediflow

The Applicative Multiprocessing System consists of about a thousand reasonably powerful processors with parts of the global memory at the leaves of tree of communication and load balancing nodes [Keller, Lindstrom and Patil 79]. It uses a demand driven form of graph reduction to execute programs written in FGL, a derivative of LISP. Each leaf processor maintains a list of available tasks, which correspond to the functions defined by the programmer. The tree network provides access to the full memory for a given task and allows tasks to be moved between the lists to balance the loads of the different processors. The Rediflow computer evolved from the AMPS project [Keller and Lin 84]. The network has changed to a more densely connected one in which each node combines processing, communication and load balancing and contains part of the global memory.

2.2.7 Dataflow machines

A large number of dataflow machines have been proposed [Agerwala and Arvind 82, Srini 86]. The dataflow computation model differs from that of reduction in that the program expression is not modified as the fundamental basis of execution. Dataflow languages minimise the side-effects of program statements, for example, disallowing the value of a variable to be changed, and this allows dataflow programs to be compiled into dataflow graphs. The nodes in these graphs contain the operator to be performed by the node, pointers to the nodes that are to receive a result token and space for storing the input tokens received from other nodes. A node may fire when all its inputs are available and, in firing, provides further inputs to other nodes. A dataflow computer achieves fine grain parallelism by distributing fireable nodes over many processors, provided that the nodes are allocated to processors in a balanced way and that there is sufficient communication between the processors to transfer data packets between nodes. Static dataflow machines use a static graph in which nodes cannot be created during execution, which prevents the use of recursion. Dynamic dataflow machines allow new instances of a node to be created during execution. This requires that tokens indicate which instance of a node is to receive them.

Dennis has proposed a static dataflow computer in which several memory units and several processors communicate through routing networks [Dennis, Boughton, and Leung 80]. The memory units select fireable nodes, called enabled cell blocks, and send them to the processors through a routing network. The processors transfer the resulting tokens back to the memory units, using pointers contained in the cell blocks.

Gurd and Watson have proposed a dynamic dataflow computer in which there may be several instances of each program node and tokens contain a field to identify among these instances. A matching unit stores tokens until they can be matched for the identical instance of a node in the dataflow graph [Gurd, Kirkham and Watson 85].

2.2.8 Traditional multiprocessors

Many parallel computer designs involve a number of reasonably complex processors (with the abilities of a typical microprocessor) which communicate through an $n \times lg\ n$ switching network either to a shared memory or to the other processors. This group includes the Ultracomputer [Gottlieb *et al.* 83], the PASM computer

[Kuehn *et al.* 85] and the Cosmic cube [Seitz 85]. The Cedar system will be presented since it is based on research in extracting parallelism from FORTRAN programs and so is closest to a language-directed design [Gajski *et al.* 83].

The Cedar system follows from the Parafrase project [Kuck *et al.* 81] which analyses FORTRAN programs to find and exploit parallelism. Programs are transformed into a macro-dataflow form: computation at the low-level is expressed in traditional instruction sequences for which efficient hardware is well understood, and, at a high level, such computation blocks are scheduled using a dataflow scheme which expresses parallelism more naturally. The Cedar computer consists of up to 128 Processor Clusters connected to a similar number of global memory modules through an Omega network. A Global Control Unit assigns Computation blocks to the different clusters. Each Cluster consists of eight or sixteen processors and a similar number of cluster memory modules connected through a fast network and controlled by a Cluster Control Unit. Compiler analysis allows objects to be cached at appropriate points in the memory hierarchy. Program and data representation follow from the traditional nature of the processors and memory. A significant problem for such medium or large grained systems is finding sufficient parallelism in programs to achieve acceptable increase in the speed of computation.

2.3 Other fine-grained parallel computers

A number of fine-grained parallel computer designs have recently been proposed that intend to provide more generality than the early fine-grained designs, such as the DAP and the MPP [Reddaway 74, Batcher 82], which were tailored solely for specific applications. An important contributing factor in this generality is whether individual processors can execute their own instructions, that is, whether the parallel computer is SIMD or MIMD. Many of the recent proposals are MIMD extensions of earlier SIMD proposals.

2.3.1 NON-VON

The NON-VON computer consists of a tree network of identical processors executing a common stream of instructions that is broadcast from the Control Processor at the root of the network or from Intelligent Disk Units distributed at some lower level in the tree [Shaw 82]. Each processor contains about 70 bytes and 8 bits of local memory, along with simple processing and communication facilities. NON-VON was derived from a specialised processor intended for supporting parallel database operations. As is usual in SIMD processors, there is an enable flag by which subsets of the processor network may be disabled from executing parts of the instruction sequence. The DADO computer is an extension of the NON-VON containing about a hundred thousand PEs, each with two thousand bytes of local memory [Stolfo and Miranker 84]. (The PEs in the one thousand cell prototype have memories of sixteen thousand bytes to allow for experimentation). Each PE may work in either SIMD mode, receiving procedure calls as its instructions from an ancestor, or in MIMD mode where it stores its program locally and may broadcast instructions to SIMD PEs below it.

The NON-VON philosophy appears to be directly opposite that of the FFP machine in the are of programming. Examples of programs are used to demonstrate how the data structures of a problem may be mapped onto the NON-VON PEs and the algorithm be performed. Different mappings of the same problem show different time/space tradeoffs that are available; programmers are expected to use these examples as guidelines in implementing their own problems. Program representation is not discussed, and data representation is dictated by previously determined hardware questions. Constraints on program layout and alignment are fundamental to the NON-VON computer, whereas they are absent from the FFP machine.

2.3.2 The Connection Machine

The Connection Machine has been derived from algorithms for semantic networks [Hillis 85, Christman 84]. It consists of up to a million PEs executing a single instruction stream, each PE containing about three hundred bits of local memory, sixteen flags and a one-bit ALU. A prototype with 64 thousand PEs has been constructed. There are two communication networks: a two dimensional grid and a boolean N-cube. The boolean cube consists of special-purpose message routers, each serving sixteen or thirty two PEs and handling buffering. Like the NON-VON, there are facilities for voting, for enabling subsets of the whole machine and for selecting one PE from some set of candidates. As with the report on the NON-VON, much of Christman's report on the Connection Machine was devoted to collecting examples of programming techniques, on which other programmers might model their own problems. The layout and alignment of data is again fundamental to programming.

2.4 Conclusions

It appears that there is little to be applied to the choice of program representation in the FFP machine from this study of other parallel computer designs. The sparsity of information about program representation suggests that decisions about program representation were not of primary importance in those designs.

Program representation naturally plays a minor role in SIMD parallel computers since they are hardware implementations of particular algorithms. That is, design choices concerning program representation play no greater role in SIMD parallel computers than they do for sequential computers, which may be explained by viewing SIMD computers as executing a sequential program over a many operands at the same time. Program representation in those machines that evolve from SIMD designs may be expected to be strongly constrained by previous design choices that would have become entrenched in the design history.

Similarly, program representation appears to be of minor importance in the control-flow, or fixed program, parallel computer designs as typified by the Cedar machine. These systems involve a number of sequential programs running concurrently in a number of small traditional processors and cooperating through operating system primitives that support communication and synchronisation among them. As such, these designs will make traditional decisions about the representation of their machine code.

In a similar fashion, the SECD-M and Kluge machines concentrate on achieving parallelism by combining several sequential machines. That the individual PEs happen to use a reduction model of computation is coincidental: the central thrust of their studies is to provide operating system facilities to support multiprocessing.

A principle of the FFP machine design is that the hardware should adapt itself to the running program. Thus, operations that transform programs to match characteristics of a specific multiprocessor, while essential for other designs, are unnecessary in the FFP machine. Such operations include program decomposition, task scheduling, resource management and some aspects of compilation, In particular, each reduction is performed by a dedicated MIMD processor in the FFP machine, in contrast with most parallel computer designs, where each computation is an atomic operation performed by a single processor. Since the value of a reducible expression is another expression, the representation of expressions not only affects the layout of the initial expression and so the availability of the input operand to the processors, it also affects the layout of the result expression and so the intended effect of those processors' operations. For these reasons, the FFP machine, with its distributed style of execution, may be particularly sensitive to decisions about program representation.

Since it appears that there is little to be gained from studying these other designs, this study of program representation in the FFP machine can only be guided by the needs and possibilities of the FFP machine itself. As Kieburtz puts it, "Evaluators have been influenced more strongly by conventional computer architectures than by the language model that they implement. This influence has not been restricted to the idea of using programmed control, but extends to the mode of representation and the use of addressable memory" [Kieburtz 85].

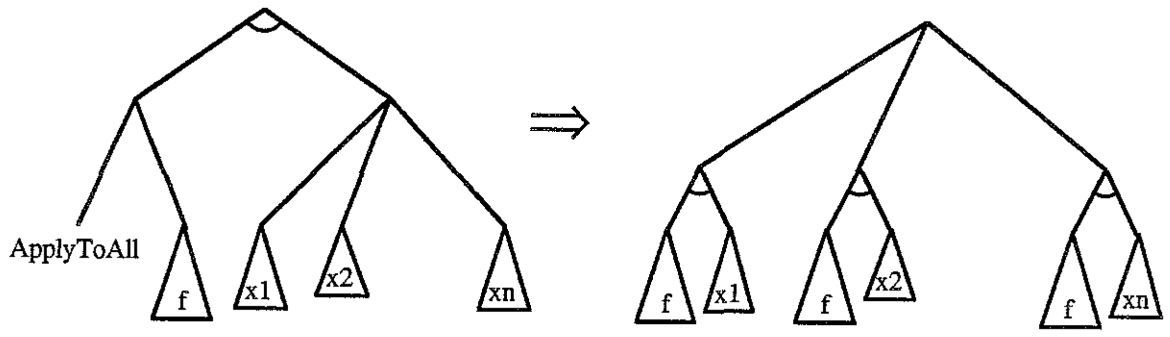
Chapter 3
A generic FFP machine

Different parts taken from several designs that extend back to the original proposal [Magó 79] have been combined to form a generic FFP machine which will serve three purposes. It provides a standard for comparing the different machines which result from different program representations. By demonstrating the FFP machine's style of string reduction, it shows the facilities which implement FFP functions; the variant machines are evaluated by how effectively they support these facilities. It shows the costs of the hardware mechanisms in the FFP machine, indicating which are to be avoided in order to achieve fast simple operation.

3.1 The FFP language and the model of computation

A familiarity with the FFP language is assumed [Backus 78]. The evaluation strategy in FFP is especially important for the FFP machine. Only innermost function applications may be reduced, such ones being called *reducible applications*, or *RA*s. Each RA is a contiguous substring of the complete FFP expression whose value depends only on its symbols and not on its context. It follows that RAs may be evaluated in any order, including simultaneously.

Description of the FFP machine involves three independent entities: the *expression tree description* of an RA around which the manipulations that accomplish the meaning of the function are designed, the *virtual machine* which is created to perform those manipulations and the *physical machine* which comprises the hardware resources that support the virtual machines. It is a distinctive characteristic of the FFP machine that the hardware is continually reconfigured so that the evaluation of each RA is performed by a virtual machine, a tailor-made dedicated parallel processor, supported by an isolated set of processors. This design approach might be called *program-directed* instead of language-directed to emphasise that the virtual machine is not only created at design-time around aspects of a programming language, but is also organised at execution time to match characteristics of one particular program. In contrast, the programs in conventional parallel computers are transformed to match characteristics of the particular PEs.



(<ApplyToAll <Insert +>> <<1 2 3> <4 5> <6 7 8 9> >)

$\Downarrow$

< (<Insert +> <1 2 3>) (<Insert +> <4 5>) (<Insert +> <6 7 8 9>) >

Figure 1. FFP functions are viewed as a manipulation of the expression tree

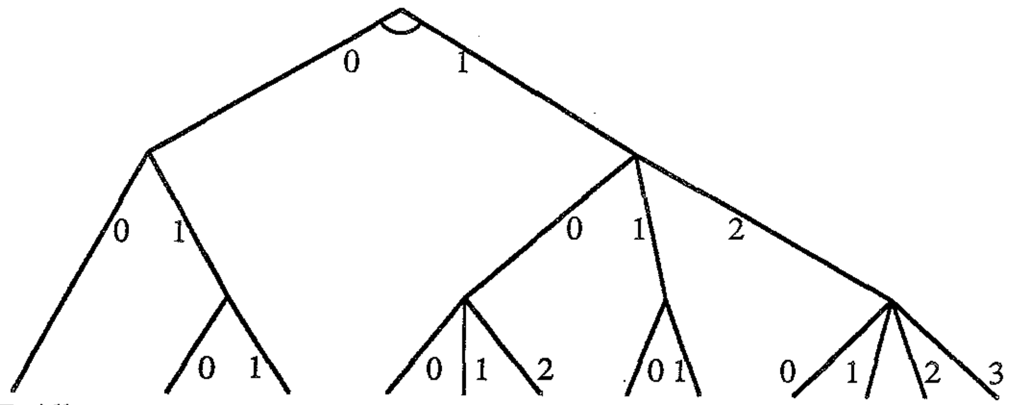
3.2 String reduction of RAs and the virtual machines to perform it

The reduction of an RA is most easily viewed as manipulation of its *expression tree*, an example of which appears in Figure 1. The leaf nodes of the expression tree correspond to the FFP atoms. The internal nodes correspond to sequences and applications, those corresponding to applications being distinguished by an added arc. The *sequence brackets*, "< >", and *application parentheses*, "( )", of the FFP expressions may either be associated with separate leaves in the expression tree, or be attached to the nearby atoms they enclose. For the manipulations implementing a given function, most of the expression tree is viewed as subtrees which are manipulated as single entities.

The *ApplyToAll* function, like the MAPCAR function in LISP, takes an arbitrary function as a parameter and applies it to each element of the sequence constituting the RA's operand. The result is a sequence of RAs which may be evaluated in parallel. Figure 1 shows the expression tree for a general RA using this function and one specific example; the manipulation shown on the expression tree can be seen to yield the result of the example. Implementing *ApplyToAll* on the FFP machine involves duplicating the symbols of the function parameter, *f*, for each member of the sequence, x_i, and altering the sequence brackets and application parentheses to reflect the new structure. String reduction, as performed by the FFP machine, consists of manipulations which may exploit both the contents and the structure of the RA and which are only required to generate well-formed expressions.

A *virtual machine* consists of a linear array of independently programmable processing elements, called *virtual L cells*, which communicate through a network of *message processors* (see Figure 4). The RA is stored from left to right in these L cells as a string of FFP symbols, one symbol in each cell. The L cells create, delete and modify symbols individually, on the basis of their symbols, local information, and information they receive from other L cells. It is by their actions, taken in concert, that the virtual machine replaces an RA with its result. The set of instructions that an L cell performs is grouped into a program called a *microcode segment*; the set of microcode segments corresponding to an FFP function describe the manipulation of the related expression tree in a distributed fashion.

Microcode segments are associated with *sites* in the expression tree, these being subexpressions (subtrees) or points in its structure. Points are the edges of subexpressions, for example, "before [x_2]", or, equivalently, "after [x_1]". For example, *ApplyToAll* would use this information to identify where to insert copies of the function parameter. Each L cell thus needs address information describing the position of its symbol in the expression tree. The address information consists of the *relative level number* (*RLN*), which is the depth of the symbol in the expression tree, the *index*, which is the symbol's position in the RA when the latter is viewed as an unstructured string, the *directory*, and the *firstL* and *lastL* logical tuples. The *directory* describes the position of a symbol in the expression tree by a tuple corresponding to the path from the root of the expression tree to that symbol. For each node in this path, the corresponding element in the tuple is that node's position among its siblings. The *directory* for a symbol is this tuple truncated to some particular length that is defined for the machine; a directory of length four has been sufficient for all the FFP functions investigated so far. The tuples *firstL* and *lastL*, truncated to the same length as the *directory*, identify the first and last L cells holding a given subexpression in the RA, that is, they identify points in the expression tree. For a subexpression of the RA at depth i in the expression tree, *firstL*[i] and *lastL*[i] are true in those L cells which are the first or last cells respectively, containing that subexpression. The *directory* is insufficient for this



	(<ApplyToAll	<Insert	+>>	<<1	2	3>	<4	5>	<6	7	8	9>>)
directory,	'0	0	0	1	1	1	1	1	1	1	1	1'
firstL, :	'0	1	1'	'0	0	0	1	1	2	2	2	2'
and lastL		'0	1'	'0	1	2'	'0	1'	'0	1	2	3'
RLN :	2	3	3	3	3	3	3	3	3	3	3	3
Index :	0,1,2	3,4	5,6,7	8-10	11	12,13	14,15	16,17	18,19	20	21	22-25

Figure 2. Address information associated with symbols in an expression tree

purpose because it is truncated; expressions occurring at a deep enough level in the expression tree will have identical directories for all their symbols.

Figure 2 shows the address information for the symbols of the *ApplyToAll* example, along with the expression tree from which it is (mostly) derived. For example, the symbol 6 has an address (read vertically) containing a *relative level number*, or *RLN*, of 3, an *index* of 19 and a *directory* of (1,2,0). The tuples *firstL* and *lastL* are indicated by quotes attached to the *directory* entries. The address of the left bracket adjacent to the symbol 6 is identical except that its *index* is 18 and the value of *firstL*[2] is true, indicating that it is the first symbol in a subexpression of depth two in the expression tree. The addresses of the symbols "9>>)" differ only in their *indices* and their values for *lastL*. The *indices* range from 22 in the L cell holding 9 to 25 in the L cell holding ')'. *LastL*[0] is true in the L cell holding the right parenthesis, *lastL*[1] is true in the L cell holding the second bracket, *lastL*[2] is true in the L cell holding the first and *lastL*[3] is true in the L cell holding the atom, 9.

The virtual machine that has been created for an RA begins evaluation with each L cell performing a standard prologue to determine the controlling function of the whole RA (*ApplyToAll* in this example) and the address information of its own symbol. The microcode segments, which are stored in an external library and broadcast into the FFP machine on demand, are labeled with an FFP function name and a directory pattern describing the parts of the RA where they should be executed. Each L cell uses its directory and function symbol during the prologue to select and load the appropriate microcode segment.

Figure 3a describes one implementation of the *ApplyToAll* function in a global fashion and shows the corresponding sequence of changes that occur to the symbols of the RA. Figure 3b shows the same algorithm refined to the distributed form performed by the set of microcode segments residing in the separate L cells. For this implementation of *ApplyToAll*, there are two microcode segments: each L cell holding part of the RA's operand executes a copy of one and each L cell holding part of the RA's function executes a copy of the other. The L cells are labeled to indicate which ones perform each of the conditional actions that constitute the microcode segments. For example, in the eighteen cells performing a copy of the second segment, the last condition only holds in the two marked with an 'H'. In general, manipulations of the kind shown in Figure 3a can require the cooperation of several L cells; for example, the cells receiving the symbols of the parameter function need the cooperation of the cells holding that subexpression.

3.3 The L cell instruction set

The microcode language has the usual set of logical, bit manipulating and counting instructions that would be found in conventional microcode languages. The contribution of these instructions to the costs of FFP machine operation is ignored because they are very simple and only play a minor part in the construction and execution of the microcode segments. The important instructions are the ones whose effects extend beyond an individual L cell: the storage allocation and communication instructions. These instructions dominate the costs of time, hardware complexity and microcode storage space in the design of the L cells. This is because they are implemented using many simple instructions and their operation involves the coordination of many separate cells in the FFP machine, L cells for storage allocation and T cells for communication [Danforth 83].

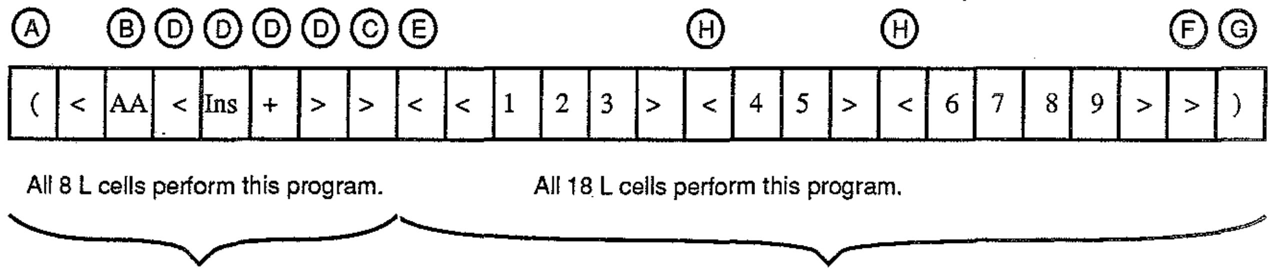
```
(   <   AA   f    >   <   x1        x2     . . .      xn     >    )

             f                x1        x2     . . .      xn

             f                x1     f  x2     . . .   f  xn

    <    (   f                x1  ) ( f  x2 )  . . .  ( f  xn    )    >
```

a) *Erase top level parentheses (1 pair), top level brackets (2 pairs) and ApplyToAll symbol.*

Duplicate expression f before expressions x2 through xn.

Create parentheses (n pairs) and enclose in brackets.

b)



Ⓐ Ⓑ Ⓓ Ⓓ Ⓓ Ⓓ Ⓒ Ⓔ Ⓗ Ⓗ Ⓕ Ⓖ
(< AA < Ins + > > < < 1 2 3 > < 4 5 > < 6 7 8 9 > >)
All 8 L cells perform this program.
All 18 L cells perform this program.

```
Ⓐ If symbol is '('
      erase symbol.
Ⓑ If symbol is 'ApplyToAll'
      symbol := '('.
Ⓒ If symbol is last '>'
      erase symbol.
Ⓓ If symbol is in f
      begin
      determine and broadcast
          the size of f.

      broadcast symbols of f.
      end.
```

```
Ⓔ If symbol is first '<'
      erase symbol.
Ⓕ If symbol is last '>'
      symbol := ')'.
Ⓖ If symbol is ')'
      symbol := '>'.
Ⓗ If symbol is first in xi, ( 2 ≤ i ≤ n )
      begin
      receive size of f.
      create (size+2) new cells.
      place ')' and '(' in the first two.
      receive symbols of f in others.
      end.
```

The notation Ⓐ associates particular statements with those L cells in which the condition holds.
'AA' stands for ApplyToAll and 'Ins' for Insert.

Figure 3. Implementation of the *ApplyToAll* function

3.3.1 Storage allocation

The result of an RA may be larger than the original RA expression, as is the case in the *ApplyToAll* example. Extra virtual L cells must be allocated to a virtual machine in order to hold the FFP text to be created. Since these L cells must participate in creating their new symbol contents, they must contain a suitable microcode segment. By executing the *fork*(n) instruction, a virtual L cell creates n duplicates of itself, both its data and its microcode. The resulting group of virtual L cells, called *clones*, is then available to hold the new symbols, usually receiving them from another part of the virtual machine. The single distinction between these clones, in order that they can behave differently, is the value in the variable *clone-id*, which labels the clones from 0 to n.

3.3.2 Communication and synchronisation

The other microcode instruction whose effects extend beyond the individual L cell is the *SendReceive* instruction. It invokes the facilities of the virtual machine's communication network in a group operation called a *message wave*. The *SendReceive* instruction specifies the message to be sent and a *filter* which is performed for each message that is received back from the network. The filter is a block of microcode instructions, excluding *fork* and *SendReceive*; it extracts and aggregates information from the incoming messages on the basis of their contents and their position in the sequence of messages.

The message processors of the virtual machine network are organised as a binary tree which, while not balanced, is typically of logarithmic depth. The specific binary tree structure of the virtual machine's network does not affect its operation, so that, for example, the two virtual machines shown in Figure 4 are considered identical. Each message processor merges the two streams of messages that it receives from its children into one stream that it sends to its parent. The stream that emerges from the message processor at the root of the virtual machine is broadcast back to all L cells. Upward traveling message streams are merged by sorting messages according to keys they contain. While the most frequent use for the keys is to keep symbols separate during copying operations, they also support the permutations needed to implement FFP functions such as *transpose*. In the case of collisions, messages are combined according to operators they contain, which must be associative since the structure of the tree is unpredictable (and, as will

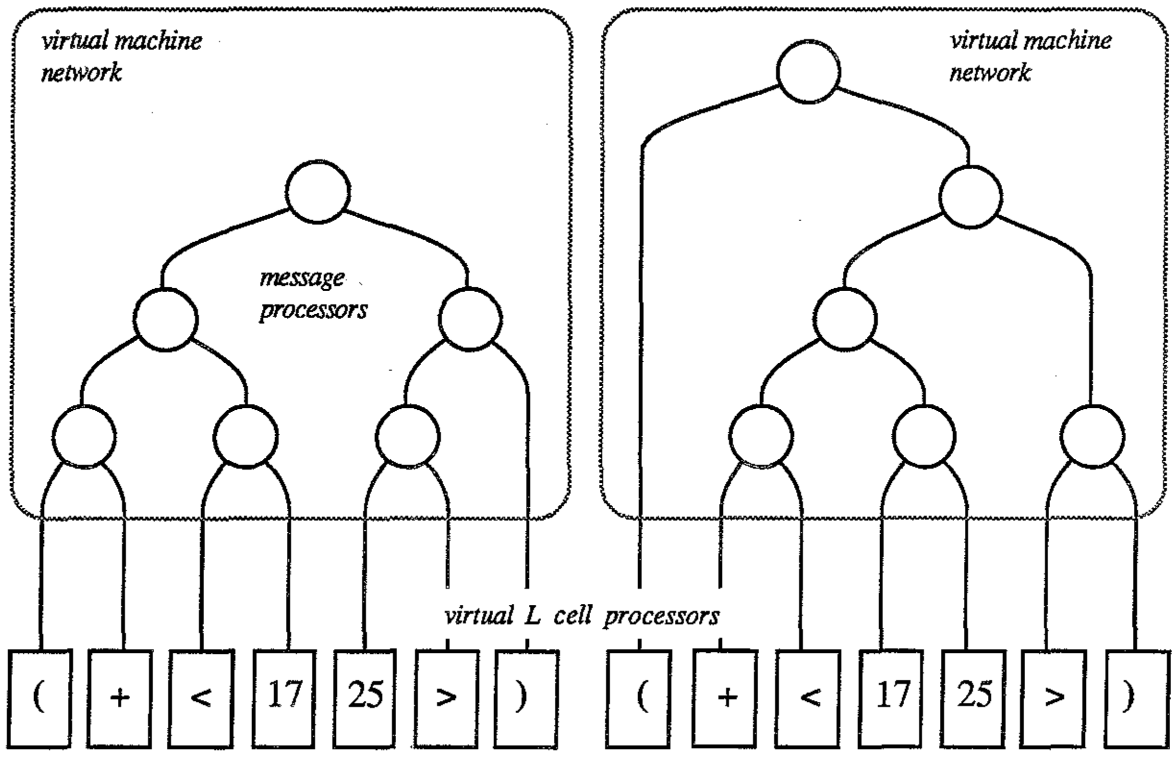


Figure 4. Two equivalent virtual machines

be seen later, can change in midstream during the virtual machine's operation). In practice, the operators *addition* and *maximum* appear to be sufficient.

The *ApplyToAll* example uses two message waves, as shown in Figure 5. The first calculates and broadcasts the size of the function to be copied and the second broadcasts the symbols of the function. In the first message wave, each L cell holding a symbol in the function parameter sends a message with 1 as the value, + as the operator and 0 as the key. The other L cells send no messages. The common key causes the values of the messages to be added together. The single message that is broadcast from the top of the virtual machine contains in its value field the size of the function parameter, measured by the number of L cells needed to hold it. The filter executed by L cells which are to insert the function parameter saves this value from the message for later use by a *fork* instruction.

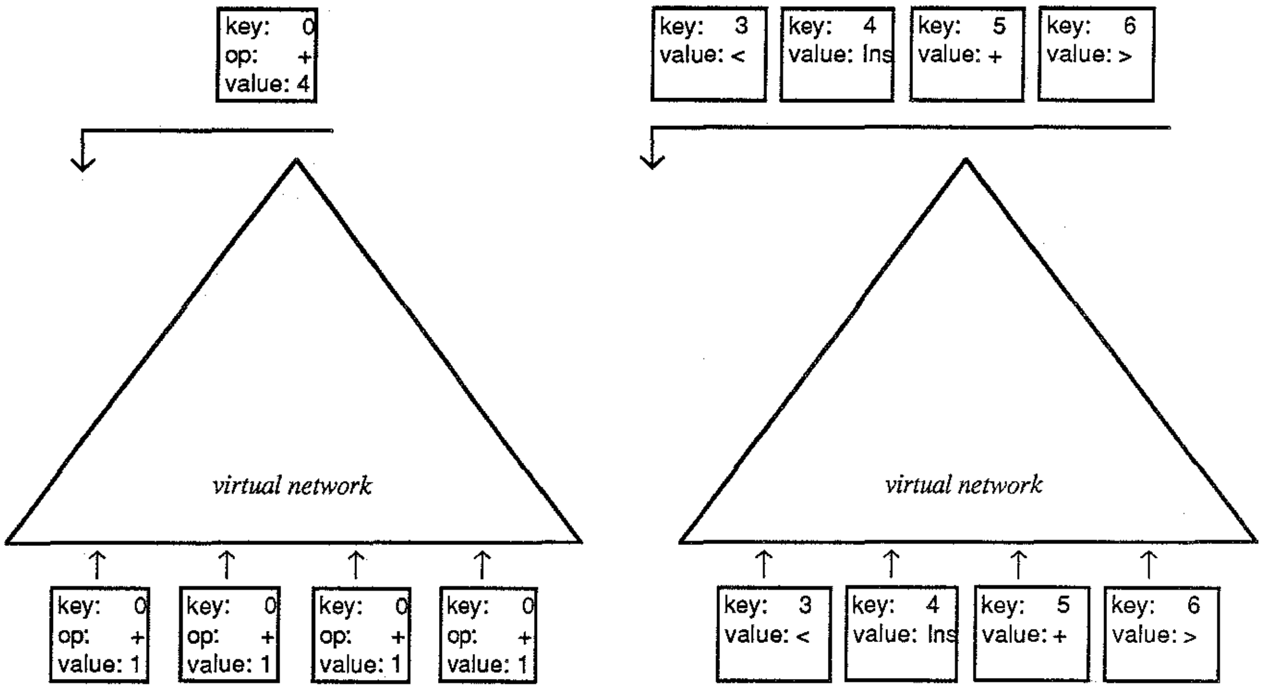


Figure 5. The two message waves used in *ApplyToAll*

A second message wave is used to broadcast the function parameter to the clones (the newly created virtual L cells). Each L cell containing part of the function parameter sends a message containing that cell's symbol as its value and the cell's *index* as its key. The unique keys prevent the messages from being combined, so the function parameter is broadcast serially from the root of the virtual machine. Each clone contains a filter which selects and keeps one symbol from the message stream on the basis of the cell's *clone-id*.

Synchronisation is an important facility for parallel computers. In the FFP machine, synchronisation is necessary so that an RA can be replaced by its result in a single step, since the distributed nature of machine operation requires that the program be in a consistent state at all times. Synchronisation is also needed in the *ApplyToAll* example to ensure that new L cells have been created before the function symbols they are to hold are broadcast to them. In the FFP machine, synchronisation is accomplished by the *SendReceive* instruction. Since message streams are of a variable length, they require an explicit endmarker, called the *EndOfStream* message. All L cells in a virtual machine, including those not contributing any messages, send one of these endmarkers. No message processor transmits a message until it has received at least one message from both children, so no message is broadcast downwards until all the L cells in a virtual machine have transmitted at least one message and so are in the middle of a *SendReceive* instruction. *EndOfStream* is the final downward message and no L cell finishes the *SendReceive* instruction until this message has been received. L cells that are suspended awaiting a *fork* instruction to be satisfied will prevent subsequent message waves from starting by not beginning their next *SendReceive* instruction.

```
ApplyToAll.0.-.-.-:

if firstL[0]
    then erase symbol ;
if symbol = 'ApplyToAll'
    and directory[1] = 0
    then symbol := '(' ;
if lastL[1]
    then erase symbol ;
if directory[1] = 1
    then begin
        SendReceive ( key = 0,
                      op = +,
                      value = 1)
                    { } ;

        SendReceive ( key = index,
                      value = symbol)
                    { } ;
    end
```

```
ApplyToAll.1.-.-.-:

if firstL[1]
    then erase symbol ;
if lastL[1]
    then symbol := ')' ;
if lastL[0]
    then symbol := '>' ;
if firstL[2] and directory[1] > 0
    then begin

        SendReceive ( )
            { size := value ; } ;

        fork (size+2) ;
        if clone-id = 0
            then symbol := ')'
        elseif clone-id = 1
            then symbol := '('
        else begin
            count := 2 ;
            SendReceive ( )
                { If count = clone-id
                      then symbol := value ;
                  increment count ;
                } ;
            end
        end
```

Figure 6. L cell instructions for the distributed string manipulations of Figure 3

The microcode which was described in Figure 3b is shown in Figure 6, in the final form that the L cells would execute. Each segment is labeled with the FFP function name, *ApplyToAll*, and a directory pattern specifying a part of the expression tree. The directory pattern "0.-.-.-" matches any directory that begins with 0 which is the case for the symbols in the function half of the RA's expression tree.

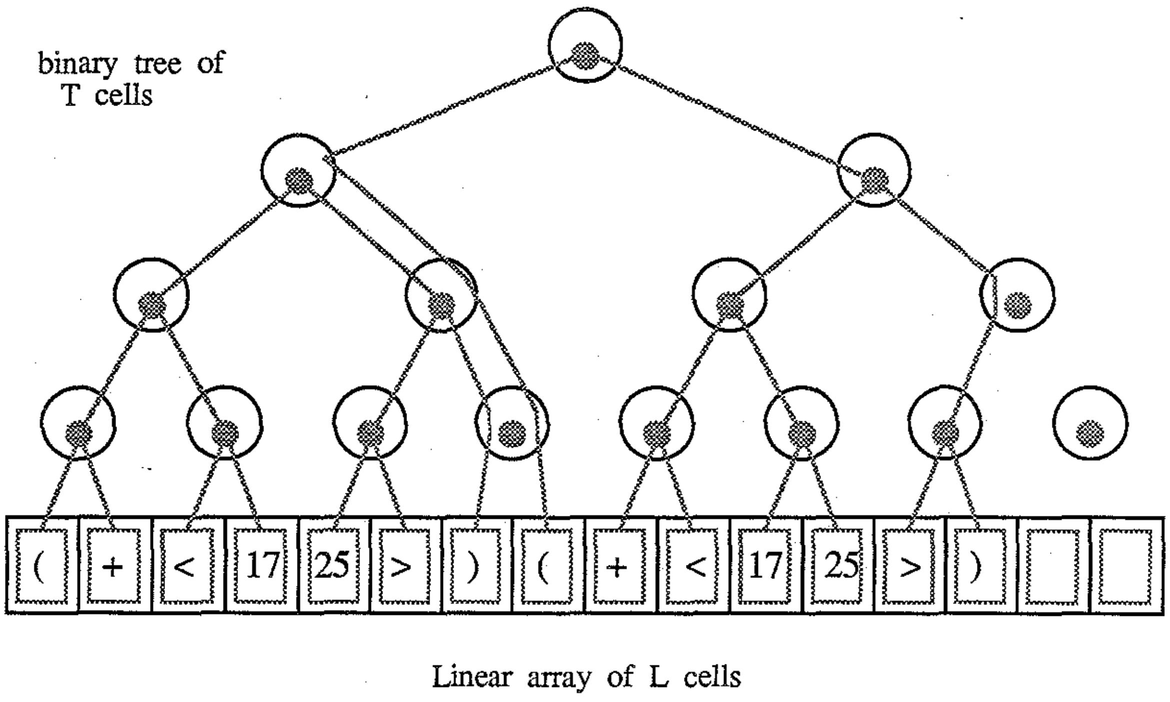


Figure 7. Virtual machines embedded in the physical machine

3.4 The physical machine

The *physical machine* consists of a linear array of independently programmable processing elements, called the *L array* of (physical) *L cells*, which are connected through the leaves of a binary tree of communication resources, called the *T cells*. These physical cells provide hardware resources which are organised into disjoint groups, called *areas*, during execution. Each area supports a single virtual machine. A physical L cell supports one virtual L cell, while a T cell simultaneously supports the communication networks of up to three virtual machines. (This limit results from the locality of RAs. Since an RA contains all the information needed for its evaluation, it needs no communication with symbols beyond its boundaries). A T cell need provide only one of these networks with a message processor; the other two networks only require that messages be relayed. Figure 7 shows, in solid lines, a binary tree of sixteen physical L cells connected to each other and to the tree of the T cells. Virtual L cells and message processors are shown embedded in the physical structure with shaded lines. The physical resources are partitioned into two areas corresponding to the two virtual machines shown in Figure 4.

3.4.1 Partitioning: allocating physical resources to virtual machines

Partitioning, which is described in detail in Section 7.1, is the process whereby the physical resources are grouped into areas to match the occurrence of RAs in the FFP program. The unbalanced nature of the two trees shown in Figure 4 can be seen to derive from the placement of the RAs within the physical machine. Virtual machine operation is designed to be independent of the network structure, in order that there is no constraint on the placement of RAs in the FFP machine. The left to right order of the virtual L cells holding FFP symbols is maintained in the physical L cells supporting them. Empty physical cells may occur within the L array without affecting the operation of the virtual machines. This concept of partitioning allows virtual machines of arbitrary size and placement to be supported within the FFP machine without incurring problems of alignment or fragmentation associated with the physical structure of the resources.

As FFP programs are evaluated, virtual machines are created, altered and deleted. The physical machine continually performs partitioning to regroup the hardware resources to match these changes. A virtual machine evaluating an RA that is copying a large expression may take a long time since all messages must pass through the root of the virtual machine. (Communication systems have been proposed to relieve this congestion [Kehs 78, Kellman 83, Plaisted 84a, Pargas and Presnell 82, Presnell 86]). These slow virtual machines may need to be moved in the physical machine to satisfy space requests in adjacent virtual machines. For this reason, a virtual machine may be supported by a sequence of different areas and a message wave transferring many messages may be supported by a sequence of different message processor networks (each having a different binary structure).

Partitioning is of fundamental importance to the FFP machine. Most machine designs, especially at execution time, subordinate the programs to the constraints of the hardware. This can be seen in the extent to which programs are modified to match the hardware, leading to the existence of a suite of control programs such as compilers, task schedulers and resource allocators. Such machine designs are faced with problems of program decomposition, that is, fitting the running program to the static structure of the hardware. Program decomposition is difficult because, while the structure of hardware is fixed, the structure and size of programs vary greatly even between different stages in the same program's execution, leading at some stage to possibly severe mismatch. The FFP machine has replaced the problem of program decomposition with that of hardware decomposition. The hardware has been designed to delay configuration choices until run-time: the hardware resources in the binary tree are continually reconfigured to adapt to the needs of the changing virtual machines. Each RA is evaluated by a dedicated MIMD processor, namely, the virtual machine supported by a sequence of areas, whose size is proportional to that of the RA. In some sense, the FFP machine can be viewed as having *variable granularity processors* which are constructed during execution from smaller entities, the L cells and parts of T cells, on an individual basis for each task. The term MSIMD has been used to describe an hierarchical machine consisting of multiple SIMD processors executing in parallel [Siegel *et al.* 84]; in this context, the FFP machine could be called an MMIMD processor since it consists of multiple MIMD processors executing in parallel.

3.4.2 Supporting virtual machine space requests

New virtual L cells are created when a virtual L cell executes a *fork* instruction. These virtual L cells must be supported by physical L cells at this specific point in the L array, so it is necessary for the adjacent virtual L cells, defined as a symbol, a partially executed microcode segment and any intermediate values, must shift sideways in order to make space available. These virtual L cells will shift until they arrive in already empty physical L cells. Because the linear form of the FFP program expression must be maintained, these shifting virtual L cells may not overtake each other, but must rather move in groups towards empty L cells. If there are sufficient empty cells between all the *fork* requests, the cost of this *storage movement* is merely the largest single request. If there are no empty cells between those cells requesting space, then the requests interfere and to satisfy them all, some virtual cells must move a distance equal to the sum of the requests. In order to avoid rejecting space requests when there are too few empty physical L cells, the ends of the L array are connected to two stacks into which the excess virtual L cells can be shifted.

This movement of virtual L cells invalidates the current set of areas for supporting the virtual machines, so partitioning must be repeated. This shifting is the only interference between different RAs, which otherwise execute completely independently.

For a virtual machine to be remapped onto the physical resources dynamically, it is necessary that any intermediate state of the partial computation residing in the message processors can be stored in the virtual L cells. In designs where information that is expensive to regenerate resides in the message processors, such as in those of Tolle and Presnell [Tolle 81, Presnell 86], the shifting may be restricted to prevent or limit invalidating currently active areas. The flexibility of being able to support virtual machines with a sequence of areas is very useful, but also incurs some costs. In particular, this shifting interferes with communication operations by invalidating the underlying network. Message waves must be designed to resume rather than restart, in order for a long message wave to make progress in the presence of interruptions caused by space requests in adjacent virtual machines.

3.5 Contrasts between the physical machine and a virtual machine

While the virtual and physical machines appear somewhat similar, they are quite independent, as can be seen by considering their nature in more detail.

The physical machine cycle can be divided into three phases: *partitioning*, *execution* and *storage management*. During *partitioning*, the hardware decomposes itself into the areas to support the current virtual machines. During *execution*, these areas proceed independently with their computations and individual areas may pause, awaiting the allocation of extra L cells. During *storage management*, the physical machine shifts the contents of the L cells to satisfy these space requests while maintaining the left to right order, and the cycle repeats. A better view is that the physical machine alternates between a *system phase* where a single machine satisfies space requests and then builds new areas around the repositioned virtual machines, and an *execution phase* where a set of disjoint areas perform individual computations.

A virtual machine begins operation as a set of virtual L cells holding just the symbols of the given RA. Each virtual L cell acquires address information for its symbol, the function symbol for the RA and its microcode segment, and begins executing that segment. A virtual L cell corresponds to the microcode segment, the symbol and the internal state; this collection of information resides in a particular physical L cell during the execution phase, but may move to a different physical L cell during storage management. A virtual machine may exist over many physical machine cycles with its operation being interrupted to allow the storage movement to be performed that is required for global storage management. In each successive physical machine cycle, the virtual machine's communication network may be a differently shaped binary tree, as the RA moves in relation to the physical tree. The physical machine cycle is transparent to the virtual machines.

Chapter 4
Creating a benchmark for evaluating representations

A benchmark is a set of measures which characterise the expected use of a computer, thereby forming a target for the design and implementation of that computer to optimise. For a conventional computer, the measures are often program fragments that reflect parts of a language, rather than, for example, specific algorithms. Being a language-directed design, the FFP machine should also be evaluated with regards to executing general programs, rather than implementing particular algorithms. The program fragments may range in level from individual instructions to entire programs. A particular machine is evaluated by the costs, such as time or space, which are incurred in supporting one of the measures.

For this research, a benchmark is needed to evaluate the variant FFP machines resulting from different representations. Since this research was motivated by the need to reduce the space requirements of FFP programs, one important measure for evaluating a representation, the space measure, is the number of L cells needed to hold FFP expressions as they are being reduced. The remaining measures in the benchmark reflect the needs of an FFP machine in implementing FFP functions and the underlying system operations. The operation of the FFP machine involves several levels of languages, ranging from the L cell microcode instructions to complete FFP programs. Section 4.1 examines which language level is appropriate for selecting the operations to form the benchmark and Section 4.2 collects those operations.

Choices about program representation are evaluated by their effect on the efficiency of implementing these operations, measured by the time and hardware complexity costs incurred by an FFP machine using that representation. The hardware complexity, which predicts the size of an L cell, is the complexity of the microcode instructions and the physical L cell facilities for supporting virtual machines, and the amount of memory needed by the L cell to hold data and instructions.

4.1 Choosing the target language level for collecting operations

There are several levels of languages in the FFP machine from which operations might be collected for a benchmark. The lowest level, that of the microcode instructions, is inappropriate for measuring the usefulness of different representations, because those instructions should follow from the requirements of implementing FFP, which require that the program representation already be determined. The highest level, that of complete FFP programs, will be avoided. It would be a major task since there is no large body of applicative programs written towards the goal of efficient execution, and, in particular, efficient execution using the FFP machine's model of parallel computation. It is fortunate that complete FFP programs are likely to provide little information beyond that provided by individual FFP functions. Complete programs are used in benchmarks of conventional machine because they expose the effects of context on the operation of instructions. The implementation of a given instruction is affected by that of preceding instructions in the program through run-time information in the form of status bits and other side effects, and compile-time information in terms of where a value is stored (such as a register, a stack or general memory). In this way, the possible choices to be made for a given instruction are affected by the design choices made for the preceding instructions. The interface between primitive FFP functions, which are the basic instructions of FFP, and the rest of an FFP program consists entirely of passing a single value expressed using the sequence constructor. The FFP machine retains this modularity from the language and this isolates the implementations of FFP functions from each other. For these reasons, individual FFP functions are as effective as, and simpler than, complete FFP programs for evaluating different program representations.

Since an FFP language may contain arbitrary primitive functions, the measures can not be specific primitives, but rather, should reflect aspects common to all primitives, including ones not yet proposed. Individual functions are implemented as a group of manipulations performed on their expression trees. A common set of manipulations recur in implementing all FFP functions and it is these manipulations, along with the space measure, that constitute the measures of the benchmark.

These mechanisms that manipulate expression trees are termed *idioms* because they are accomplished by standard phrases of microcode instructions. Idioms correspond to the level of description of the *ApplyToAll* function shown in Figure 3b;

they are distinct from the L cell microcode instructions (such as those shown in Figure 6) that perform them. Idioms relate to expression trees and are independent of the representation*, whereas the microcode instructions that implement idioms change as the choice of representation changes. No relative frequencies for the use of idioms are available, but, in the same way that the actions of instruction fetching and address decoding are common to most instructions in conventional instruction sets, the actions described by the idioms are sufficiently frequent that they may be assumed to be equally important. The results of this research are reasonably insensitive to this assumption.

Idioms are the appropriate level for elements of the benchmark because of their general utility for implementing the FFP language as well as for implementing new facilities such as non-innermost evaluation rules and new languages such as Prolog and Scheme [Smith 84, Dybvig 86, Middleton and Smith 86]. While the emphasis is on the FFP language for historical reasons, the style of execution in the FFP machine is more general, and research into alternative representations should not ignore this generality.

4.2 Collecting the idioms used to implement FFP

The next step is to collect the idioms necessary to implement FFP by examining many FFP functions. Since FFP languages may be defined to include any functions as primitives, it is not possible to examine all FFP functions. Instead, the structure of the FFP language is organised to direct the search for those functions that are likely to demand the most from the FFP machine, and functions proposed in the future may need to be excluded from the language executed by the FFP machine if they require idioms that are too much more complex to implement than the ones discovered here.

The FFP language distinguishes between primitive functions, functional forms and user-defined functions [Backus 78]. For this research, these are all grouped together as *FFP functions* because they are all implemented with the same set of idioms (just as different conventional programs compile to similar machine code)

* This distinction is useful, but overly simple: Chapter 6 introduces some idioms necessary for controlling the mapping of FFP symbols into the L array. Those idioms have no effect on the expression trees and do depend on the choice of representation.

and because the FFP language distinction does not relate to the cost of implementing an FFP function in the machine. For example, the implementation of the *ApplyToAll* functional form is very similar to that of the *distl* primitive and the implementation of the *transpose* primitive is much more complex and expensive than that of the *Compose* functional form. FFP functions are better categorised by the need in their implementation for storage allocation and communication facilities.

Certain FFP functions use neither facility. Their result is either a substring of the RA, or can be constructed from it by simple local operations in the L cells. Examples include the *Constant* functional form, and primitive functions such as *identity*, *tail*, *appendl* and *appendr*. Various FFP functions almost belong in this category, requiring one or other facility in a small fixed amount that is independent of the operand. For example, *assocl* and *assocr* create a new pair of brackets which requires two new virtual L cells. The *maximum* function uses a message wave with one message, independent of the size of the operand.

The second category of FFP functions use only storage allocation. The majority of these are the user-defined functions (functions constructed from other functions), but this group would also include an *iota* function like that in *APL* [Gilman and Rose 84, p. 103].

The third category of FFP functions use only the communication facilities. These functions, such as *reverse*, usually reorganise symbols that are already present to produce their result.

The final category of FFP functions use both communication and storage allocation to a significant extent. The *fork* and *SendReceive* instructions are often combined to support the idiom of duplicating FFP objects within the RA. *ApplyToAll* has already been seen as one example of this. The *transpose* function lies in this category because it requires communication to permute the matrix elements and storage allocation to create space for extra sequence brackets when the operand is not a square homogeneous matrix.

This organization suggests that the last category will be the richest source of idioms and that the consequences of different representations are most evident in implementing FFP functions from that category.

4.3 Implementations of some illustrative functions

A collection of FFP functions that the FFP machine should implement would begin with those defined by Backus [Backus 78, Backus 73] extended with various operations, (primitive functions, functional forms, and general language capabilities that lie outside FFP), that have arisen from investigations into the FFP machine [Chen 81, Dybvig 86, Magó 81, Magó 82, Middleton and Smith 86, Williams 81]. The FFP functions *Compose*, *transpose*, *ApplyToAll*, *Insert*, *equals* and the implementation of user-defined functions are discussed here since they demonstrate the idioms the FFP machine uses to support FFP functions in general.

The descriptions of the implementations are kept at the level of expression tree manipulations to be independent of choices for program representation. No implementation should constrain the generality of the FFP function, that is, it must not restrict the operands beyond that in the definition of the FFP function. The notation [**x**] stands for a general FFP expression, x: it may be either an atom residing in a single virtual L cell or a structure residing in several adjacent virtual L cells. Questions of error handling are largely ignored in these descriptions because they place no new requirements on FFP machine operation and while many different errors might be possible, assumptions can be made about the correctness of operands that are the results of other function applications. When an RA yields an error, the result contains an error flag which modifies the mechanism for determining the function symbol in an RA so that a different set of microcode segments is loaded to handle the error in enclosing applications.

The *Compose* function transforms the RA "(< **Compose** $[\mathbf{f}_1]$ $[\mathbf{f}_2]\ldots[\mathbf{f}_n]$ > [**x**])" into "($[\mathbf{f}_1]$ ($[\mathbf{f}_2]\ldots$($[\mathbf{f}_n]$[**x**])...))". The two original parentheses, the brackets enclosing the function expression, and the *Compose* symbol are erased and n pairs of parentheses are created, one opening parenthesis before each $[\mathbf{f}_i]$, and n closing parentheses after the operand, [**x**].

A programmer may define any function by combining simpler FFP functions using the functional forms. Such a function definition equates a single symbol with an expression describing that combination. When that symbol is applied to an operand in an RA, it is replaced by the function expression, as shown in Figure 8. For the FFP machine, such a definition is converted into a microcode segment which includes the function expression as data, and is targeted for the L cell holding the function symbol. (The other symbols of the RA execute microcode segments that do nothing). That segment requests space to hold the expression, and the resulting clones, each with a copy of the definition, use their *clone-id* to select one symbol from the expression as their own.

$$\text{Mean} \equiv \div \circ [\,\text{Sum}\ ,\ \text{Length}\,] \qquad (in\,FP)$$

$$= \ <\text{Compose}\quad \div \quad <\text{Construct}\quad \text{Sum}\quad \text{Length}\ >> \qquad (in\,FFP)$$

$$(\quad \text{Mean} \qquad\qquad <24\ \ 49\ \ 60\ \ 35>\ )$$

$$\Downarrow$$

$$(\ <\text{Compose}\quad \div \quad <\text{Construct}\quad \text{Sum}\quad \text{Length}\ >>\quad <24\ \ 49\ \ 60\ \ 35>\ )$$

Figure 8. Implementation of user-defined functions

The *transpose* function transforms an $n \times m$ matrix, that is, a sequence of sequences of the form "$<< [\mathbf{x}_{1,1}]\ldots[\mathbf{x}_{1,m}] > \ \ldots\ < [\mathbf{x}_{n,1}]\ldots[\mathbf{x}_{n,m}] >>$" into an $m{\times}n$ matrix of the form "$<< [\mathbf{x}_{1,1}]\ldots[\mathbf{x}_{n,1}] > \ \ldots\ < [\mathbf{x}_{1,m}]\ldots[\mathbf{x}_{n,m}] >>$". The permutation of the symbols is accomplished by sorting them in a message wave in such a way that the matrix is broadcast in column major order from the top of the area. These symbols are stored from left to right in the L cells, yielding the transpose of the matrix operand, by the process of each L cell comparing the position of a message in the stream with its own position in the RA given by its *index*. For matrices of atoms, the sorting can be done by using as keys *directory*[2] and *directory*[1], which are the column and row indices of the matrix. For matrices with more complex entries, the symbols of the matrix need to be permuted within the L array in such a way that the symbols of each $[\mathbf{x}_{i,j}]$ are kept together while the different $[\mathbf{x}_{i,j}]$'s are transposed. The symbols of $[\mathbf{x}_{i,j}]$ share *directory*[2] and *directory*[1] values which would cause messages using them as keys to be merged.

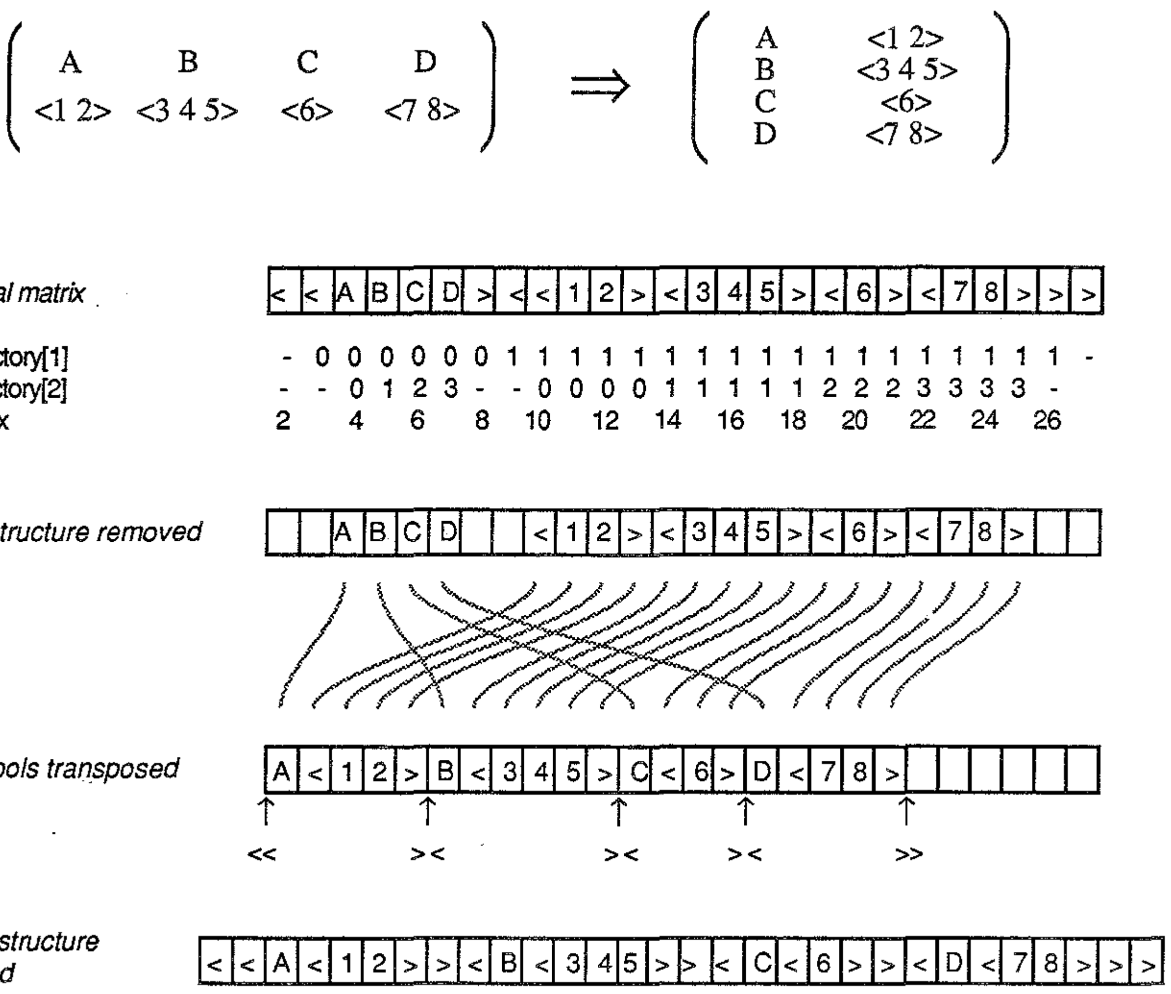


Figure 9. **The stages in transposing matrices on the FFP machine**

Using *directory*[2] and the *index* as the two sorting keys performs the correct permutation while ensuring that each message has unique keys.

To build the correct structure around the permuted $[\mathbf{x}_{i,j}]$'s, it is necessary to create the pairs of brackets to delimit the new rows (corresponding to the old columns), and to delete the pairs of brackets that delimit the old rows (as well as deleting the parentheses and the '*transpose*' symbol). Since it is not necessary that m equal n, nor that the individual $[\mathbf{x}_{i,j}]$'s be of the same size, brackets appear and disappear at different points in the operand and there may be no apparent pattern to the movement of symbols, as for the case shown in Figure 9. The old brackets occur in L cells in which *firstL*[2] or *lastL*[2] are true. An opening bracket, "<", should be created before each $[\mathbf{x}_{1,j}]$ and a closing bracket, ">", should be created after each $[\mathbf{x}_{m,j}]$. If these are created before the permutation is performed, the duplicated *index* values created by the *fork* instructions interfere with the counting in the filters. If these brackets are to be created after the permutation, their position can be inferred from discontinuities in the messages' first keys which are the column numbers of the symbols.

The *transpose* function uses communication and storage allocation in non-trivial amounts; communication to permute the symbols and storage allocation to create new brackets. The two interfere: the needs of the communication operation limit when the brackets may be created and to perform the storage allocation increases the work performed by the communication system (to note the divisions between columns). The communication method places the transposed symbols in adjacent cells which causes the storage movement associated with the different *fork* operations to accumulate.

The implementation for an RA using the *ApplyToAll* function in the form of "(< **ApplyToAll** [**f**] > < $[\mathbf{x}_1] \ldots [\mathbf{x}_n]$ >)", has been described in detail in Section 3.2. Conceptually, the original parentheses and the function expression, "(< **ApplyToAll** [**f**] >   )", are erased, the function [**f**] is duplicated in front of each $[\mathbf{x}_i]$ of the sequence operand and each $[\mathbf{f}][\mathbf{x}_i]$ pair is enclosed in application parentheses. In general, the implementations of FFP functions may reuse L cells, such as those holding the original [**f**] here, to reduce (slightly) the amount of storage allocation.

The *Insert* function is *APL*'s *reduction* operator [Gilman and Rose 84, p. 34], extended to allow user-defined functions as the parameter. *Insert*, denoted in both languages by the symbol '/', repeatedly uses the binary function given as the parameter to combine a sequence of values into one, starting from the right end. An RA of the form "(< **Insert** [**f**] > < $[\mathbf{x}_1] \ldots [\mathbf{x}_n]$ >)" generates a result of the form "([**f**] < $[\mathbf{x}_1] \ldots$ ([**f**] < $[\mathbf{x}_{n-2}]$ ([**f**] < $[\mathbf{x}_{n-1}][\mathbf{x}_n]$ >) >) ... >)". Figure 10 shows the transformation of the expression tree, the insertions and deletions of FFP symbols, that perform this transformation and an abbreviated, dataflow graph, form for the result which shows more clearly that only a single application of [**f**] is being evaluated at a time. The parentheses, the function expression < **Insert** [**f**] >

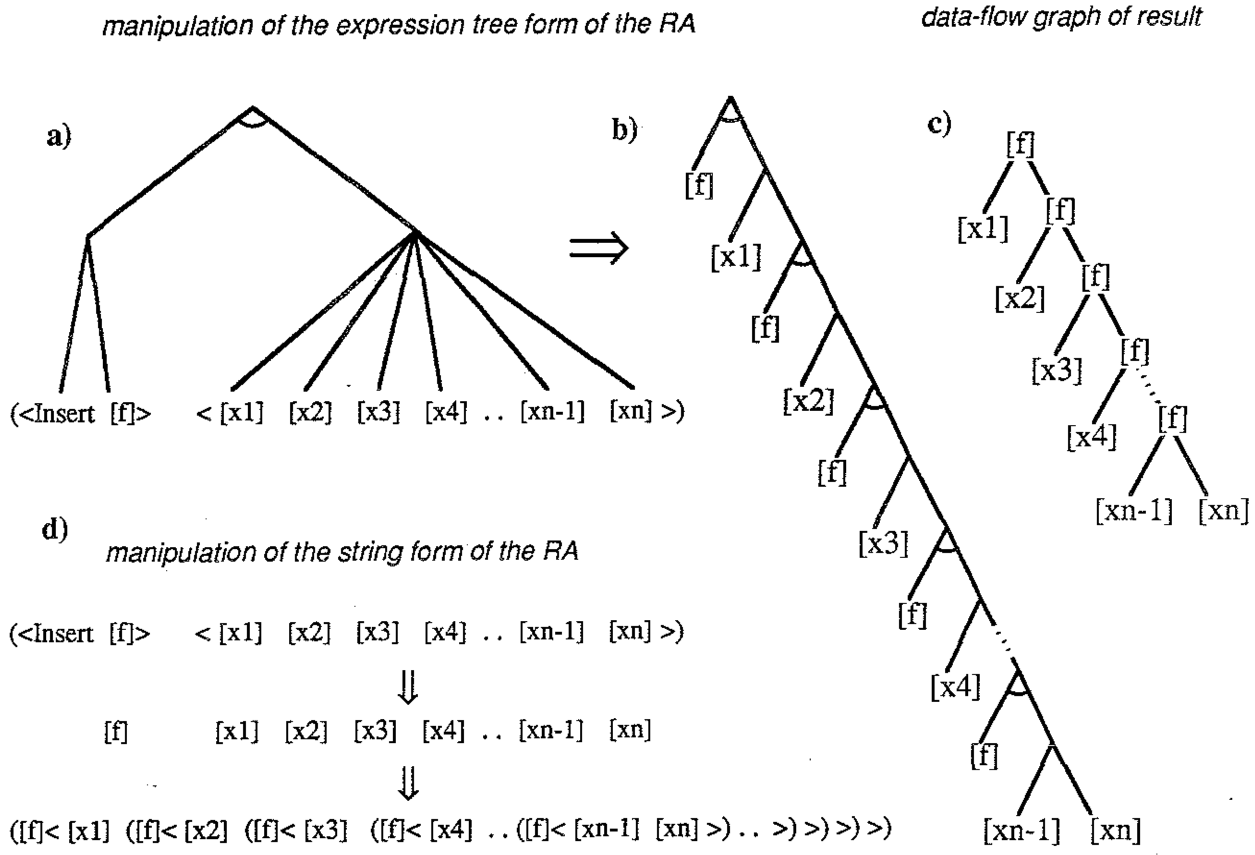

manipulation of the expression tree form of the RA
data-flow graph of result
a)
(<Insert [f]> < [x1] [x2] [x3] [x4] .. [xn-1] [xn] >)
b)
[f]
[x1]
[x2]
[x3]
[x4]
[xn-1] [xn]
c)
[f]
[x1]
[x2]
[x3]
[x4]
[xn-1] [xn]
d)
manipulation of the string form of the RA
(<Insert [f]> < [x1] [x2] [x3] [x4] .. [xn-1] [xn] >)
[f] [x1] [x2] [x3] [x4] .. [xn-1] [xn]
([f]< [x1] ([f]< [x2] ([f]< [x3] ([f]< [x4] .. ([f]< [xn-1] [xn] >) .. >) >) >) >)

Figure 10. Expression tree manipulation for the *Insert* function

and the sequence brackets enclosing the operand are erased. The string "([f] <" is duplicated in front of the first $n-1$ elements of the operand, and the string ">)" is duplicated $n-1$ times at the end.

This function is interesting because variations of it probe the appropriate limits for the complexity of L cells. The *Insert* function creates a deeply nested program expression tree, as shown in Figure 10. Since only one application is innermost, and so reducible, at any time, execution time is linearly proportional to the length of the operand sequence. A natural alternative, which may be used when the function, [f], is associative, is a *TreeInsert* function [Williams 82, p. 80], which, instead of the linear shaped dataflow graph of *Insert*, would generate a tree shaped one, as shown in Figure 11 The different data flow graph demonstrates

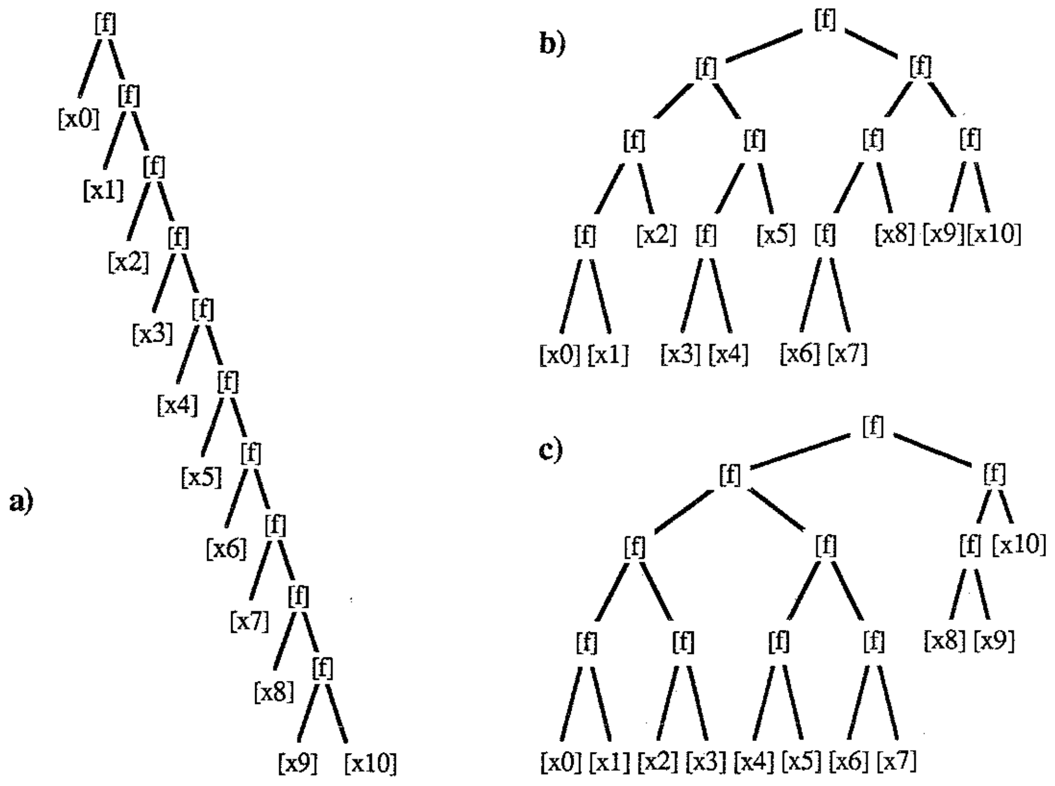


Figure 11. Various forms for the result of the *Insert* function

how the execution time becomes logarithmically proportional to the length of the operand sequence. There are several ways in which the *TreeInsert* function might be implemented, given that any approximately balanced tree of reductions will significantly improve execution. Two simple ways to generate the FFP expressions corresponding to Figures 11b and 11c are shown in Figure 12a and b, respectively. Figure 11b shows a top-down method which splits the operand into two lists of similar size, applies the function "< **TreeInsert** [**f**] >" to each and combines the two results with the function [**f**]. This recursively creates an expression of the same size (in the number of symbols) as *Insert* would create but it requires a logarithmic number of housekeeping reductions (that is, applications of *TreeInsert*) to prepare for actual computation. Figure 11c shows a bottom-up method which inserts the strings "([**f**] <" and ">)" around pairs of operand elements; these applications of [**f**] reduce in parallel to leave a sequence which has been halved in length. Both of these methods are simple to implement and reduce the overall execution time from linear to logarithmic. The bottom-up method has the further advantage that only those copies of [f] that are immediately reducible are created; it incurs only half the storage allocation costs that the top-down method (Figures 11b) or the original method (Figure 10) need. Both of these methods involve a logarithmic number of applications of *TreeInsert*, which perform no useful computation, in comparison with the single such application for the *Insert* function.

```
a)      (<TI    [f]>    <[x0]         [xj]                       [xj+1]    . .    [xn] >)
                                                ⇓
 ([f]<  (<TI    [f]>    <[x0]  . .    [xj]     >)(<TI [f]><      [xj+1]    . .    [xn] >)  >)
```

Determine n, and from that, j=n/2

Insert the symbols ">)(<TI [f]><" before [xj+1]

Insert the symbols "([f]<" before the first L cell

Insert the symbols ">)" after the last L cell.

```
b)

(<TI   [f]>     <       [x0]  [x1]          [x2]  [x3]       . .      [xn-1]  [xn]       >)
                                                ⇓
(<TI   [f]>     < ([f]<  [x0]  [x1] >) ([f]< [x2]  [x3] >)   . . ([f]< [xn-1]  [xn] >) >)
```

Determine n (in case the sequence has an odd number of elements)

Insert the symbols "([f]<" before the first L cell in each even operand element ([x0], [x2], . . .)

Insert the symbols ">)" after the last L cell in each odd operand element ([x1], [x3], . . .).

Figure 12. Two implementations for variants of the *TreeInsert* function

Figure 13 shows an algorithm which creates the same (top-down) expression as Figure 11b, but in a single application. (Once again, copies of [f] are created that are not immediately used). The L cell actions to accomplish this are somewhat lengthy and oppose the assumption that the costs of local L cell instructions can be ignored; this is the first implementation of an FFP function that has required iteration. The value 'left' is the depth of the deepest tree (in Figure 11b) of which $[x_i]$ is the leftmost element. The result expression corresponding to that dataflow graph must have 'left' copies of "([f] <" created immediately before $[x_i]$. Similarly, 'right' copies of ">)" should be inserted following $[x_j]$.

The *equals* function compares two arbitrary FFP objects. In an RA of the form "(**equals** $< [x_1]\ [x_2] >$)", the symbols of $[x_2]$ are broadcast to the L cells holding $[x_1]$. Each of these L cells must identify the appropriate symbol in that message stream and compare that symbols with its own. In a second message wave containing one message, these L cells vote on the equality of the complete structure by indicating whether their own parts matched. Equality is indicated by a unanimous vote which might be accomplished by indicating yes with zero, no with one, and taking the maximum value. The difficult part of this function lies in the L cells holding $[x_1]$ recognising the matching part of $[x_2]$ arriving serially in the message stream. For the generic FFP machine described above, the recognition could be done using the *index* which will have the value 2 in the leftmost cell holding $[x_1]$; an L cell with index $i-1$ compares its symbol with that in the ith message.

4.4 A summary of measures for evaluating program representations

A representation is evaluated by the time, L cell complexity and space costs incurred as the corresponding FFP machine executes FFP programs. The space cost is the number of L cells used to hold FFP expressions. The time and complexity costs quantify the ease with which an FFP machine can implement the low-level system operations and the idioms.

Idioms describe the manipulations to be performed on the expression tree of an RA to implement the associated FFP function. Groups of idioms are distributed to different sites in the expression tree; they are implemented by instructions executed in L cells holding symbols at those sites. The frequently used idioms are: erase symbols from the expression; restructure the expression by creating and deleting brackets and parentheses; and, duplicate a subexpression at another point in the RA.

a)

Erase parentheses, <TI [f]> and top-level operand brackets
Determine 'left' and 'right' as shown below
The first L cell of each [xi] inserts 'left' copies of "([f]<"
The last L cell of each [xi] inserts 'right' copies of ">)".

b)

```
left := 0 ;  right := 0 ;  i := directory[1] ;
while  n>0  begin
   if  i=0  then  increment left ;
   if  i=n  then  increment right ;
   n' := n÷2 ;
   if  i>n'
      then begin  i := i - n' - 1 ; n := n - n' - 1 ; end
      else  n := n' ;
   end
```

c)

i	0	1	2	3	4	5	6	7	8	9	10
n	10	10	10	10	10	10	10	10	10	10	10
n'	5	5	5	5	5	5	5	5	5	5	5
i	0	1	2	3	4	5	0	1	2	3	4
n	5	5	5	5	5	5	4	4	4	4	4
n'	2	2	2	2	2	2	2	2	2	2	2
i	0	1	2	0	1	2	0	1	2	0	1
n	2	2	2	2	2	2	2	2	2	1	1
n'	1	1	1	1	1	1	1	1	1	0	0
i	0	1	0	0	1	0	0	1	0	0	0
n	1	1	0	1	1	0	1	1	0	0	0
n'	0	0		0	0		0	0			
i	0	0		0	0		0	0			
n	0	0		0	0		0	0			
left	4			2			3			1	
right		1	1		1	2		1	1		3

d)

```
                (<TI +>  <[x0]  [x1]  [x2]  [x3]  [x4]  [x5]  [x6]  [x7]  [x8]  [x9]  [x10]>)
                                                       ⇓
(+<                                                                                            >)
  (+<                                                >)(+<                                   >)
    (+<                >)(+<                   >)      (+<                  >)(+<         >)
      (+<       >)          (+<       >)                  (+<       >)
          [x0][x1]  [x2]       [x3][x4]   [x5]               [x6][x7]   [x8]      [x9][x10]
```

Figure 13. A third implementation for the *TreeInsert* function

FFP functions usually erase the function and the two parentheses associated with an RA. (The exceptions are restricted to functional forms in the FFP language, which contain parentheses in their result and, when implemented recursively, include the function symbol again. In these cases, it may be possible to reuse symbols instead of erasing them and creating another copy). Erasing symbols can be accomplished by the appropriate L cell clearing its symbol registers; this idiom is performed locally in the L cells with instructions of negligible cost.

One of the most prevalent idioms used in the FFP machine is restructuring. Once the symbols of the result have been moved into position, the correct structure must be built around them to yield the final result. Restructuring involves erasing and creating *braces*, that is, application parentheses and sequence brackets. Braces carry the syntactic, or structuring, information in an FFP program, in contrast to the atoms which carry the semantic information. Primitive FFP functions need only create groups of brackets since their results are constant FFP expressions. FFP functional forms require more complex abilities of the L cell to create parentheses and brackets in particular patterns. *ApplyToAll* and *Construct* create '<(' and ')>' at the ends of the result and create ')(' at points in between. *Insert* and *TreeInsert* generate sequences of '>)' following elements of the operand sequence and copies of the function parameter enclosed between '(' and '<', preceding elements of the operand. The idiom of erasing braces is performed by a local instruction in the L cells, but creating braces implies the use of storage allocation, which is costly.

Another prevalent idiom is copying some FFP subexpression to another part of a virtual machine. This idiom is accomplished through the cooperation of a number of instructions executed in different L cells. At the destination site, a particular L cell receives the size of the subexpression during one message wave, requests this number of L cells with a *fork* instruction and then the resulting clones create a copy of the subexpression when it is broadcast through the virtual machine's communication network. At the source, the L cells holding the subexpression participate in two message waves; the first determines the size of the subexpression, and the second broadcasts the subexpression for the clones to receive.

Further idioms are more specific to individual FFP functions and harder to distinguish from the instructions that implement them. These include aggregating information from L cells as happens in the voting process for *equals*, permuting values from L cells, as happens in implementing *transpose*, allocating space to create new symbols other than as part of the copy idiom as happens in implementing FFP function definitions and performing local computations, such as preparing sorting keys prior to a message wave.

These idioms provide a language for implementing the FFP functions and are in turn implemented by the L cell instructions. The communication and storage allocation instructions alone are considered to contribute to the time and size complexity, because they require cooperation and synchronisation among different parts of the FFP machine. Other L cell instructions, while just as frequent and important, are simple to implement and operate entirely within a single processor and so are considered of negligible cost. The former set of instructions, the ones considered expensive, are termed *significant instructions*. They are important for evaluating alternative representations in the FFP machine because the choice of representation affects whether particular idioms, both for generating control information and for creating a final result, need significant instructions for their implementation. One goal of a representation is to minimise the use of these instructions and the overall size of the microcode segment.

Chapter 5

Some alternative program representations

The previous two chapters provide the context for evaluating program representations. The generic FFP machine provides two things. It demonstrates a model of computation that remains unchanged across the variant machines, that of FFP functions being implemented by sets of distributed expression tree manipulations. The generic FFP machine also indicates that L cell instructions other than communication and storage allocation can be ignored in measuring costs. The benchmark provides specific measures by which variant FFP machines can be evaluated. The choice for a program representation involves several factors, each affecting different parts of the benchmark. This chapter demonstrates some of these factors to show typical consequences of altering the program representation. The next two chapters look in more detail at particular effects. The representations that are introduced allow FFP expressions to have multiple denotations; this is a new situation that has diverse and far-reaching consequences. The representations also strongly affect implementing the low-level system operations of the FFP machine, and these are examined in Chapter 7. Chapter 8 presents a more complete list of the consequences of design choices relating to program representation.

At this stage, we present some candidate representations to demonstrate the range of consequences that can result from different choices of program representation. These examples also demonstrate that certain choices can interfere with certain properties of the FFP machine that are considered essential. A set of constraints is developed which, if satisfied by a candidate representation, appear to ensure that these beneficial properties are maintained. In the light of these examples, the search for good representations involves satisfying the constraints while pursuing other goals, such as increasing the number of symbols an L cell can hold.

There are two considerations about program representation in the FFP machine, *manipulation* which covers the operations by which the L cells can individually alter their symbols to achieve, in concert, the reduction of the RA, and *denotation* which covers the concrete form a given abstract program takes during execution and the way L cells are organised for holding such an internal form as it changes with execution.

The *template* is the set of symbol registers in the L cell that hold the part of an FFP expression residing in that cell. Template design, which involves choices about the number of symbol registers and the restrictions on their contents, is the principal fabric of this research.

5.1 Some templates and symbol registers

A template will be denoted by the (not necessarily legal) FFP string obtained by filling all the symbol registers. For example, if B can be an atom or a bracket, the term '(B)', which is not a legal FFP expression, represents a template with three symbol registers: the first can hold a left parenthesis, the second can hold an atom or bracket and the third can hold a right parenthesis. There is an implicit positional ordering among the registers which reflects the order of the symbols they hold in the complete FFP expression. Any symbol in the B register follows a left parenthesis held in the '(' register and precedes a right parenthesis held in the ')' register.

5.1.1 Templates containing closing braces

In previous FFP machine designs, the closing braces (sequence brackets or application parentheses) were removed from the program representation in order to reduce the number of L cells used [Magó 79]. The *absolute level number*, or *ALN*, was added as a tag to the L cell's symbol to compensate for this lost information. The *absolute level number* of a symbol describes its depth in the expression tree of the entire program. It is generated during the same preprocessing operation in which the closing braces are removed and it must be maintained by every L cell during the execution of all functions. The end of any FFP expression, whether a sequence or application, is now detected by the presence of an ALN smaller than any inside the expression, that ALN being the one associated with the first symbol beyond the expression.

This overlapping of structure information between adjacent expressions adds complexity to the microcode segments and the low-level system operations. Generating or copying FFP symbols now requires generating or copying (and modifying) ALNs. Maintaining and using ALNs is awkward; in particular, modifying them to reflect new structure may be complex even if only a few sequence nodes in the expression tree are altered. The *lastL* component of a symbol's address was not available in the FFP machines that used ALNs; computing it requires added communication in such machines. This reduced the available choices for implementing FFP functions.

Partitioning demonstrates the complexity incurred when an expression's extent must be determined from subsequent symbols.

The major differences between the *partitioning* mechanism described in Section 7.1 and those used in other FFP machines are summarised here to demonstrate the advantages of explicit sequence markers for machine operation. The method described in Section 7.1 uses messages of three bits to reconfigure a T cell containing one message processor and three switches. Furthermore, it configures the T cells immediately, allowing virtual machines to use their network (in a pipelined fashion) after a small constant delay. Magó's design, as implemented by Danforth, involved T cells with four message processors which could be configured in eight different ways, using messages containing several numeric fields. That method required information to travel from the L cells to the root of the physical machine and back, which incurred a delay logarithmically proportional to the size of the machine. In Tolle's design, *partitioning* configured many programmable message processors per T cell (which would likely have to share physical resources) using an iterative procedure in every cell; each T cell could be in one of sixty-four states.

Another example of the complexity incurred by the lack of explicit endmarkers exists with *storage management*. When an RA extends from the L array into a stack, as described in Section 3.4.2, its virtual machine must remain inactive. The lack of closing parentheses to mark the end of an RA in previous machines prevented connecting a stack to the right end of the L array. Having two stacks (one at each end of the L array), which explicit closing parentheses make possible, makes the storage management process simpler to perform and more effective in its results (since it provides a second I/O port to secondary memory [Stanat and Magó 81]). This was the original motivation for this research into alternative forms for program representation.

In summary, a number of advantages can be realised by not removing closing braces from the denotation of programs: a second virtual memory stack can be added to the L array, and the system operations are significantly simpler using closing braces instead of absolute level numbers (which can, consequently, be discarded). In particular, the partitioning operation can be significantly simplified, so that the virtual machines may now be considered for tasks below the level of RAs [Middleton and Smith 86]. Most improvements found by this research have appeared as better implementations of idioms. The improvements appearing in the low-level system operations, such as *partitioning*, are uncommon, and result from changing the representation from that of previous machines to one which more closely follows the abstract form of an expression. Within the restrictions given in Section 5.2, the implementation of the system operations is insensitive to the choice of representation.

While time costs and the L cell's complexity have been reduced, replacing the closing braces in the denotation of the FFP program increases the number of L cells needed to hold an expression. The number of extra cells depends on the structure of the expression, which depends in turn on the kind of program being run. The expression tree is used to analyse the space requirements for particular expressions. Every atom in an expression corresponds to a leaf in the expression tree and every sequence or application constructed from other FFP expressions corresponds to an internal node. Returning closing braces to the denotation associates two braces instead of one with each internal node of the expression tree, implying that an FFP expression requires an extra L cell for each sequence or application. Shallow structures such as lists or matrices of atoms suffer negligible increases in the number of L cells needed to hold them: one more for a list, and $m+1$ more for an $m \times n$ matrix. The space used by closing brackets in more complex structures may be large. For a binary tree with atoms only at the leaves, since there are about as many internal nodes as there are leaves, returning closing brackets increases the number of L cells needed by a factor of 1.5.

The remainder of this work attempts to recover this lost space by *compressing* the FFP expressions into fewer L cells, ones with templates that contain additional symbol registers. The success of this compression depends on how well the given choice of template allows the denotations of particular expressions to collapse.

5.1.2 Templates containing '$<$ A $>$'

Templates with symbol registers that can hold a single bracket can recover about as many L cells as are lost by denoting closing braces explicitly. For such templates, separate L cells are not required for the brackets associated, in the expression tree, with sequence nodes that are the parents of atom nodes. (More accurately, a left or right bracket uses an extra L cell if and only if the corresponding sequence node's first or last child, respectively, is not an atom). On shallow structures such as matrices of atoms, most brackets in an expression can be stored without using extra L cells. The removal of closing brackets from the denotation in previous representations saved only half of these L cells. In deeply nested structures, such as binary trees, the brackets of the lowest level sequence nodes are stored for free, and, since half the nodes of a tree are at the lowest level, the L cells freed from holding opening brackets balance the L cells spent holding the added closing brackets. Only deep narrow FFP expressions require more L cells under this representation than they require under previous representations; for example, the denotation of "$<<<$ a $>>>$" would now use five L cells ("$<$ a $>$" in one cell and a single bracket in each of the others) instead of the four cells needed to hold the original denotation for this expression, "$<<<$ a".

These templates provide improvements under the space measure, but they provide little if any savings for the time and hardware complexity incurred in implementing idioms. For example, a left bracket can only be created in the same L cell as the FFP expression it is to precede when that expression is an atom; storage allocation is not avoided when the expression is a sequence or application. For these templates, the idiom for creating brackets is implemented by different instructions depending on the context of the surrounding symbols. For example, the *tail* function, having deleted the first element of its operand, should move the outermost left bracket to the second element exactly when that element is an atom. The microcode segments grow in order to contain all the instructions that might apply in different situations as well as the instructions to choose among them. For the L cell initially holding that bracket to decide whether to erase it, it must communicate with an L cell holding the second element. In contrast, the *tail* function can be implemented on the generic FFP machine without any use of significant actions. The *transpose* function, in which brackets must be created for each of the new rows in the result provides another example. Whether this can be done internally by L cells or requires storage allocation depends on whether the matrix entries in the first and last rows are atoms. Figure 14 shows transposing an example matrix using such templates; a bracket can be inserted in the same L cell as the symbol "B", whereas a new L cell must be allocated to hold the bracket to be inserted before the symbols "<1". As with its implementation on the generic FFP machine, *transpose* still requires L cells to perform more than

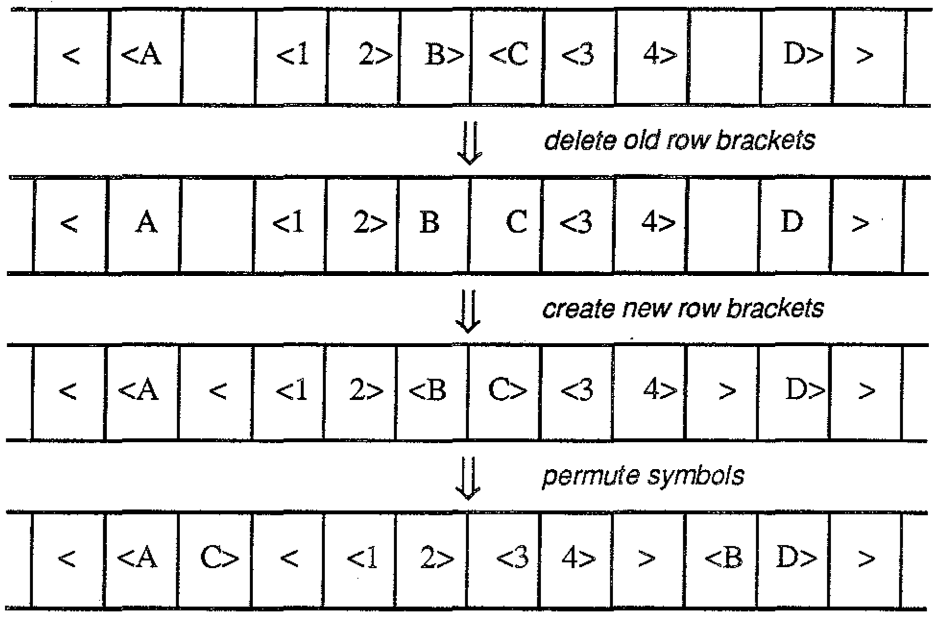


Figure 14. Implementation of the restructuring idiom is context dependent

one idiom involving significant instructions. These templates have complicated the implementation since there are different alternatives to be detected, and this increases the amount of microcode that an L cell must be able to contain.

5.1.3 *Templates containing '<* **A** >*'*

Consider a template that includes the three symbol registers '<* **A** >*', where '**A**' may hold an atom and '<*' is a register which may hold a string of consecutive left brackets, enough so that overflow may be considered a program error. If this brackets register can hold 2^{20} brackets, then the resulting FFP machine is no more restrictive in the programs it can represent than an FFP machine of the previous design that has 2^{20} L cells, each with a template that can hold one symbol. Many registers in the L cell are already of this size and so such a template is more uniform than a template such as '(< **A** >)', where four of the symbol registers consist of one bit indicating the presence of their particular type of brace.

These templates allow all brackets that are associated with sequence nodes having atoms or sequences as their first and last children, to be stored for free.

This includes all brackets within RAs. The need for L cells to hold other brackets (ones associated with sequence nodes that have application nodes as their first or last children) will depend on other parts of the template. As with templates containing '< A >', these templates recover the L cells formerly used to store the brackets of shallow expressions. Furthermore, for deep expressions such as trees, these brackets registers, '<*' and '>*', recover all the L cells used for storing brackets. This is twice as many L cells as were saved in previous FFP machines by omitting closing brackets from the denotations of expressions. These templates turn out to be almost optimal with respect to the space costs incurred in denoting FFP expressions.

A major, unexpected, advantage of these templates is that many FFP functions become significantly easier to implement than under previous templates. This can be seen by considering the *transpose* function. For these templates, a matrix and its transpose use identical amounts of space since their brackets are stored for free and they contain the same atoms (albeit permuted). The idiom which creates brackets for the rows of the result can now be implemented by incrementing '<*' or '>*' registers, an action of negligible cost in comparison with the storage allocation that previous templates required be used. Also, since implementing the idiom for creating brackets does not depend on the context, the code implementing the idiom remains short. A consequence with all compressed representations is that the size of a message increases when it is transferring the symbol contents of an L cell. This is balanced by the number of messages decreasing in proportion to the reduction in the number of L cells. The total volume of traffic is reduced by the grouping of consecutive brackets. Given this, the newly created brackets for the *transpose* function are carried along with whatever brackets exist within the matrix entries, during the message wave that performs the permutation. For these templates, the manipulations of brackets needed to implement *transpose* can be performed without storage allocation and the corresponding code fits easily with the code permuting the symbols.

Many simple functions that originally needed small amounts of storage allocation can now be implemented in a single machine cycle without significant instructions. Figure 15 shows the microcode segments for implementing *assocl* under these templates. Including brackets registers in the template has altered

```
assocl.0.-.-.-:

erase symbol ;
decrement left.parentheses ;
```

```
assocl.1.1.0.-:

if firstL[2]
    decrement left.brackets ;
if lastL[2]
    increment right.brackets ;
```

```
assocl.1.-.-.-:

if firstL[1]
    increment left.brackets ;
if lastL[1]
    begin
    decrement right.brackets ;
    decrement right.parentheses ;
    end
```

Figure 15. *assocl* can be implemented without significant instructions

the denotation of expressions to a form which makes it simpler for microcode segments to create the result of their RA. A secondary advantage of such templates is that simplified microcode segments are easier to design correctly.

The distinctive advantage of these templates is that they allow restructuring idioms to be implemented without recourse to the significant action of storage allocation, because brackets can be created by local instructions in the L cells. Storage allocation causes a virtual machine to exist through at least two physical machine cycles, so removing storage allocation from simple functions that have no other significant instructions doubles their execution speed. (Danforth provided a variant of the *fork* instruction, the *forkc* or constant fork [Danforth 83,p62], which allows an L cell already holding the FFP text to be created, to allocate space and insert the new FFP text before the next physical cycle. *Forkc*, with necessary extensions, would still be less effective than these brackets registers for implementing FFP functions).

Templates that include groups of closing brackets incur the costs of maintaining them, such as larger messages, more microcode (due to the merging of separate microcode segments) and the extra symbol registers. These costs turn out to be small in comparison to the savings in the number L cells and the reductions in microcode for other parts of machine operation. The major advantage of these templates lies in the simplified implementation of restructuring idioms and does not occur with the previous templates considered. Thus, where compressing '<' and '>' registers around an atom register in one template causes the collision of different segments containing significant instructions, compressing '<*' and '>*' registers around an atom register in one template merges segments most of whose instructions are no longer significant.

5.1.4 Templates containing several atoms

Consider a template which includes a sequence of atom registers (that is, registers constrained to hold atoms). The number of L cells saved by these templates depends on the particular FFP expressions being represented; such a template does little to reduce the space used to hold sequences shorter than the number of atom registers. This possibility of low utilisation of the templates presages the problem of scattering examined in Chapter 6; using some templates to their full extent may be impossible (as is the case here) or difficult and costly (as is the case with scattering). Low utilisation of L cells in the L array is accepted in the FFP machine since it simplifies storage management and reflects the fine-grained philosophy which prevents alignment problems between programs and the hardware from arising. There is no corresponding advantage to the low utilisation of the template inside the L cell and these templates introduce a number of difficulties.

Some of these difficulties are demonstrated by an RA involving *transpose*. Figure 16 shows the contrast from the simple implementation possible with the previous template. A template containing '<* **A A A A** >*' is assumed; '.' represents an empty atom register. The first two rows of L cells show the initial and final forms of the matrix; the last two rows of L cells show the intermediate forms when the new brackets are created before or after, respectively, the permutation is performed. There are several possible initial placements for one matrix. For the particular matrix shown, each sequence of six atoms can be placed in two L cells in three ways; this effect will arise in different ways for different matrix operands. The implementation of *transpose* must perform the correct permutation, independent of this placement, which suggests that each matrix symbol be sent in a separate message; until now, the *SendReceive* instruction sent only one message. The filter of the *SendReceive* instruction must now choose and combine several messages to assemble the final contents of its L cell. There is no longer the simple method for selecting messages by counting their position in the downward message stream. In this example, the result requires six L cells, in contrast to the four L cells required by the input operand. The difference arises despite both expressions containing the same number of atoms because the denotation of the result does

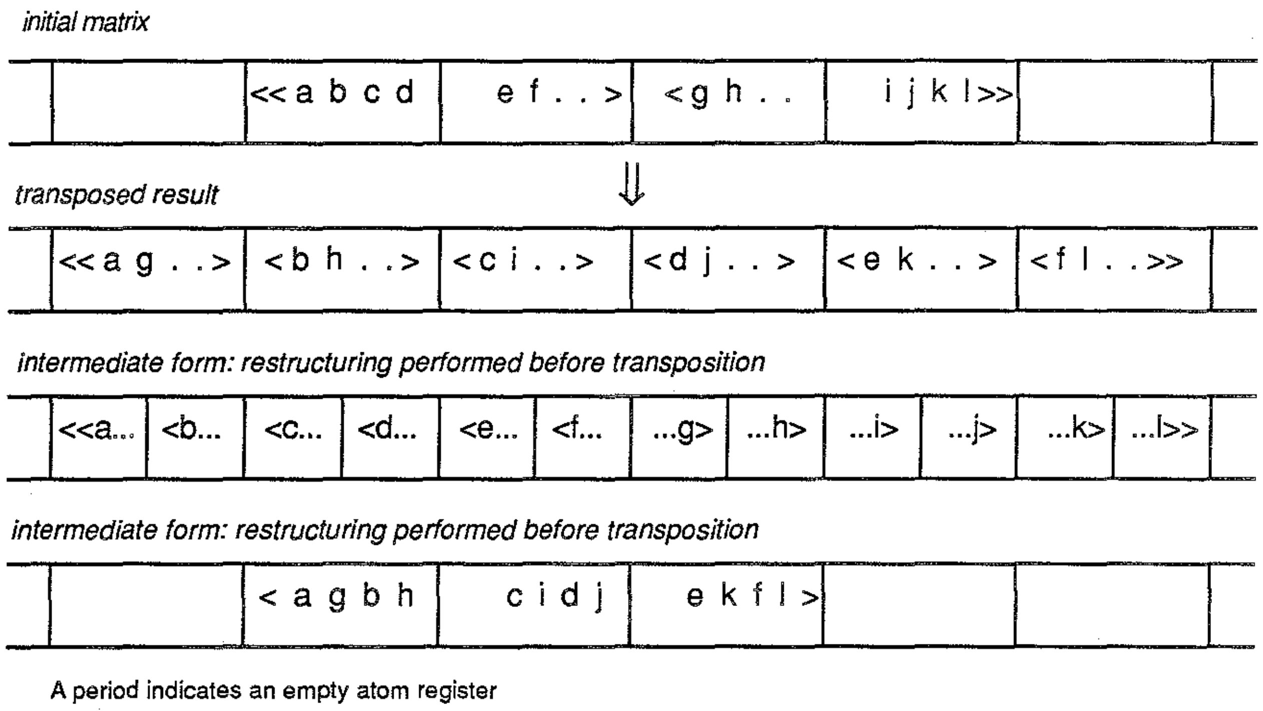


Figure 16. Multiple atoms in an L cell complicate the implementation of *transpose*

not fit the template as well. The brackets belonging to the new rows must be created either before or after the permutation, and whenever brackets are inserted among the atoms, new L cells must be allocated to hold the atoms thus separated. Determining the amount of expansion and where to insert the new L cells is more complicated with this representation and depends on the specific matrix operand. All function implementations must attempt to pack atoms, since otherwise, after several reductions, each atom of an expression might reside in a separate L cell and there would have been no advantage to having several atom registers in the template.

In general, functions require storage allocation under these representations when braces are inserted between atoms. "(< **Compose f g h** > **x**)", which needs two cells, becomes "(**f** (**g** (**h x**)))", which needs three. Furthermore, the first of the two original L cells requests two extra L cells while the symbol '**h**' should be merged into the L cell holding '**x**' when the brackets disappear. This storage allocation could be avoided if templates with several atom registers also contained registers to hold intervening braces.

The presence of parentheses between the atoms would allow the L cell to be part of two independent virtual machines. In disallowing parentheses to be placed between the atoms, storage allocation again becomes necessary to implement *Compose* with these templates. (Registers that can hold intervening braces except right parentheses are not considered, being excessively asymmetrical). The presence of brackets between the atoms increases the possible distance in the expression tree between two atoms that reside in the same L cell. This increases the likelihood that one L cell will contain several sites for manipulations and so that more than one group of idioms will have to be performed by that cell. As was seen above for *transpose*, idioms involving significant instructions interfere with each other; the cell may have to perform more than one *fork* instruction or it may contain several sources and destinations for copying operations. Either a single microcode segment must handle interleaving the groups of idioms or an L cell must be able to handle several microcode segments at once, but in either case, the L cell is effectively simulating a small subtree of L cells and T cells, (or L cells and message processors if the template may not hold intervening parentheses).

For templates that contain multiple atom registers, the space savings are offset by a number of complications in implementing FFP functions. These complications increase the size of microcode segments which the L cell must be able to hold. Any idiom that inserts atoms in an L cell sometimes require storage allocation depending on whether the number of atoms to be inserted is greater than the number of empty atom registers. Finding an atom by its position in the RA may require checking whether several registers are empty. Creating brackets to support restructuring idioms may require the symbols in an L cell be moved to new L cells, incurring storage allocation costs. Where the context of an idiom may determine which of several instructions implement it, the segment must include the code for each as well as the code to choose among them.

It appears that any approach to these problems introduces further difficulties. Templates that include several atom registers cause large increases in the hardware complexity of the L cell. The L cell is increasing in size and the system operations are increasing in complexity at a greater rate than the number of L cells used to hold an RA is decreasing. This suggests that the loss of fine granularity manifests itself in an actual loss in effectiveness of the hardware use. The time costs increase because the L cells are larger, the idioms are now performed by fewer L cells which reduces the parallelism, and tests are introduced to determine from the context the correct instructions needed to implement an idiom.

The difficulties with repacking symbols and context dependent instructions were evident to a lesser extent with templates containing '$<$ **A** $>$'. In both cases, there is the possibility of overflow from the template that must be handled. From an aesthetic point of view, packing symbols together appears to be decomposing the program to fit the hardware, the inverse of the design philosophy that the hardware should be configured to fit the program expression. Templates containing '$<^*$ **A** $>^*$' avoid many of these problems. Creating a bracket by incrementing a brackets register does not require the specific positioning required to insert an atom in a template such as '$<^*$ **A A A A** $>^*$'.

Comparing the three kinds of templates proposed above suggests that an L cell should be prevented from holding a string of symbols where those symbols are located at different sites with which are associated idioms requiring significant instructions. Fine granularity is not simply minimising the number of symbols in a cell; that would suggest that '$<^*$ **A** $>^*$' were a worse template than '$<$ **A** $>$' because it could hold more symbols. This difficulty with one L cell simulating several also suggests that mapping trees of processes onto trees of hardware is not profitable; in both cases, the physical nodes must simulate several virtual nodes (in the case of tree reduction, in order to avoid alignment constraints). The partitioning diagrams for Tolle's machine design, shown in Figure 28, demonstrate this situation [Tolle 81, Figures 5.2, 5.4 and 5.16].

5.1.5 *Templates containing '$(^*$' and '$)^*$'*

FFP programs often contain sequences of adjacent (right) parentheses, arising from the use of the *Compose* function which formed thirty percent of all reductions in some studies of FFP programs [Pozefsky 77]. *Compose* is implemented entirely with restructuring idioms which create several pairs of parentheses. The closing parentheses, one for each of the functions being composed, are inserted together after the operand.

This suggests that one of the symbol registers in the template should be '$)^*$'. Adding '$(^*$' for symmetry allows the left parentheses to be created as easily and yields templates like '$(^*<^*$ A $>^*)^*$'. These templates allow the idioms implementing *Compose* to be supported without using storage allocation. The only significant instruction needed is a single message wave containing one message that counts the number of functions, to determine the number of parentheses to be created. Assuming that such a message wave would complete in a single machine cycle, these templates halve the execution time of the third of the reductions that perform function composition.

These parentheses registers can interfere with implementing FFP functions that need to create symbols outside the original parentheses. Such functions include the functional forms, *ApplyToAll* (Figure 1), *Construct* and *Insert* (Figure 10), which, in building results that are sequences of applications, need to create brackets outside the initial parentheses of the RA. This process may separate a group of adjacent right parentheses into distinct L cells, and so require storage allocation. Selecting the instructions to perform the restructuring idioms for these functional forms depends on the other symbols in the L cell, specifically, whether the right parenthesis register holds more than one parenthesis.

The advantages of parentheses registers are the number of L cells saved for the quite frequent pattern of "$)^n$" (where n is relatively small) and the simpler implementation of *Compose*. The disadvantages are the added complexity for implementing other functional forms, the low utilisation of the five registers in the template, '$(^*<^*$ A $>^*)^*$', and the unappealing property that the L cells now have access to symbols outside their RA.

5.1.6 Templates containing '$\{^*$' and '$\}^*$'

Parentheses registers cause difficulties in implementing some of the functional forms. These difficulties might disappear for templates containing '$\{^*$A$\}^*$', where '$\{^*$' is a register that can hold any sequence of left braces (brackets and parentheses). Since the order of brackets and parentheses is important, such a symbol register is capable of holding far fewer symbols than '$(^*$' or '$<^*$'. For this reason, template overflow can not be assumed to be program error for templates containing such braces registers; it must instead be checked for and handled by the microcode segments. Braces registers allow the restructuring involved with creating and rearranging brackets and parentheses to be performed without using significant instructions. These registers also eliminate all space costs associated with application and sequence nodes in the expression tree. They are optimal for space cost under the restriction that L cells hold no more than one atom.

These reductions in time and space costs are balanced by difficulties in encoding and decoding the strings of braces. Restructuring idioms involve creating and deleting brackets and parentheses. When the brackets and parentheses are combined in a single register the representation of parentheses likely depends on the presence of brackets and so changes when the brackets are altered. Instructions associated with parentheses, either encoding them, for example, creating parentheses, or decoding them, for example, counting them, must now also consider the number of brackets in the L cell. Having the L cells perform complicated calculations such as these conflicts with the assumption that the cost of local L cell instructions is negligible.

There are a number of ways that the ‘{*’ and ‘}*’ registers might be implemented. A straightforward encoding of braces with one bit for each appears wasteful of storage in the L cell. A more compact encoding of the braces is needed, and can be achieved by exploiting common patterns in FFP expressions. Patterns that frequently arise in FFP programs are “<”^{n} and “>”^{n} in deeply nested expressions, “)”^{n} in reductions that involve *Compose*, “<(” and “)>” in reductions that involve *ApplyToAll* or *Construct* and “)>”^{n} in the variants of *Insert*.

These frequent patterns might suggest a template with a series of registers ‘(<*’ ‘(<*’ ‘(<*’ . . ‘A’ . . ‘>*)’ ‘>*)’ ‘>*)’, where the braces register ‘(<*’ contains a count of the number of adjacent left brackets and an implicit left parenthesis preceding them (that is, outside them in the FFP string, or above them in the expression tree). Expressions with frequent parentheses, such as those created by *Compose* and *Insert*, utilise these registers poorly. A braces register might be designed to hold different alternatives, such as “>”^{i} or “)”^{i} or “>)”^{i}. Such a register would consist of a counter to hold i and an accompanying tag to distinguish among the cases.

Braces registers can easily cause expressions to have several different denotations. For example, the symbols “>)” could be stored in one register as ‘>)’^{1}, or two as ‘>’^{1}‘)’^{1}, requiring the microcode to test for the different possibilities.

The increased complexity in manipulating braces that is incurred with the various ‘{*’ and ‘}*’ registers examined here seems to outweigh the advantages of such templates, which are the number of L cells saved and the reduced conflict between the implementations of *ApplyToAll*, *Construct* and *Insert* and that of *Compose*.

5.1.7 Summary of example templates

Figure 17 summarises the behavior of the partial templates described above. Each column compares templates containing the indicated symbol registers with the representation used in previous versions of the FFP machine, for which right braces are removed and other symbols are held in individual L cells. Advantages are indicated by '+', corresponding to a decrease in the factor being considered. The factors considered correspond to the principal costs of operation in the FFP machine: the absolute size of expressions, the amount of storage allocation, the amount of communication, and the complexity of the L cell, measured principally by the amount of storage needed to hold microcode segments. The entry for the system operations emphasizes that these representations provide major improvements in other parts of the machine operation.

Measure	Partial template					
	S	..<*A*>..	..<**A*>*..	..*AA*..	(*..)*	{*..}*
number of L cells						
shallow expressions	−	+	+	++		
deep expressions	−−	=	++	+		
fork						
create brackets	−	+	++			
create parentheses	−				+	+
restructure symbols				−−		
message waves	=	−	=	−−	=	=
microcode storage	=	−	=	−−	=	−−
system operations	++	++	++	++	++	++

Figure 17. Comparison of templates with previous representation

An unexpected result occurred with the brackets and parentheses registers, '<*', '>*', '(*' and ')*'. L cells using such templates can now perform what used to be costly restructuring idioms, that is, the ones that create braces, (usually) without the use of significant instructions. Although fine granularity prohibits compressing symbols to the point where different idioms using significant instructions occur in the same L cell, these registers, at the same time as they compress symbols in that way, also simplify the implementations of those idioms enough to dispense with the use of significant instructions.

It appears that FFP symbols can be divided into two classes on the basis of whether significant instructions are attached to them in implementing FFP functions. FFP symbols which may have significant instructions associated with them are termed *solitary* and cannot occupy the same L cell without causing the L cell to become more complex, that is, require larger programs. *Attachable* FFP symbols can reside in the same L cell as other symbols without increasing its complexity. In representations where the L cell contains more symbols, it can profitably exploit both the extra information and the possibility of creating FFP symbols without resorting to storage allocation.

5.2 Restricting the candidates for program representation

In the previous section, we proposed several partial templates, that is groups of symbol registers that might be included in the complete template. In this section, we propose four qualities that seem necessary in a good template, as suggested by the behavior of these partial templates. We derive reasons justifying the importance of these qualities from basic characteristics of the FFP machine. To the extent that these characteristics are shared by other parallel computers, we may expect that the four qualities may usefully direct choices about representation of programs in those machines, too.

A principal aspect of the FFP machine is that the hardware is reconfigured during execution to match the running programs. As a consequence and because computations can occur in an almost continuous range of sizes, the individual hardware constituents should be small, in order for the FFP machine to construct areas in a similar range of sizes. These two basic characteristics of the FFP machine, that it uses fine-grained processors and that it performs hardware decomposition rather than program decomposition, lead to the need for representations to be natural, order-preserving, well-aligned and fine-grained.

A *natural* representation is one in which the denotation of an FFP expression corresponds directly, without translation, to that abstract expression's string form. Previous versions of the machine left symbols out of the denotation to save space or included the expression tree structure in the denotation to extend the set of possible manipulations. Hardware decomposition, the process by which the hardware is reconfigured to match the structure of the expression, rapidly becomes complex if natural representations are not used. Mapping a string of processors onto a string of symbols is easier than mapping a tree of processors onto an independent tree of expressions and explicitly representing all symbols obviates reconstructing the information they hold.

An *order-preserving* representation maintains the left to right order of symbols from the abstract expression to its denotation. With such representations, the FFP machine can exploit the locality of RAs (RAs require no information about their environment, that is, their context), to limit the communication contention among groups of L cells and so be able to support each virtual network with dedicated disjoint hardware. Various schemes have been examined for embedding the FFP machine in a larger virtual computer that could execute programs larger than would fit in the FFP machine [Frank *et al.* 84]. Some of these schemes involve replacing FFP expressions with pointers to secondary memory where they are stored. Such schemes are not order-preserving; they incur large costs for the FFP machine which must provide rapid access from all RAs to this secondary memory. Kellman's proposal, in repeatedly shuffling the virtual L cells so that they could communicate through the nearest neighbor connections, is not an order-preserving representation; that machine required a rich interconnection network to provide the necessary increase in communication bandwidth.

A *well-aligned* representation is one that forces expressions to be distributed in the L array in such a way that disjoint subexpressions do not share L cells. Consider storing the expression "(< **ApplyToAll** < **Insert** + >><<1 2><3 4>>)" in L cells with the template '<* **A** >*<*'. The brackets immediately preceding the 1 may be stored in the same L cell as either the 1 or the +; such diversity complicates implementing the idioms in the L cells. Moreover, an L cell holding "+ >><<" contains two sites for idioms that use significant instructions to implement the *ApplyToAll* function. The idiom which transmits the parameter function is attached to a site that includes the symbols "+ >" and the idiom which inserts the parameter function is attached to the site of the second left bracket. Interleaving the concurrent execution of significant instructions in the same L cell, such as occurs with the two *SendReceive* instructions in this case, greatly increases the complexity of those parts of the L cell that implement the significant instructions and this conflicts with the fine granularity of the L cells.

A *fine-grained* representation is one that allows the L cells to remain fine-grained processors. Fine granularity is typically defined in terms of hardware measures, such as the area of, or the number of transistors in, a VLSI circuit. A more appropriate definition for language-directed designs would involve language characteristics and such a definition is attempted in Section 8.3. For the moment, let us define fine granularity as the attempt to reduce the hardware complexity of the L cell while still allowing it to perform the idioms associated with the symbols it may contain. Hardware complexity is measured by the storage for microcode segments and data and the complexity of implementing the microcode instructions and system facilities.

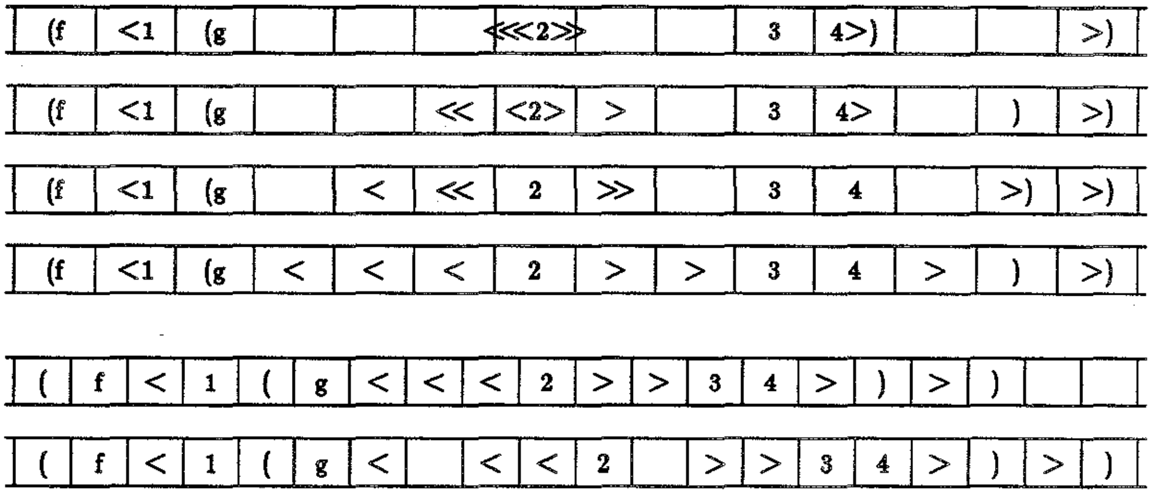
Increasing the number of symbols held in an L cell, the main thrust of this research, directly conflicts with maintaining the fine granularity of the L cells. As a representation packs symbols into one L cell, that cell acquires further work in the form of the idioms associated with the added symbols. As symbols are progressively packed together, the microcode segments take on the character of being many independent segments interleaved together with added conditional statements to control the combination. L cells are effectively acquiring task switching operations and this contravenes the philosophy that the hardware is to be fitted to the running program. It appears sufficient to avoid interleaving significant instructions or allowing multiple cases to arise.

Interleaving significant instructions is complex, and is to be avoided because this complexity is manifested as a relatively large amount of extra microcode. The difficulties of interleaving significant instructions do not arise for the copying idiom which uses two *SendReceive* instructions and a *fork* because these instructions have an obvious sequential order. An L cell performing more than one group of idioms must interleave the corresponding instructions in some way. For an L cell to perform two *SendReceive* instructions (in the same message wave), it must sort the messages they send, and perform both filters on the incoming messages. For an L cell to perform two *fork* instructions, it must divide the resulting range of *clone-ids* to share between the two requests. Serialising unrelated *fork* and *SendReceive* instructions causes unnecessary synchronisation. It appears that only the significant instructions are difficult and so expensive to interleave, since they involve the cooperation of many L cells; the local L cell instructions do not suffer the same concurrency constraints as the significant instructions do, so they can be performed in any order.

A fifth quality, that of high utilisation of the processing elements, while a common goal for parallel computers, is deliberately not used to constrain program representation in this research. The utilisation of processors in a fine-grained parallel computer is ignored in much the same way that memory utilisation is ignored in conventional computers; memory is added without regard to the amount that is idle at a given moment. To accomplish high utilisation in a parallel computer, the processing elements gain responsibilities like those in a multitasking operating system, and care is needed in the programming and compilation stages to ensure (by transforming them as necessary) that the programs conform to the hardware. These extra system operations increase the complexity of the processing elements, opposing their fine granularity, and considerations for the multitasking process lead to an attitude of mapping the program onto the hardware rather than mapping the hardware onto the program. Because fine granularity takes precedence, L cell utilisation is not a primary concern in the FFP machine, although it may be considered if other factors are equal.

Chapter 6
Maintaining unique layouts during execution

In the previous chapter, we discussed design-time decisions regarding the selection of symbol registers for the L cell. In this chapter, we examine some run-time aspects of representation, namely, the ways in which programs are able to use the symbol registers that have been made available to them. The *layout* of an FFP expression is the particular placement of the symbols in that expression's denotation, within the virtual L cells. Layout is distinct from, although constrained by, other aspects of program representation, such as template design. The choice of template provides the possibility for saving space; layout realises this potential.



(f <1 (g <<<2>> 3 4>) >)
(f <1 (g << <2> > 3 4>) >)
(f <1 (g < << 2 >> 3 4 >) >)
(f <1 (g < < < 2 > > 3 4 >) >)
(f < 1 (g < < < 2 > > 3 4 >) >)
(f < 1 (g < < < 2 > > 3 4 >) >)

Figure 18. Templates with multiple symbols lead to scattering

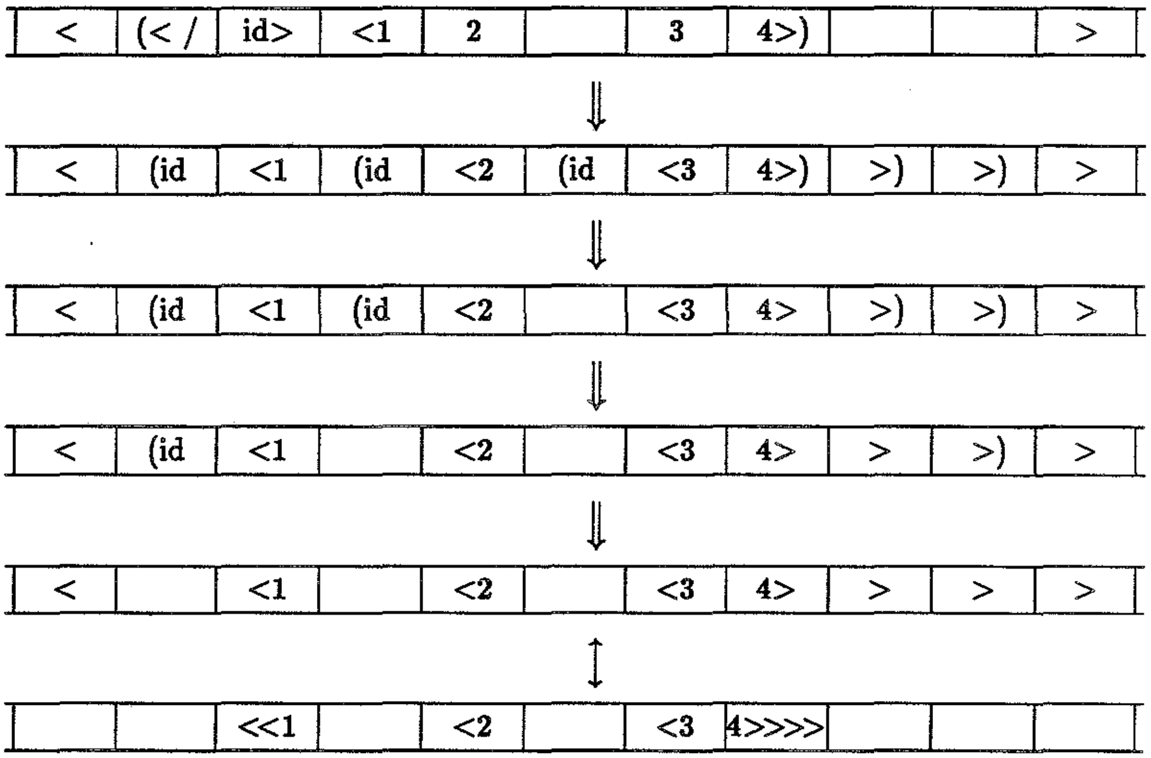
The use of compressed representations raises several questions about uniqueness of layout which did not arise for previous versions of the FFP machine. If the layout for an expression is not unique, a number of complications may arise in the manipulation process since the microcode segments must recognise and handle the different possibilities. The first four layouts in Figure 18 show how templates like '(*<* A >*)*' allow the same expression to have many layouts. Similar complications could arise as easily in aspects of machine operation other than designing the manipulations of the expression trees. In proposing the use of compressed representations, it is necessary to investigate how non-unique layouts of expressions can arise and vanish during execution, what problems such layouts may cause, and what responses are available to the problem. A *compressed representation* is one in which the L cell can hold more than one FFP symbol. A *compact layout* is a mapping of symbols into the fewest necessary L cells.

6.1 The occurrence of non-unique layouts

For previous versions of the FFP machine, uniqueness of layout is guaranteed because L cells could hold only one symbol. (The machine is designed so that the presence of empty physical L cells among those supporting virtual L cells has no effect on microcode segments or system operations). The last two layouts in Figure 18 show this situation for comparison. Tolle's design is an exception in that it also allows several symbols to reside in an L cell, and so the possibility of non-unique layouts also exists for that machine. However, what correspond in that design to the virtual machines are tree structures matching the expression trees of resident programs. The manipulations in that design center around this tree denotation rather than a string denotation and the only consequence of multiple layouts is wasted space.

For representations that satisfy the constraints described in Section 5.2, a compact layout is unique. This can be seen by considering the following method for placing symbols in virtual L cells. Each atom is placed in a distinct L cell; the templates considered in this research may contain several symbols, but the well-aligned and fine-grained constraints both require that only one may be an atom. Each brace that is adjacent to a symbol already in a virtual L cell is placed in that cell if the template allows; otherwise, it is placed in a new virtual cell. Given the structure of FFP expressions and the templates being considered, this process has no opportunity for creating different layouts. For representations that do not satisfy the constraints, layouts may be compact and still not be unique. For example, if the template contains three atom registers, seven adjacent atoms can be laid out in six different ways, all using the minimum number of L cells, namely three.

Since compactness and uniqueness are equivalent for the representations being considered, non-unique layouts are only caused by *scattering*, which is the situation of a layout developing that uses more L cells than necessary. Figure 19 shows such a development occurring. The expression "< / id >" represents the *Insert* functional form with the *identity* function as a parameter. The term is also used to describe the process by which scattering comes about. Scattering can always occur in an FFP machine whose templates can hold more than one FFP symbol, since the expressions that can be denoted in a compact form can also be laid out with one symbol in each L cell. Furthermore, such scattering occurs quite readily in practise.



< | (< / | id> | <1 | 2 | | 3 | 4>) | | | >
< | (id | <1 | (id | <2 | (id | <3 | 4>) | >) | >) | >
< | (id | <1 | (id | <2 | | <3 | 4> | >) | >) | >
< | (id | <1 | | <2 | | <3 | 4> | > | >) | >
< | | <1 | | <2 | | <3 | 4> | > | > | >
| | <<1 | | <2 | | <3 | 4>>>> | | |

Figure 19. Removing parentheses causes scattering during reduction

Some scattering arises as a result of the particular implementations of FFP functions. If an implementation of *tail* simply deletes the first element of the operand, the opening bracket delimiting the operand becomes separated from the first element of the result sequence. For *tail*, a better implementation can be constructed cheaply and easily, however, such better implementations may be more difficult for complex functions, and in particular for data dependent functions such as *ApplyToAllIf* [Magó 82]. The complexity and therefore size of the microcode devoted to recovering unique layouts for function results may require an increase in the space in the L cell devoted to microcode storage. Once scattering has occurred, the extra L cells may remain in the layouts of subsequent expressions, although the implementations for some FFP functions do remove some scattering without specific effort. For example, the *transpose* function generates the brackets that delimit each row of the result compactly, and, in deleting the original brackets, also removes any scattering associated with them.

In any case, for most of the representations considered here, scattering also occurs at the edges of the RAs where a parenthesis that is separating brackets may be erased. This can be seen in the *Insert* function of Figure 19. Thus, the problem of uniqueness cannot be solved simply by assuming that all FFP functions can be implemented so as to leave their result in a compact form. If scattering is to be avoided in order to maintain unique layouts, some mechanism must be created by which the result of each RA is compacted with respect to the symbols adjacent to it.

6.2 Complications arising due to non-unique layouts

One disadvantage of non-unique layouts is that the FFP expression uses more L cells than necessary. The increased L cell complexity associated with the extra symbol storage in the compressed templates has not been balanced by a reduction in the number of required L cells. It was seen in Section 5.4 that a poor choice of template might cause low utilisation of the template registers because few expressions satisfied the template constraints; in this case, the low utilisation is a result of the symbols not being reorganised as they are manipulated.

Another disadvantage of non-unique layouts is that certain FFP functions may become extremely difficult to implement in the presence of scattered layouts. Implementing the *equals* function is simple under the assumption that one argument can be laid on top of the other allowing an L cell by L cell comparison to be made. Where two identical expressions can be laid out in different ways, as shown in Figure 20, there is no (practical) way to ensure the registration of the two arguments. (Address information, being truncated, is insufficient to align general expressions).

A third disadvantage with non-unique layouts is that the complexity of machine operation increases to handle the wider variety of situations. The implementation of every FFP function must allow for each set of symbols that might appear in a single virtual L cell to be spread over several adjacent ones. A microcode segment must have the same effect whether it occurs once, in a single L

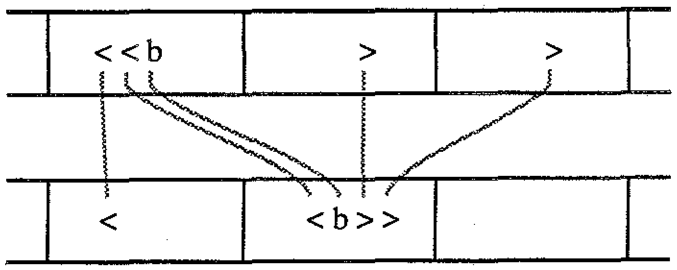


Figure 20. Scattering interferes with aligning symbols for implementing *equals*

cell, or a number of times, in the L cells holding the scattered symbols. A segment that deletes a single bracket in the former case, must be arranged so that when there are several copies of it in the scattered case, all but one avoid this deletion. Similarly, where a segment is to insert symbols, only one copy of the segment should execute the *fork* in the scattered case. The *SendReceive* instruction must be used carefully in order that scattering does not alter the overall message stream excessively. At the very least, scattering will cause more messages to be sent by the copying idioms and those idioms will replicate the waste of L cells. The templates containing several atoms demonstrated problems where the particular instructions that implement an idiom depend on the context. The presence of scattering is similar in the difficulties it raises.

To overcome these complications with non-unique layouts, it appears sufficient that the initial layout of every RA be compact. Scattering that occurs with intermediate expressions during reduction does not appear to affect evaluation of an RA. (This may be because each FFP function performs a conceptually atomic operation, so the functions do not examine intermediate results to control their execution). This means that if compaction is to be performed, it is sufficient to compact the result of an RA, * which involves compacting both the symbols with each other and with the symbols surrounding the result (as all those symbols will become a new RA). It is fortunate that compacting symbols after reduction is sufficient since combining virtual L cells in the middle of a reduction, with all the internal state information, is much more complex and ill-defined.

* Alternatively, compaction might be applied to applications as they become innermost, and so reducible.

6.3 Handling non-unique layouts

The choices available for approaching the problem of scattering (and so non-uniqueness) are shown in Figure 21. The first choice is whether to perform compaction to recover a unique representation for each RA, or to allow scattering to occur and require all system facilities and FFP functions implementations to handle the variety of possible layouts.

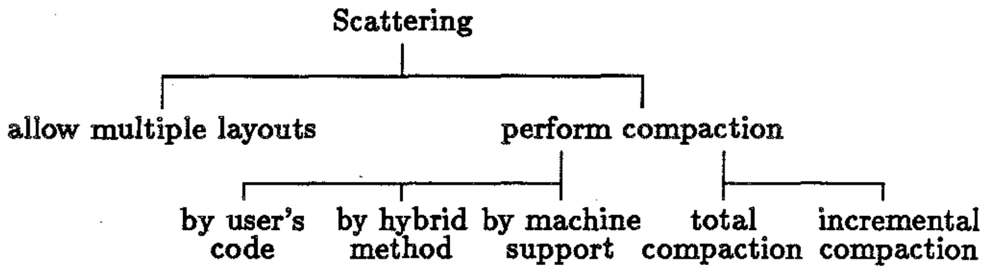


Figure 21. Alternatives for approaching scattering

While allowing scattering to occur raises several problems, it has the advantage that it avoids the cost of compaction; no method for compaction has yet been found that does not incur relatively large costs for the overall machine operation. Many FFP functions can easily be implemented in a way that avoids scattering; some, such as *transpose* may incidentally remove some scattering. Programs that only use such functions do not cause scattering even though no action is taken to avoid it and those programs will execute more rapidly in an FFP machine that is not performing unnecessary compaction. Potential scattering, such as the *Insert* function threatens to cause, as shown in Figure 19, may be undone by such functions. Certain computations may require an explicit garbage collection function, either to implement functions like *equals* or to recover wasted space.

Two further questions arise if scattering is to be repaired by performing compaction. The first question, examined in detail below, is where the responsibility for compaction lies: whether it is a system operation performed during the system phase or an operation performed by the microcode segments during execution phases. The second question is how much compaction is necessary. If compaction is being maintained, there are limits to the extent of the scattering that can occur with a single reduction, before compaction again creates a unique layout. These limits might be used to design incremental algorithms that assumed an almost compact initial layout and were cheaper than compaction algorithms that could compact arbitrary layouts. Such algorithms have not been investigated in any depth.

Viewed as a system operation, such compaction would merge the symbols of a newly completed RA with their context (the surrounding symbols) while they are purely FFP symbols, that is, before they become part of a new RA's virtual machine, and acquire state information whose merging would complicate compaction. This approach of performing compaction during the system phase does not increase the size of the microcode segments, which must be distributed during segment loading and storage management, but the basic physical machine cycle is lengthened which is a significant cost.

A second approach is to make compaction the responsibility of the microcode segments implementing the FFP functions. As a responsibility of the microcode segments, compaction is only performed at the end of the reduction rather than during each physical cycle. Since a single reduction can span many machine cycles this reduces the amount of unnecessary compaction. Furthermore, each microcode segment can exploit properties of its particular FFP function and only repair the specific scattering to which that function is susceptible.

Scattering may occur not only within the result but also between the result and its surrounding symbols, where the application parentheses were deleted, as shown in Figure 19. For microcode segments to recover this scattering, they must have access to brackets outside the RA. This access requires new brackets registers be added to the L cell, yielding templates like '<*(*<* *A* >*)*>*'. (Rarely will more than three of these seven registers be occupied in a single L cell, although such wasted space is a small fraction of the storage in an L cell).

This particular template, while recovering scattering associated with brackets being incorrectly placed, still allows scattering to occur due to incorrectly placed parentheses. Consider an FFP expression which is computing an inner product, "(+ < (× <1 2>)(× <3 4>) >)". The symbols at the end may occupy cells as "(×", "<3", "4>) >" and ")", which become "12", " ", ">" and ")", as a result of the RA being evaluated. The cell that deleted the single right parenthesis broadcasts the presence of an 'outer bracket', allowing the microcode segments to generate "12>", " ", " " and ")". The virtual machine for the RA cannot access the following right parenthesis and so cannot create the compact result, which is "12>)", followed by three empty L cells. The template '<* P >*' described below avoids this problem, while including registers for holding outer brackets.

The implementations of the FFP functional forms (that is, FFP functions that leave parentheses in their results) change slightly to accommodate this approach to compaction. For example, the *Insert* function is usually viewed as using idioms that create a number of '>)' pairs after the final parenthesis. For this approach to scattering, *Insert* would use idioms that create a number of ')>' pairs before the parenthesis. This change is necessary to ensure that the result is compact and that brackets enclosing RAs reside in the same L cells as the parentheses of those RAs.

Making compaction the responsibility of the function implementations in this way requires the language that supports them, that is, the set of idioms, be extended to include idioms that perform compaction with those that perform the actual rewriting. One aspect of fine granularity has been the segregation of significant idioms into separate L cells to increase the parallelism and, in particular, to avoid the complexity of interleaving significant instructions. Since the idioms for compaction require communication, a significant instruction, the segregation suggests the compaction idioms should be performed by L cells other than those performing the actual reduction. This can be done by classifying the parentheses as solitary, that is, separating them from the L cells holding nearby atoms. This classification yields a template like '<* P >*', where P stands for an atom, "$(^n$", or "$)^n$". Such a template suits the implementations of functions like *Construct*, *ApplyToAll* and *Insert*, which already associate distinct idioms with the closing parenthesis.

A third approach in assigning the responsibility for compaction is to distribute different parts to the system phase and the function implementations. The microcode segments are required to create an internally compact result, for which they can exploit the behavior of the specific function, and a system operation will perform a simple movement of braces that may be required following the removal of parentheses.

Various compaction algorithms are described in Chapter 7. As with many machine operations, there are two independent issues: determining what must be done, and doing it. Because of the property of well-alignedness, the process of compacting is easy. The contents of the L cells can be merged entirely if they can be merged at all; they require no splitting first, as would be the case for example, if three pairs of atoms were to be merged in two L cells that could contain three atoms each. The possibility of compaction is also simple to determine by examining the contents of adjacent L cells.

6.4 A summary of uniqueness in representations

For the templates that have been considered, a compact layout, one that uses a minimal number of L cells, is unique. Figure 21 shows a tree of design choices covering the approaches to the problem of scattering. The first choice is between the costs associated with performing reductions on scattered layouts, and the costs associated with recovering compactness at the end of each reduction. If compaction is to be performed, there are two further choices: where the responsibility for compaction lies and how much compaction is necessary. The responsibility can be assigned in various proportions to the microprogram segments or a system facility. Representations that are *well-aligned* allow compaction to be performed easily. Once the opportunities for compaction have been detected, it is only necessary to merge the L cell contents, there is no need to separate symbols that are in one cell.

Scattering appears to be the single disadvantage of the compressed representations proposed in this research. Otherwise, these compressed representations uniformly reduce time, space and hardware complexity costs. Investigating scattering is difficult because the choice of how to attack scattering affects the choice of template, and the choice of the template affects how scattering can occur. There appear to be links between the existence of scattering as a problem, and the choice to perform string manipulations over tree manipulations, such as were used by Tolle [Tolle 81]. The ability to use tree manipulations incurs complexity elsewhere while avoiding the problems of scattering.

Chapter 7

System operations under compressed representations

System operations are those operations performed by the physical hardware to provide the facilities used by microcode instructions, for example, *partitioning*, *storage management* and *address generation*. Choices for program representation can affect existing system operations and can require new system operations, such as compaction, to be created. As a corollary to studying several versions of the FFP machine, this research has improved understanding of the system operations, by separating their essential and surface characteristics. The results of this improved understanding are reflected in the simplification of the algorithms described below, in comparison with those of previous designs. There are two independent aspects to the distributed algorithms that support the system operations of the FFP machine: deciding what should be done and doing it.

The operations of *partitioning*, *address generation* and *compaction* are significantly affected and so are presented in detail. Except for the *compaction* part, *storage management* is basically unchanged, as is communication; the same communication facilities still satisfy the new uses required by different program representations. Since the *absolute level number* no longer exists as information that must be maintained, it need not be carried around when symbols are copied; instead, the messages which transfer symbols must carry the extra FFP symbols of an RA that an L cell now holds, namely the brackets. Messages do not need space for other symbols, such as parentheses or outer brackets, because the idioms that implement FFP operations do not copy them. For the same reasons, these symbols should also not be duplicated when an L cell containing them executes a *fork* operation. Instead, they should remain in the first or last L cell in a set of clones, as is appropriate.

7.1 Partitioning

Partitioning demonstrates the strong dependence of system operations on the choice of representation. The algorithm allowed by the representations considered here is presented and compared with partitioning algorithms in previous FFP machines. *Partitioning* also demonstrates the flexibility available in deciding how system operation will be performed within the constraints of a given representation.

7.1.1 The partitioning algorithm

Partitioning decomposes the physical network of hardware resources into the areas that support virtual machines. Each T cell reconfigures itself on the basis of information received from its children, and passes similar information to its parent. Figure 22 shows that an arbitrary T cell may be involved with at most three areas. From the T cell's point of view, the FFP machine consists of three pieces: the two subtrees reached through its children and the rest of the machine reached through its parent. The locality of RAs obviates the T cell from supporting an area residing entirely within one of these three pieces. Therefore, a T cell need only provide hardware resources to those areas whose RAs span the boundaries between these pieces. The four different ways that RAs can straddle the three boundaries are shown in Figure 23.

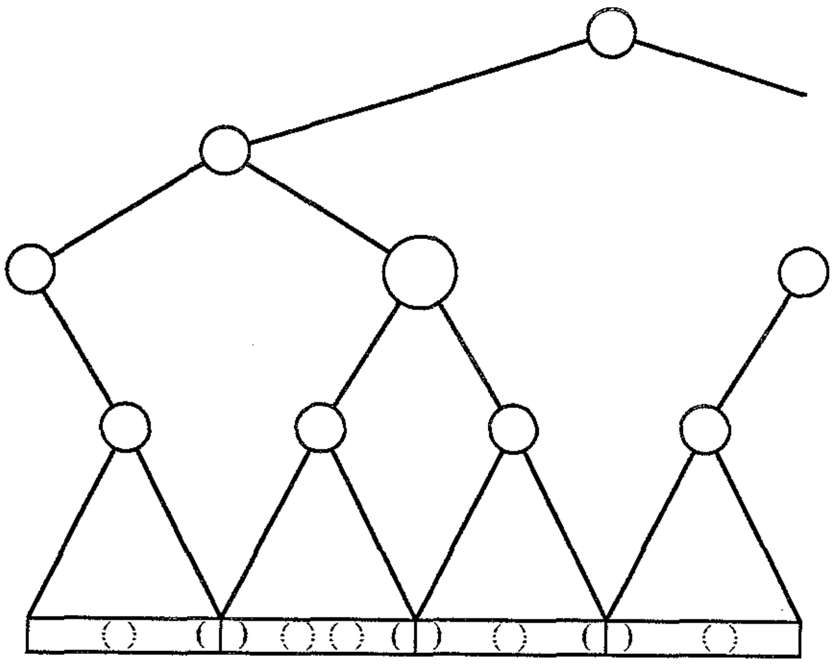


Figure 22. T cell's view of the machine and the RAs it must support

In Figure 23a, three separate areas each cross one boundary. The middle area requires a message processor in this T cell to merge the two streams of messages arriving from the two subtrees. This T cell is holding the root of that area, so the resulting message stream is directed back to the L cells; this area does not need communication to the parent T cell. The outside areas need messages relaying, via the parent T cell, between the L cells inside this tree and those outside. These areas do not need message processors, because there is only one stream of messages traveling upwards.

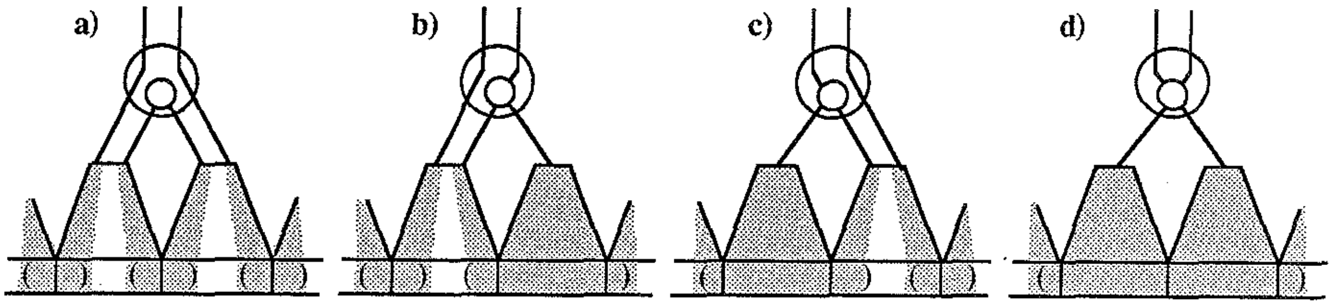


Figure 23. The four configurations by which a T cell supports several areas

Figure 23b shows one area crossing the right and center boundaries. As before, the left area only needs a communication channel connecting the cells of the area inside this tree with those outside. The right area is different; the message processor merges all messages from the right subtree with those messages from the part of that area in the left subtree. The resulting message stream is sent via the parent to the remainder of that area, which is outside this tree. Figure 23c shows the reverse case where one area crosses the center and left boundaries.

The fourth possibility, shown in Figure 23d, is that of a single area crossing all three boundaries. In this case, a single message stream arrives from each of the

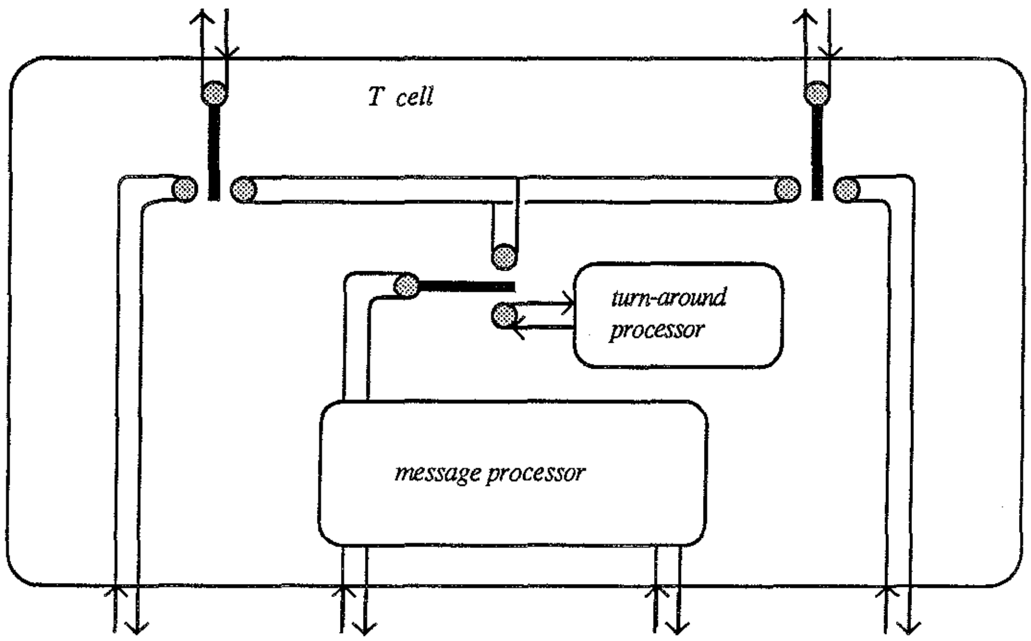


Figure 24. Hardware resources inside a T cell

subtrees to be merged in the message processor and the resulting message stream is sent to the parent.

Figure 24 shows a T cell with the switches which allow it to be configured in these four patterns. The process for setting the switches is extremely simple. The left switch is set outwards for the T cells of Figures 23a and 23b and inwards for the T cells of Figures 23c and 23d. The difference is that the left subtree contains at least one parenthesis in the former case and none in the latter. In a similar fashion, the right subtrees in Figures 23a and 23c contain at least one parenthesis, so their right switch is set outwards, whereas the right subtrees in Figures 23b and 23d do not contain a parenthesis so their right switch is set inwards. The middle switch distinguishes the case of Figure 23a (where both subtrees contain parentheses), for which the output from the message processor is redirected downwards, from the other three cases where that output is sent to the parent T cell and a different stream of messages, traveling downward from the parent, is broadcast.

The process for setting the switches of Figure 23 is as follows. Each subtree will send a partitioning message containing one bit which means "this subtree contains a parenthesis". The switch on each side of the T cell is set outwards if the partitioning message from the subtree on that side is true; the center switch is set to turn messages around if both partitioning messages are true; and the T cells sends the logical sum of the two partitioning messages to its parent.

The actual process is slightly more complex. In the case shown in Figure 23d, both switches in the T cell are set inwards. The message processor is connected to both parent channels in this way exactly when the parent is only taking messages from one channel, the inside one which is connected to the message processor. The inside channel is on the right for T cells that are the left child of their parent, and on the left for T cells that are the right child of their parent. The hardware communication protocols must handle the outside channel being ignored by the parent T cell in these cases.

An L cell must behave like any other subtree in the machine. It can easily generate the message "this subtree contains parentheses" on the basis of the symbols it contains. *Partitioning* uses two *partitioning bits*, which mean, "this L cell contains left parentheses" and "this L cell contains right parentheses", respectively. Since microcode segments can modify parentheses at will, but the virtual machine derived from these parentheses must remain until all virtual L cells have completed, these bits are only updated to reflect the parentheses in the L cell at the end of a reduction, in synchrony with the other L cells in the virtual machine. When a virtual L cell contains a parenthesis, the physical L cell containing it is connected to two separate areas by the two channels to its T cell parent. The virtual L cell belongs in the area into which its parenthesis opens; the physical L cell determines from the *parenthesis bits* which channel to use for messages from the virtual L cell.

Partitioning creates many additional areas (from otherwise unused resources) that do not support active virtual machines. These areas contain parts of the FFP program that are not yet innermost reductions. *Partitioning* divides the FFP program in the following manner and each of the resulting substrings is allocated its own area. The FFP expression is grouped into maximal substrings of symbols other than parentheses. This includes null strings to separate adjacent parentheses. The string ". . 5 >) (+ < 6 7 >) 8 >)) 9 . ." yields ". . 5 >", an empty

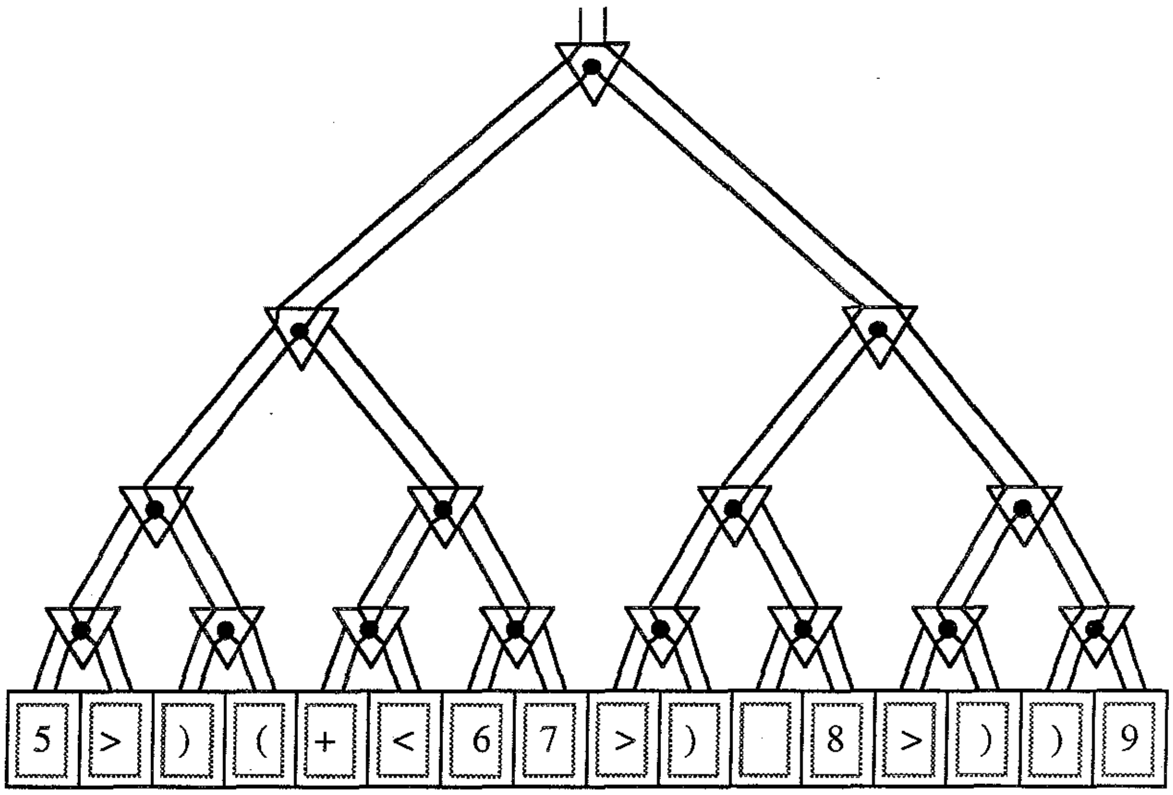


Figure 25. Partitioning physical resources to create virtual machines

string, "+ < **6 7** >", "**8** >", another empty string and "**9** . .". Each parenthesis, now surrounded by two such substrings is added to the one it should enclose. Figure 25 shows a subtree of the physical machine containing this string partitioned into six virtual machines holding the substrings ". . **5** >)", " ", "(+ < **6 7** >)", "**8** >)", ")" and "**9** . .". The two outside areas extend into the rest of the physical machine.

For FFP, an RA is any such substring that contains well balanced parentheses, such as "(+ < **6 7** >)" (and possibly ". . **5** >)") above. No two adjacent areas can contain RAs; these two FFP strings are separated by a null string held in an inactive area consisting of a single message processor (and, in general, some number of relay elements). Therefore, when an L cell is in two adjacent areas, at least one of them is inactive.

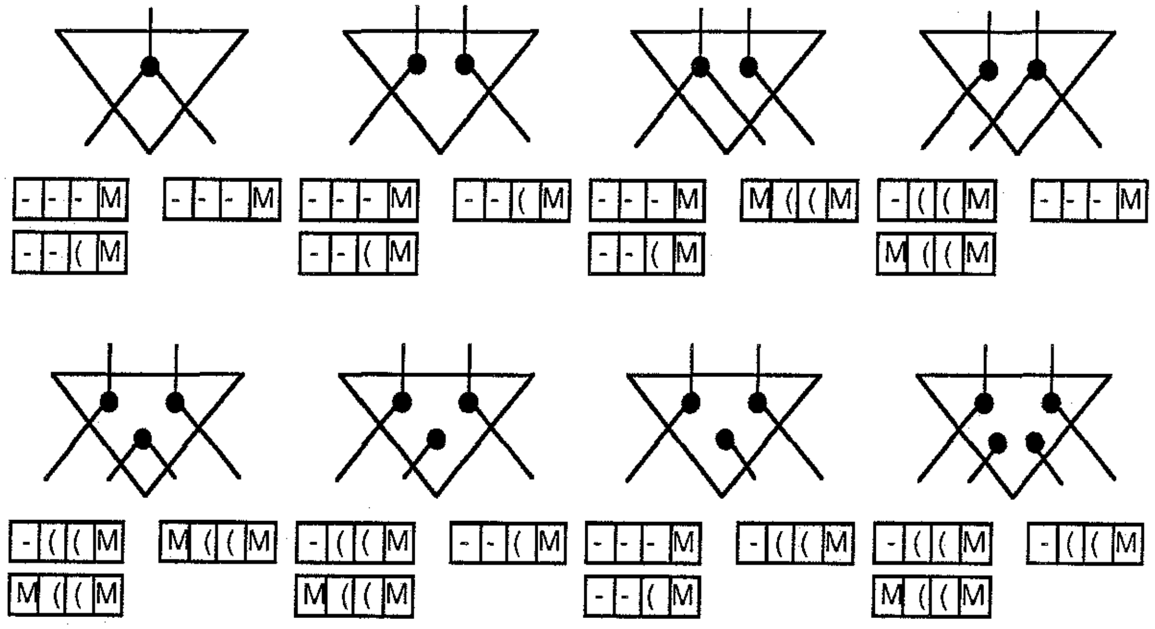


Figure 26. Partitioning in Magó's design

7.1.2 Partitioning operations in previous FFP machines

T cells in the original FFP machine design had four message processors which could be configured in eight different ways. Figure 26 shows some of the partitioning diagrams of that design [Magó 79, Section 4.6.1]. Partitioning used messages of four main types; with one such message from each subtree, a T cell chose the appropriate configuration from a four by four table. The processors transferred ALNs to each other to find innermost and so reducible applications; there were twelve different states that the processors might be in, in order to coordinate these transfers. The partitioning message contained several fields serving different system operations. At least two of these fields were ALNs necessary for *partitioning*, and so the partitioning messages can be considered to contain at least forty bits. (The word size in the FFP machine depends strongly on the number of cells since numbers are frequently fields in addresses; an FFP machine with a million cells leads to the word size being twenty bits). A second phase in that

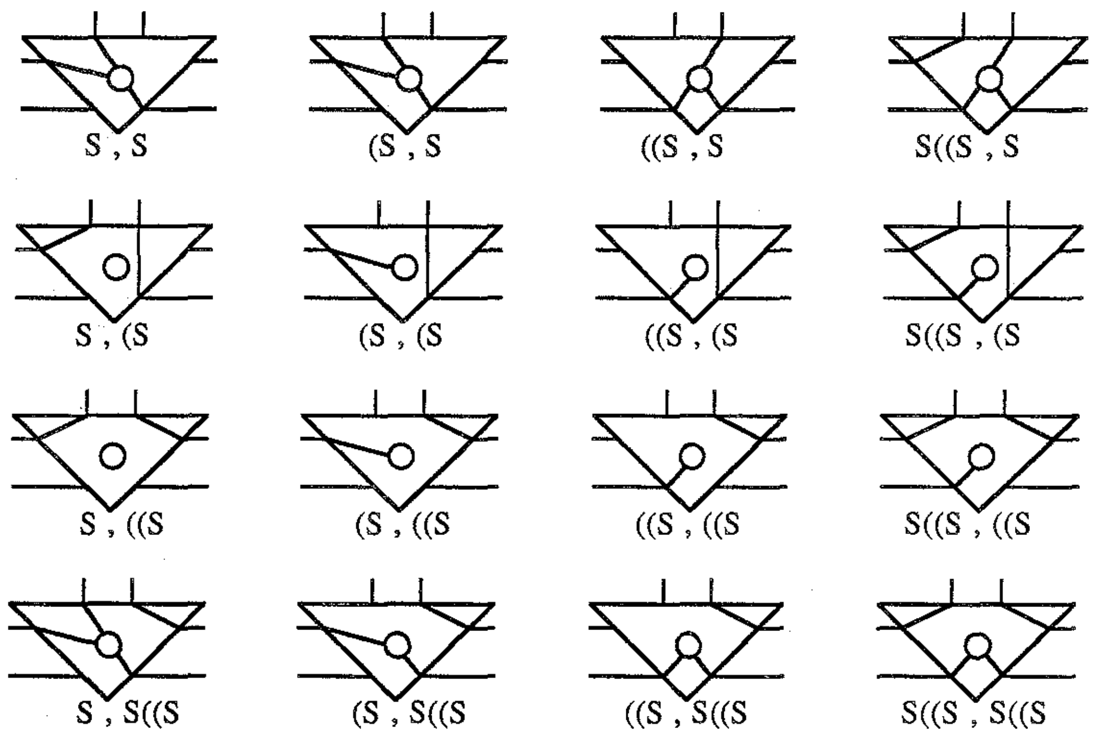


Figure 27. Partitioning in Danforth's design

partitioning scheme involved pruning from an area those L cells holding symbols (not parentheses) that were outside the RA.

In contrast with that method, the T cell described above can be configured in four different ways. The message processor is not needed by the *partitioning* process, which is performed by direct control of switches without using tables. The partitioning messages create the virtual networks immediately, so there is only a constant delay in the physical machine cycle before the virtual L cells can start transmitting messages. The T cell having been demonstrated to need only one message processor, Danforth was able to improve his design to that shown in Figure 27 [Danforth 83, Figure 3.27], but *partitioning* remained the most complicated aspect in his machine, due to the use of ALNs, and the second phase was still required to prune unconnected symbols from the area [Danforth 83, pp. 151-168].

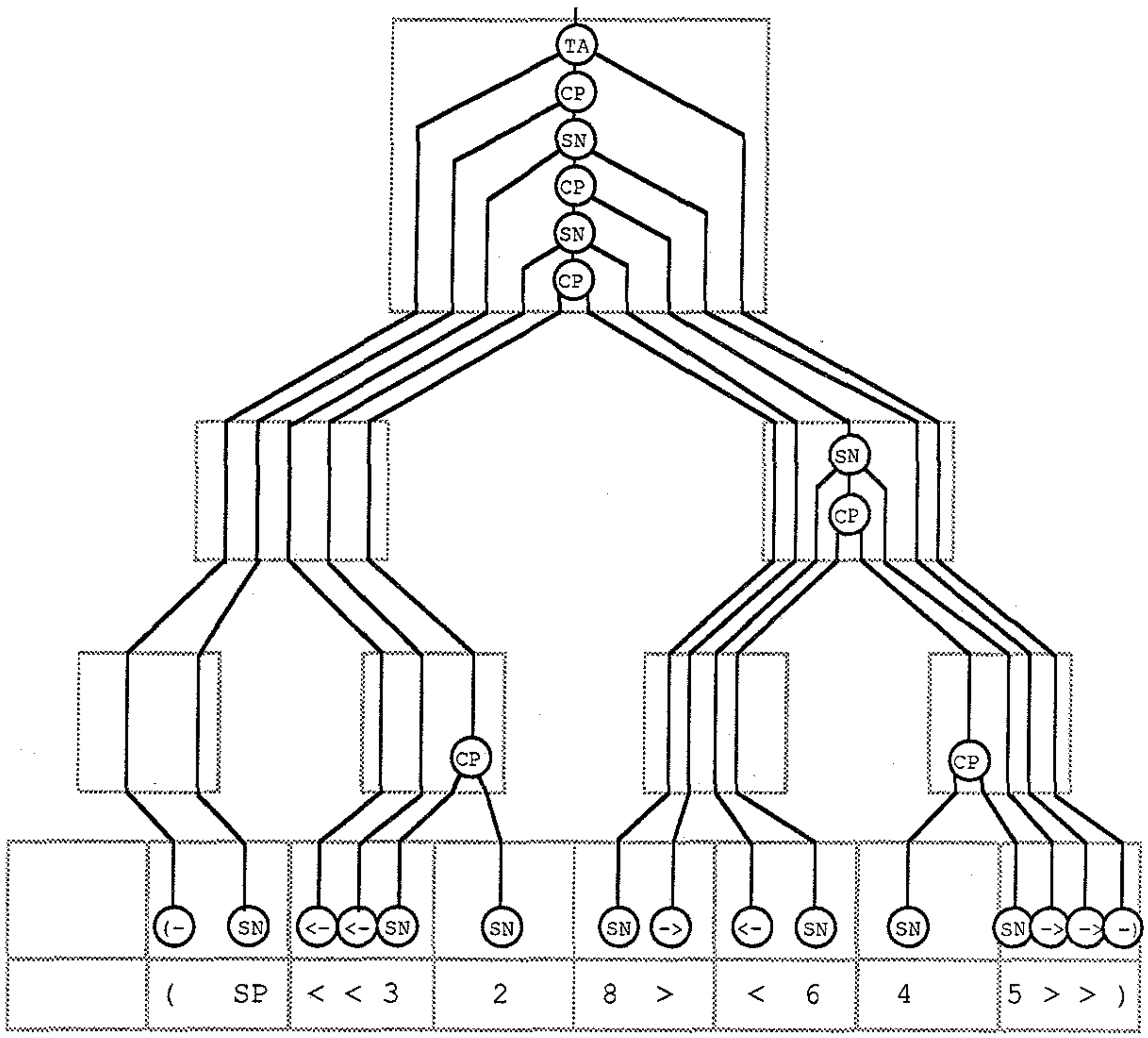


Figure 28. Partitioning in Tolle's design

This complexity arises because adjacent RAs overlap; the endmarker for an RA is the first symbol of the next area.

Tolle's design explicitly embeds the expression tree in the physical tree network, during execution [Tolle 82, pp. 51-80]. To mitigate alignment constraints, the expression tree is extended to contain nodes other than the application and sequence nodes, as shown in Figure 28. A T cell might contain up to sixteen nodes and sixty connections of these extended expression trees [Tolle 82, p. 88]. Each such node might be called on to execute a stored program, which suggests the T cells must contain complex programmable processors to be shared by the nodes.

7.1.3 Design alternatives in the partitioning operation

There are often choices in the FFP machine, between designing the machine so that certain information is unnecessary, generating the information from data distributed over many cells, or by storing and retransmitting the information from the L cells. Several examples can be seen in the *partitioning* process.

```
Given (P.l,L.l,R.l) and (P.r,L.r,R.r), a T cell sends the message:

    P = P.l or P.r
    L = if P.l
          then L.l
          else L.r
    R = if P.r
          then R.r
          else R.l

and sets the switches:

    set left switch outwards   iff P.l
    set right switch outwards  iff P.r
    set center switch down     iff P.l and P.r

The cell holds the root of an active area iff

    P.l and P.r and L.r and R.l
```

Figure 29. Partitioning algorithm

Determining whether an area is active, that is, whether it contains an RA, is an example of information that is regenerated rather than stored. Since an RA can move partly into a stack after it has begun reduction, the partially completed virtual L cells remaining in the L array must suspend operation, and so each area's ability to proceed is recomputed after storage management in every physical machine cycle. Two bits are added to the partitioning message which describe the outermost parentheses in each physical subtree. Their value is ignored if the subtree contains no parentheses, and they refer to the same parenthesis if the subtree contains only one. The T cell of Figure 23a is the top of an area. That area is active if it contains an RA, which is indicated by the two inside parentheses forming a well balanced pair, that is, the rightmost parenthesis in the left subtree is '(' and the leftmost parenthesis in the right subtree is ')'. Figure 29 shows the complete partitioning algorithm using these messages of three bits. The three bits are labeled 'P', 'L' and 'R'; the suffices '.l' and '.r' distinguish messages from the left and right subtrees from that to be sent to the parent.

Distinguishing between a virtual machine which is executing microcode and a virtual machine which is generating addresses is an example of information which can be avoided if the machine is designed to consider both cases as being sufficiently similar. In previous FFP machines [Danforth 83, p. 125, Magó 79, Section 4.6.2], a virtual machine behaved sufficiently differently between generating addresses and executing microcode that the message processors had to be informed which type of operations to expect. Although state information can be maintained in virtual L cells through storage management, the same is not true of state information in the message processors, since the remapping of message processors during storage management is much more complex than the remapping of L cells. Therefore, in those previous FFP machines, this state information had to be retransmitted to the T cells from the virtual L cells. In the FFP machine presented here, the message operations used in the prologue stage are now available as general operations of the virtual machines, so the distinction between these two stages becomes unnecessary.

Another example of these design alternatives is the handling of empty physical L cells. Either empty physical subtrees must be pruned from the virtual machine areas, which increases the size and complexity of *partitioning*, or else empty physical L cells must avoid interfering with operations, such as synchronisation, occurring in the virtual machine.

7.2 Address Generation

Address generation is another system operation that is greatly affected by the choice of program representation. Manipulation of FFP expressions depends on attaching groups of idioms to different sites in the expression trees of RAs. The address in an L cell describes the position of the symbols it holds in the expression tree representation of the RA denoted by those symbols. The components of an address are defined in Section 3.1. They provide information about the expression tree that is needed, first, to resolve the structure sufficiently to deliver microcode segment to the appropriate L cells, and second, to provide the structure information that the segments use. In all FFP functions examined so far, two entries in the *directory* suffice to identify and distinguish microcode segments, and four entries provide enough information for those segments to operate.

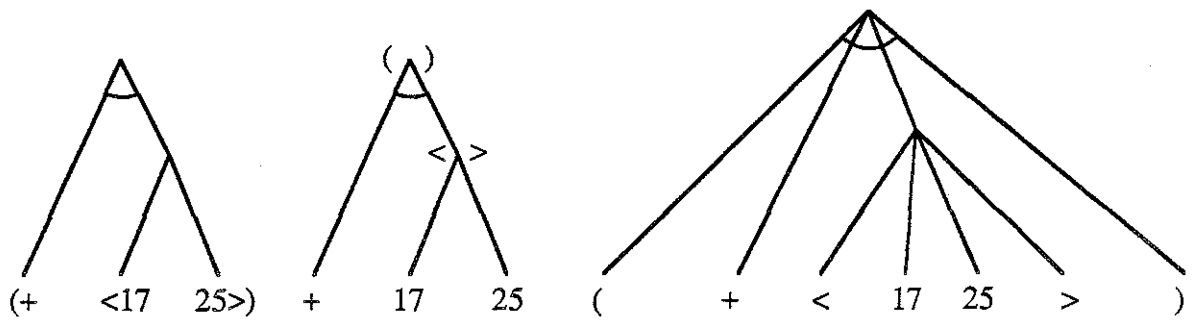


Figure 30. Possible variations on the form of expression trees

There is some flexibility in defining the expression tree. Tolle treated the brackets (corresponding to sequences nodes) beneath a certain level in the expression tree as atoms and brackets above that level as distinct leaf nodes of the expression tree at the same level as the elements of the sequence they enclosed. Magó treated the (opening) brackets above the cutoff level as internal nodes of the expression tree. The FFP machines using the compressed representations of this research seem best suited by an expression tree in which only the solitary FFP symbols, such as atoms, are leaves. The attachable symbols, such as brackets, do not have explicit nodes in the expression tree; instead, they are attached to the appropriate solitary symbol and share the address information of that symbol. Figure 30 shows some possible expression trees for one RA. The choice among these forms for the expression tree has only minor effect on FFP machine operation.

The *directory*, the *relative level number* and the *index* are all generated using a mechanism called a *cumulative message operation.* The message processors are

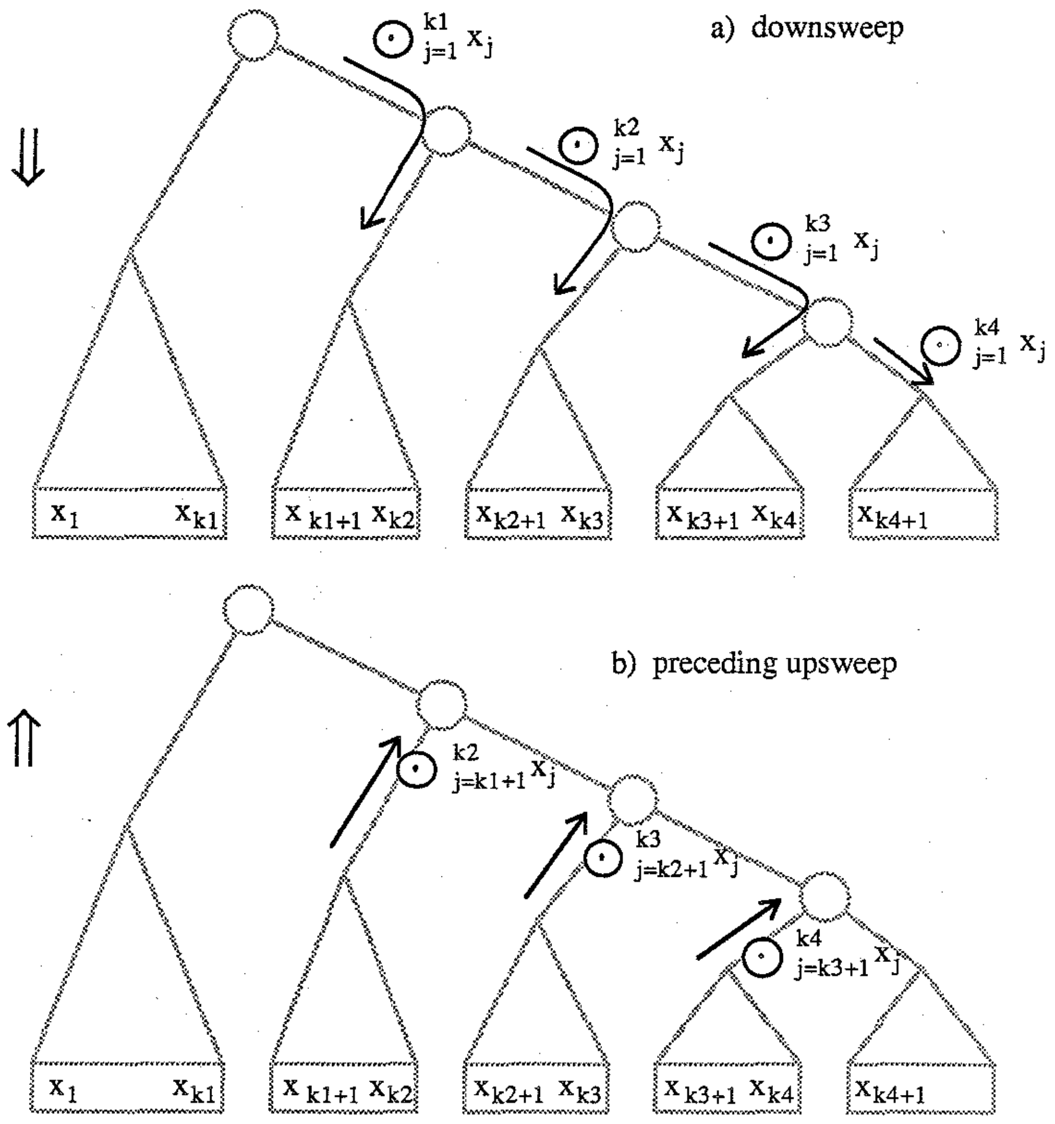


Figure 31. Cumulative message operations

extended to include memory in order that a different message for each individual L cell can be generated using no more time in a message wave than a single message would take (assuming that the speed of the message processor is unchanged).

7.2.1 Cumulative message operations

Cumulative message operations are a particular form of parallel prefix computation [Ladner and Fischer 80], in that each operand resides in a different processor (specifically an L cell) and may only be used once. Pargas describes the use of the tree-structured virtual network for computing solutions to recurrence relations using cumulative message operations [Pargas 84].

Given a sequence of values, x_i, stored in the L cells, L_i, $1 \leq i \leq k$, and an associative operator performed by the message processors, $\odot$, the aim is to generate in each L_i the value $\bigodot_{j=1}^{i-1} x_j$. ($\bigodot_{j=1}^{0} x_j$ is defined to be the unit value of $\odot$). Figure 31 demonstrates the calculation that each message processor in a virtual network must perform, in order that the combined effort implements cumulative message operations. The description works backwards from the final result. Consider the message processor that is the parent of L_i and L_{i+1}. It must receive the value $\bigodot_{j=1}^{i-1} x_j$ from the rest of the virtual network, through its parent, to send to its left child, L_i. It must send to its right child the value $\bigodot_{j=1}^{i} x_j$, which is equal to $(\bigodot_{j=1}^{i-1} x_j) \odot x_i$, that is, the value it received from its parent combined with the value in its left child. In general, consider a message processor whose left subtree contains L_{m+1} through L_n. This processor will receive from its parent, the value $\bigodot_{j=1}^{m} x_j$. It transfers this value to its left child and the value $(\bigodot_{j=1}^{m} x_j) \odot (\bigodot_{j=m+1}^{n} x_j)$ to its right child. Therefore, in an upsweep of messages prior to this downsweep, each message processor, on receiving the two values $\bigodot_{j=m_1+1}^{n_1} x_j$ and $\bigodot_{j=m_2+1}^{n_2} x_j$ keeps the value from its left subchild and sends $(\bigodot_{j=m_1+1}^{n_2} x_j) = (\bigodot_{j=m_1+1}^{n_1} x_j) \odot (\bigodot_{j=m_2+1}^{n_2} x_n)$ to its parent (where $n_1 = m_2$). The message processor at the root of the virtual machine requires special provision. By its design, after the message processor sends a combined value upwards, it waits for $\bigodot_{j=1}^{0} x_j$ before initiating the downsweep. This is most easily accomplished by enhancing the T cell as shown in Figure 24 so that the turnaround loop now replaces such values by the appropriate unit value for the operation, rather than simply transmitting all messages, as was the case before.

In summary, the cumulative message operation works by extending each message processor to retain one member of the pair of values that are combined and sent to the parent, as shown in Figure 32. The value that the parent returns is sent to one child directly, and combined with the stored value in an additional ALU, before being sent to the other child.

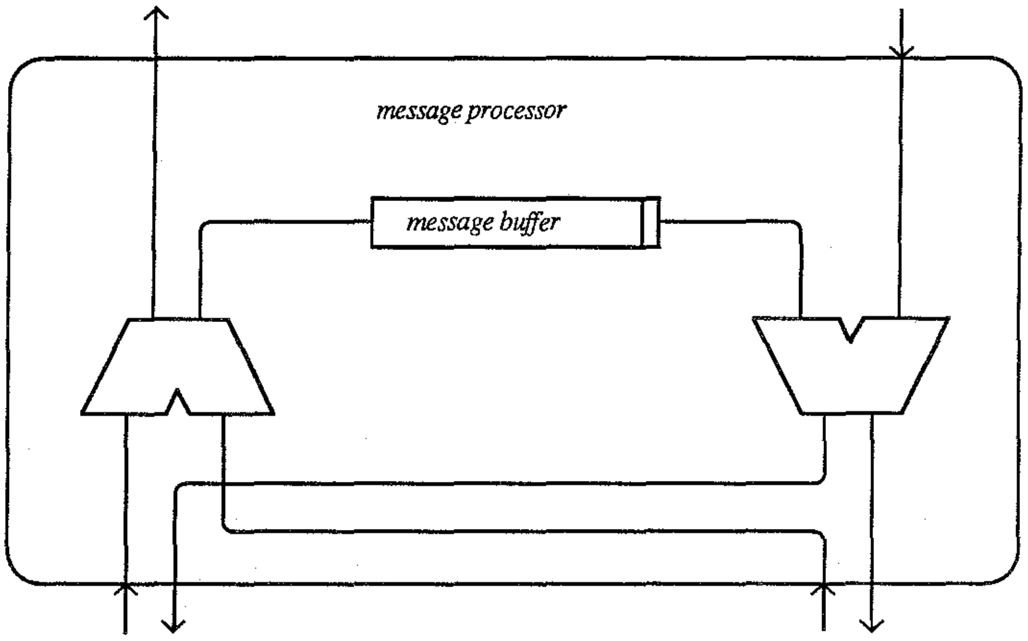


Figure 32. Message processor within the T cell

Different operators can be used to accomplish quite different effects; it is only necessary that they be associative so that the particular tree structure of the communication network is irrelevant. The advantage of this mechanism is its speed, which makes it not only an attractive mechanism where it is necessary, but also a useful alternative in a number of other situations. The presence of empty L cells among the virtual L cells can cause difficulty for this scheme, where it did not for ordinary message operations, because a cumulative message operation requires a message from every L cell. Rather than returning to pruning subtrees during *partitioning*, the message processors will accept a single unmatched message for these operations, and in that case, broadcast the corresponding message returning from the parent message processor. This choice allows for cumulative message operations to be performed on strict subsets of the L cells in a virtual machine.

7.2.2 Calculating the RLN, index, firstL and lastL components

The *relative level number*, being the depth of a symbol in the expression tree, can be measured by the number of unmatched left brackets preceding the symbol in the RA. The value x_i in L_i is $|<^*| - |>^*|$, the difference between the number of left and right brackets which that L cell contains. $\odot$ is addition and the unit value is zero. Since addresses refer to the solitary symbol in an L cell, the L cell must add $|<^*|$ to the value it receives because these brackets precede that symbol. Furthermore, since the expression tree has an application node at its root, 1 should be added to the RLN in every L cell to reflect the corresponding left parenthesis.

The *index* provides a consecutive numbering of the L cells of the area. The value x_i in L_i is 1; $\odot$ and the unit value are again addition and zero.

FirstL and *lastL* are generated locally in an L cell, in microcode segments that use them, from the RLN and the brackets. They are truncated to the same depth as the *directory* vector. These flags are easy to generate because FFP expressions (in an RA) are atoms or are delimited by brackets, which appear explicitly in natural representations. Since an atom is an FFP expression and the RLN is the depth of the atom, *firstL*[RLN] and *lastL*[RLN] are true in every L cell holding an atom. Each left bracket in an L cell corresponds to an FFP expression of progressively larger extent that begins in that cell. Thus, *firstL*[i] is defined to be true when

$$RLN - |<^*| \leq i \leq RLN$$

and similarly, *lastL*[i] is defined to be true when

$$RLN - |>^*| \leq i \leq RLN.$$

Since the RLN counts the left parenthesis, these calculations yield incorrect values for *firstL*[0] and *lastL*[0] in the very first and last L cells, respectively. The calculation described above is simple to correct, or alternatively, the *partitioning bits* can serve as *firstL*[0] and *lastL*[0].

7.2.3 Calculating directories

This description will use the associativity of the combining operator used in the directory computation to present the operation as though it were performed sequentially from left to right. It assumes the template contains '$<^*$ A $>^*$'; the changes to handle different templates are simple.

A symbol's directory counts the number elder siblings for each node in the expression tree on the path from the root to that symbol. An elder sibling is one that precedes the node in an in-order traversal of the expression tree. Each elder sibling is either an atom or a sequence, so the directory operation counts the occurrence of atoms or closing brackets at each level in the expression tree. The structure within an elder sibling is irrelevant, so as a sequence is completed, the counting of deeper expressions within that sequence is discarded. Figure 33 shows the combining operator used to calculate *directories*. The expression(s) corresponding to first non-zero value in the second operand terminate any preceding expressions that are deeper in the expression tree. These preceding deeper expressions correspond to later values in the first operand tuple; these values are therefore suppressed. For each pair of entries in two directories, the directory operator chooses the first value until the first occurrence of a non-zero second value, at which time it adds the two values and subsequently chooses the second value. The unit value for the directory operator is a vector of zeroes. The value generated by each virtual L cell is a partial directory vector with a single one in *directory*[RLN–|>*|] and zeroes elsewhere.

```
counting.left := true ;
for i := 0 to directory.length - 1
    if counting.left
      then directory[i] := dir1[i] + dir2[i]
      else directory[i] := dir2[i]
    counting.left := counting.left and dir1[i] = 0 ;
```

$$\begin{array}{ccccc} 1 & & 0 & & 1 \\ 3 & \otimes & 0 & \Longrightarrow & 3 \\ 5 & & 1 & & 6 \\ 7 & & 2 & & 2 \\ \textit{dir1} & & \textit{dir2} & & \textit{directory} \end{array}$$

Figure 33. The *directory* computation

The directory calculation demonstrates that the simplicity gained by using natural representations is not a trivial advantage. In designs where the closing brackets were implicit, the values to be generated by the L cells were difficult to describe and several incorrect algorithms were proposed. The first executable directory algorithm was derived for these natural representations and then extended to apply to the other representations. Denoting closing brackets explicitly obviates the use of information from outside FFP expressions in determining their extent.

7.2.4 *Other uses of cumulative message operations*

This mechanism was originally designed to implement the address generation operations, but it also implements several other useful operations. For example, if $\odot$ is the *second* selector on pairs, then $\odot_{j=1}^{i-1} x_j$ is x_{i-1}. This cumulative operation performs a right shift by one position of the values x_i. There is no reasonable unit value for the *second* operation, but if the value $\odot_{j=1}^{k} x_j$, that is x_k, which turns around at the root of the virtual machine is left unchanged, then this cumulative operation rotates a sequence of values to the right by one position. Such rotation operations have been proposed before by Pargas and Presnell [Pargas *et al.* 81]; this implementation has the advantage that it does not require that stored programs be executed in the message processors.

Cumulative message operations can be applied to proper subsets of the virtual L array. This allows some values to be rotated while others are unmoved. It also allows a cumulative operation that finds the *relative level number* with respect to particular delimiters. This could be used to determine the scope of bound variables in suitably represented expressions. The cumulative operation facility naturally extends to allow similar cumulative message operations to be performed from right to left, that is, L_i receives $\odot_{j=i+1}^{k} x_j$.

7.3 Compaction

Three mechanisms are presented which can be used to remove scattering, depending on the choices outlined in Section 6.3. A particular machine design might use some combination of these mechanisms.

Because of the well-aligned property of the representations considered, there is never a need to separate the contents of templates before compaction, to achieve denser layouts. If template overflow must be considered, as would be necessary, for example, in a template that could only hold a small number of brackets, more complex decision processes are necessary. For example, a brackets register of four bits can hold up to fifteen brackets. If three adjacent virtual L cells each held ten consecutive brackets, then compaction would need to reorganise these thirty brackets into two L cells, each holding fifteen.

7.3.1 Compaction on demand

Scattering may be allowed to accumulate, in which case, explicit compaction would be necessary to recover space or to implement certain FFP operations, such as *equals*, that require a predictable denotation for the RA. The mechanism described here could be packaged into an FFP garbage-collector function invoked explicitly by the programmer. Since scattering can occur at the edge of any RA, such a function would not guarantee a unique denotation after it had completed and its parentheses were erased. Either *equals* would be extended to handle the limited scattering that might occur immediately following the application of such a function, or the mechanism could be directly included in the implementation of *equals*.

This mechanism generates a pseudo-index value using the cumulative message operations. The pseudo-index differs from the index in that it counts atoms instead of virtual L cells. (Actually, it counts solitary symbols, but for the representations considered, the only solitary symbols in the middle of an RA are the atoms). A virtual L cell that does not contain an atom, receives the pseudo-index for the atom on its left as a side-effect of the implementation of cumulative message operations. Such an L cell contains brackets that should be copied to a neighboring atom, so the cell transmits a message, with the pseudo-index as the key, the number of brackets as value, and + as the operator. Brackets from several cells that are destined for the same atom will be added together when these messages meet in the virtual network. The destination site can recognise incoming brackets by comparing its pseudo-index with the incoming keys. The left and right brackets are handled separately, being sent to the following and preceding atoms, respectively. Where scattering might have caused an empty sequence to be represented as "< >" in separate cells instead of the atom, ϕ, in one cell, the compaction of left and right brackets can be performed in separate waves, and in between, the two brackets, having met in a single L cell, can be replaced by ϕ.

This mechanism uses one message wave of one message to generate the pseudo-indices; it then uses a second message wave containing as many messages as there were groups of brackets that had been separated. Hence, the cost of this mechanism is proportional to the amount of scattering, and independent of the total size of the expression or the extent to which a group of brackets is dispersed. Since this occurs within an RA, it only compacts symbols within the RA and does not leave its result compact with the surrounding symbols.

7.3.2 Compaction on scattering

On the other hand, scattering may be undone as it occurs. In this case, it is either done within the microcode of the individual FFP functions, or it is done by a system operation. The following two methods would be performed as system operations during the physical machine cycle. They should only compact symbols that are not part of a reduction in progress, because the merging of internal data in partially executed microcode segments depends on the meaning of each individual value, and so could not be incorporated in a system operation. Thus, these methods of compaction should be performed after the execution phase in which a virtual machine finishes reduction and before *partitioning* in the subsequent system phase creates new active virtual machines.

These methods have a number of disadvantages: first, they are performed during every machine cycle on the whole FFP machine despite the possibility of compaction only occurring where an RA has been reduced; second, no use is made of the fact that since compaction is being maintained, the amount of scattering that can occur is likely to be limited; and third, they fail to exploit the knowledge, available from the implementation of the FFP functions being reduced, about the specific forms of scattering that will have occurred.

One of these approaches to compaction is to modify the *storage management* algorithm. *Storage management* is that part of the physical machine cycle in which empty physical L cells are allocated to the clones of a virtual L cell created by a *fork*; these empty cells are made available by shifting other virtual L cells [Magó *et al.* 84]. Figure 34 shows how each T cell contributes to calculating the movements. During the upsweep of information, each T cell finds out the number of empty L cells in its two subtrees, requests for space being represented by negative values. During the downsweep, the T cell receives a command from its parent for its subtree to receive or send a certain number of virtual L cells at each of its two edges. This command is in the form of a pair, *arrivals* and *departures*, which describe the flows at the left and right edges, respectively; for both, a positive value indicates a flow to the right. The T cell sends similar commands to its children by resolving its command with its knowledge of the available space in each subtree. The command that arrives at an L cell indicates how many symbols pass through that cell and in which direction.

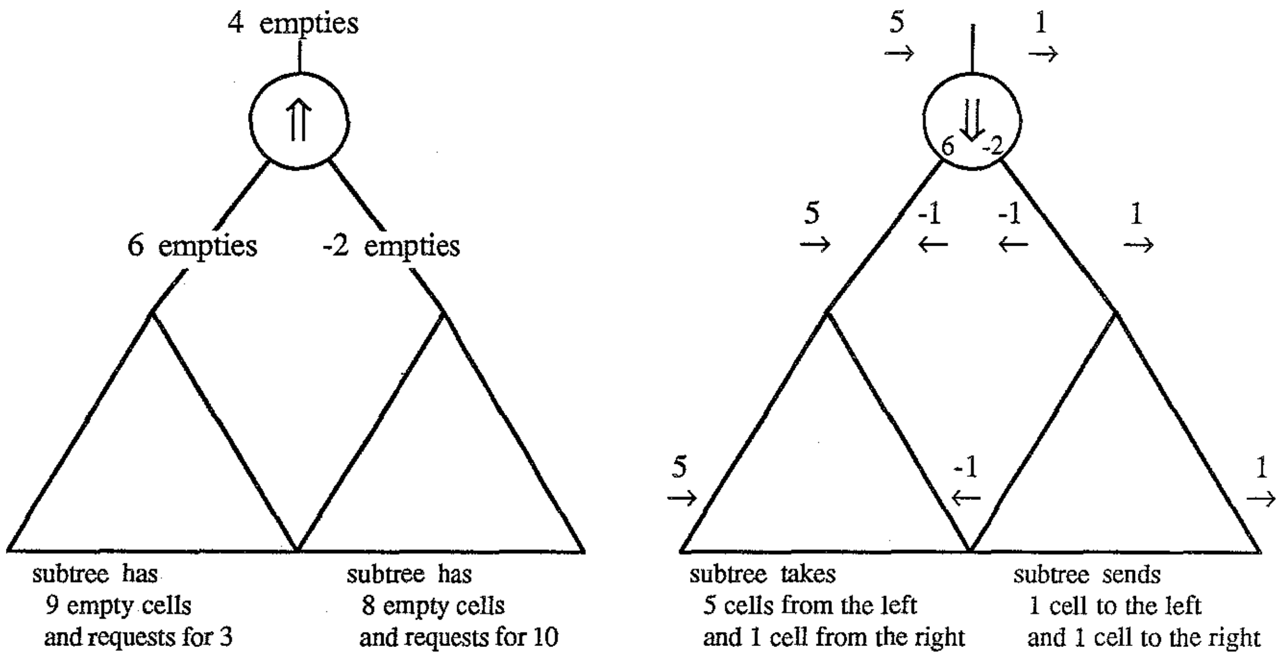


Figure 34. The *storage preparation* algorithm

In the first compaction scheme, each T cell can cause one L cell of wasted space to be recovered by having some virtual L cells in its two subtrees merge during *storage movement*. Several T cells will cooperate in the compaction when the contents of several cells can be merged into one. Since merging is not performed on symbols in partially reduced applications, the cells in such virtual machines pretend to have no empty symbol registers available and so prevent merging occurring. Virtual L cells that are not in this situation consider registers like '<*' to be empty since more symbols can always be added to them. Completely empty L cells are handled as before and so are left out of the following description. The contents of two consecutive non-empty cells (possibly with empty cells between) can be merged when there are more empty symbol registers between the symbols of the two cells than there are registers in the template. This merging will create one new empty cell. Figure 35 shows the extended *storage management* algorithm. Each subtree transmits a quadruple that describes the symbol contents of

```
Given (V.l,L.l,R.l,E.l) and (V.r,L.r,R.r,E.r) arriving from the children,
 a T cell sends to its parent the message:

    V := V.l and V.r ;
    L := if (not V.l) then L.l else L.r ;
    R := if (not V.r) then R.r else R.l ;
    E := E.l + E.r + Gen
      where Gen := if (R.l + L.r ≥ registers in a template)
                   then 1   else 0


Given (ML,MR,Arr,Dep) arriving from the parent,
 a T cell generates two messages for its children:

    ML.l  :=  ML and (not V.l  or  Arr ≤ E.l) ;
    ML.r  :=  ML and not ML.l ;
    MR.r  :=  MR and (not V.r  or  Dep ≥ -E.r) ;
    MR.l  :=  MR and not MR.r ;

    Arr.l  :=  Arr ;
    Dep.r  :=  Dep ;
    Dep.l  :=  Arr.l - E.l - ML.l ;
    Arr.r  :=  Dep.r + E.r + MR.r ;

    if Gen = 1
    then if (Dep.l > 0)
         then ML.r := 1 ;  Arr.r := Arr.r - 1 ;
         else MR.l := 1 ;  Dep.l := Dep.l + 1 ;
```

(V,L,R,E) stands for (Vacant, Leftscraps, Rightscraps, Empty cells).
(ML,MR,Arr,Dep) stands for (Mergeleft, Mergeright, Arrivals, Departures).
The suffixes '.l' and '.r' indicate the left and right child of the T cell respectively.
'True' and 'false' have the values 1 and 0 for the purposes of performing arithmetic.

Figure 35. The *storage preparation* algorithm incorporating compaction

that subtree. *Empties* is the number of completely empty L cells that are available, including cells that compaction will produce within those trees. *Leftscraps* is the number of consecutive empty registers on the left of the first non-empty L cell's template. *Rightscraps* is the number of consecutive empty registers on the right of the last non-empty L cell's template. For an L cell with a template of '$(^{*}<^{*}$ **A** $>^{*})^{*}$' containing the symbols ">)", *leftscraps* is 4 and *rightscraps* is 1 (all registers are considered empty since all could accept more symbols). *Vacant* indicates that a subtree is completely empty, which is necessary to enable a T cell with one empty subtree to propagate to its parent the *leftscraps* and *rightscraps* from the other subtree. The *arrivals* and *departures* of the downsweep are augmented with the flags *mergeleft* and *mergeright*. *Mergeleft* indicates that the first incoming template should be merged into the first non-empty L cell in the subtree. Typically, the *mergeleft* flag arriving in a T cell propagates to the left subtree, and *mergeright* to the right subtree. If one subtree is empty then both flags go to the other. Any T cell which noted during the upsweep that it could recover an L cell, generates a new *merge* flag which is sent to that subtree into which symbols are flowing. This algorithm can be reorganised to allow pipelined evaluation in the T cells, that is, a T cell can be generating the messages for both children while receiving the message serially from its parent, rather than needing to wait until the entire message has been received.

Storage movement, which involves each physical L cell shifting virtual L cells according to the command it received during this *storage preparation* stage, is modified to perform merging. When a physical L cell receives one of the *merge* flags, it accepts the first virtual L cell before, rather than after, sending out its own virtual L cell. *Storage movement* in the L cells is extended to include the proper merging of the two templates into one. An L cell may receive a template to be merged from both neighbors.

Besides the disadvantages described above, the major disadvantage with this method is that, in order to guarantee compaction, flows must be generated that also remove all empty L cells from the middle of the FFP expression. The cost of *storage movement*, a major aspect of time delays in the FFP machine, depends on the largest distance that any single cell moves, which is greatly reduced by interspersed empty L cells as described in Section 3.4.2 and demonstrated elsewhere [Danforth 83, pp. 240-247]. While maintaining compaction, this method severely degrades the *storage management* process.

The following method for compaction avoids this disadvantage. This method adds a new system operation to the *system phase* to move the braces through the tree prior to *storage management*. Each T cell determines whether braces in one subtree could be moved to an L cell already holding symbols in the other subtree.

This method varies slightly depending on the particular template; it is described here for the template '$(^*<^* \mathbf{A} >^*)^*$'. For simplicity, it only compacts right braces. Left braces could be compacted in a distinct operation or the two operations could be combined so long as care were taken that separated brackets were properly converted into ϕ's.

Each subtree sends a message consisting of a *place* and a *request*. The *place* describes the contents of the rightmost non-empty L cell into which braces might be packed. There are three kinds of places. If that non-empty L cell contains parentheses, then it cannot accept a string of right braces that include brackets. If that cell does not contain parentheses, it can accept a sequence of right brackets followed by a sequence of right parentheses. The third kind of place describes a left subtree that is empty; it may be that braces from the right subtree can be compressed into L cells outside this physical subtree. The *request* consists of two numbers describing the braces that may be moved should an appropriate *place* be available. These braces are as many of the leftmost symbols in the tree that match the pattern "$>^i)^j$". There are two kinds of *requests*, those with no brackets, which are always accepted, and those with brackets, which are accepted when the *place* does not contain parentheses. The *request* includes a flag to indicate whether its braces are the only symbols in that tree. In those braces move for compaction, further braces which may be sent into the tree for compaction, are no longer destined for the right subtree.

Figure 36 demonstrates the T cells sending messages (down, as well as up) as the operation progresses upwards through the levels of the tree. L cells are shown grouped according to the physical tree structure. In each case, the request is shown moving upwards from each group of L cells. The T cells receiving these messages perform some compaction within their subtree and send a message to their parent. For example, the rightmost T cell in the first row receives a request containing one parenthesis from each of its L cells. It sends an '*Erase*' command to the right cell, one parenthesis to the left cell and a request to take two parentheses to its parent. The T cell next to it receives a request to take a bracket which it cannot satisfy given the parenthesis in the other L cell. It sends '*No change*' commands to its children and propagates the request to take the parenthesis that it received from its left child. The L cells in the second row show the effect of the commands sent by the lowest level T cells in the first row. The L cell which received a parenthesis receives a second command, '*Erase*', from the T cell at the second level. The final row of L cells shows the result of the overall operation.

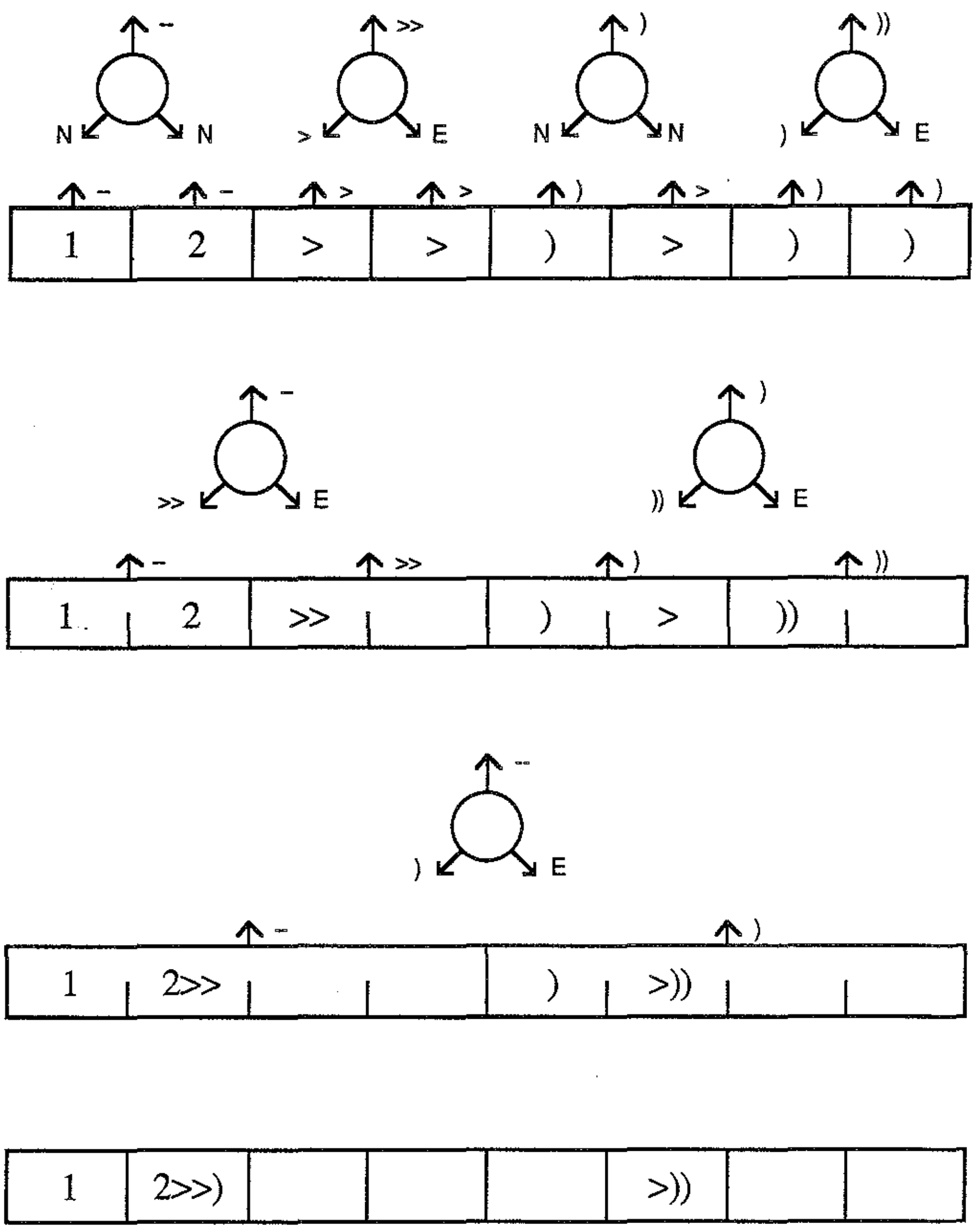


Figure 36. Compaction performed by moving braces through the tree

Both of these methods for performing compaction, either moving braces through the tree or during *storage movement*, are extremely complex and will significantly increase the complexity of the T cell and the period of the physical machine cycle.

Chapter 8
Summary of results

Having described separately in previous chapters the different issues concerning compressed representations, we now examine the combination of these issues by reviewing the approach of the research as described in Chapter 1. From these issues, a procedure for choosing a representation is derived which involves three steps: classifying symbols on the basis of their being sites for significant idioms and then choosing registers to hold these symbols and grouping those into a template. A catalog of templates is presented to demonstrate this procedure, and some concluding remarks are made about the use of significant instructions to direct the design of templates. Finally, this research on alternative program representations has provided insight into the meaning of processor granularity in parallel computers, in terms of the amount of computation performed by the processing elements.

8.1 Review of the research

An FFP function is implemented as manipulations on the expression tree that describes an application involving that function. The manipulations are expressed as a set of groups of idioms, the groups being executed concurrently at different sites in the expression tree. In the FFP machine, the groups of idioms are distributed over the L cells in such a way that each group is executed by the L cell holding the symbol occupying the site in the expression tree with which that group is associated. The generic FFP machine presented in Chapter 3 demonstrates this string reduction model of computation. To a large extent, idioms are determined by the needs of the functions, but there remains some flexibility in the particular way they are used. For example, in implementing the *ApplyToAll* function, one idiom was used to create the string ")([f]" instead of two idioms to create "([f]" and ")" separately. The L cell design does not affect the choice of idioms, but rather their implementation.

Examining several dozen FFP functions, of which half a dozen illustrative ones were presented in Chapter 4, yielded a few idioms that are used throughout the FFP language and beyond. Idioms used in individual FFP functions, because of their rarity, are a minor factor in evaluating the usefulness of program representations.

The usefulness of a program representation is measured by the space costs for holding expressions and the time and hardware complexity incurred in implementing the idioms and system operations. The time and hardware complexity costs are dominated by the use of two particular operations: communication and storage allocation. These operations use a large number of microcode instructions in each L cell, and microcode storage is the major aspect of hardware complexity. They also require the organised cooperation of many L cells rather than being local operations that a single cell can perform independently.

Some partial templates were presented in Chapter 5 which demonstrate the possible advantages gained from using different representations. The compressed representations reduce the number of L cells needed to hold FFP expressions. The optimal savings, given that an L cell may contain no more than one atom, occur when no brackets or parentheses require their own L cell. The packed brackets registers recover most of this space. More importantly, the compressed representations have reduced time costs in three ways. By allowing much simpler system operations, the physical machine cycle has become much shorter. By allowing many restructuring idioms to be performed without recourse to significant instructions, the number of machine cycles needed by many functions has been reduced. For example, *Compose* should now complete in a single cycle instead of the two it needed in previous versions of the FFP machine. Of less importance, simpler L cells, by requiring less circuitry, can be implemented with shorter delays.

It was shown in Chapter 5 that the choice of template affects the ease of designing the microcode segments. Microcode becomes simpler with the appearance of typed symbol registers in templates, such as a register that can only hold left brackets. Microcode becomes more complex if idioms require context dependent implementation, such as the idiom to create a left bracket given '(< **A** >)' as the template. The examples of Chapter 5 also indicate that, in order to avoid some serious disadvantages, representations should be natural, well-aligned, order-preserving and fine-grained.

The compressed representations allow a given FFP expression to have several distinct denotations in the FFP machine, and this variety may cause problems. For the representations that satisfy the constraints above, the denotations of FFP expressions will be unique if they use a minimal number of L cells. Chapter 6 examined the choices available in approaching the problem of scattered representations. Some approaches cause registers to be added to the template (the outer brackets registers), alter the choice of idioms to be used (the functional forms create the string ")>" instead of ">)"), and have a significant impact on the duration of the system phase of the physical machine cycle.

Program representation affects the low-level system operations of the various FFP machines, both in the system operations it requires be provided and in its effect on the implementation of those operations. These effects were examined in Chapter 7.

Care is necessary when a machine design is modified in answer to the principal costs during operation, since those modifications may also alter how the principal costs occur and so no longer be optimal solutions. Two examples of this effect have been seen in this research. As the template is altered to allow solutions to the problem of scattering, the ways in which scattering can arise change. As the template is altered to pack multiple brackets into one L cell, idioms which had needed significant instructions can be implemented without them. Since these idioms would come to reside in the same L cell by the packing of the brackets, such packing seems at first to be unacceptable; the modification to the template alters the problem at the same time as it appears to be causing it.

8.2 Templates and their consequences

Choosing a template involves three steps: classifying FFP symbols as *attachable* or *solitary*, organising appropriate symbol registers, and assembling symbol registers to form a template.

8.2.1 Classifying FFP symbols

The difference between attachable and solitary symbols arises in considering the groups of idioms that implement a (reasonably complete) set of FFP functions.

An FFP symbol is classified as *attachable* if no group of idioms that may be associated with that symbol (in the implementation of any function) requires significant instructions for its implementation. Several such symbols may reside in an L cell without allowing situations to arise during execution that compromise that cell's fine granularity.

A symbol is classified as *solitary* if there are groups of idioms associated with that symbol that require significant instructions in their implementation. A template that allows two solitary symbols to reside in the same L cell may allow situations to arise in which an L cell is required to perform more significant instructions than are necessary.

Significant instructions need only be associated with a symbol in the implementation of one FFP function for that symbol to be classified as solitary. This is because the amount of microcode storage in the L cell is the maximum of the sizes of the microcode segments needed for any function that the FFP machine must implement. The alternative to classifying that symbol as solitary because of a single function is to find an different implementation in which significant instructions are not associated with that symbol or to disallow that function as a primitive that can be implemented directly by the FFP machine.

From the templates in Chapter 5, it is clear that atoms should be classified as solitary, and brackets as attachable. The classification of parentheses is less obvious.

Implementing FFP's functional forms, such as *Compose* and *Insert*, associates idioms that require communication and storage allocation with the closing parenthesis. This can be seen by considering an L cell holding the symbols "5>))" in an RA involving *ApplyToAll*. This L cell now executes two independent *fork* instructions: one creates L cells to hold the parameter function to be duplicated before the '5' and the other creates the L cell needed to insert a bracket after the closing parenthesis. Parentheses can avoid becoming solitary because of *ApplyToAll* if the idiom '*create the string* ")([f]" *before* $[x_2]$ *through* $[x_n]$' is changed to '*create the string* ")([f]" *after* $[x_1]$ *through* $[x_{n-1}]$'. Imposing such limitations is likely to be acceptable for implementing FFP's functional forms, but restricts alternate uses of the FFP machine that call for more extensive manipulations of and around parentheses, such as might be the case, for example, in implementing normal order execution.

If the microcode segments are to maintain compaction, they will use communication to move the brackets surrounding the original RA (when there are single parentheses which are deleted). This communication is associated with the parentheses, and suggests they be classified as solitary.

In summary, classifying right parentheses as attachable constrains the implementation of functional forms and strongly suggests that microcode segments not perform all compaction. The left parenthesis can be classified as attachable more easily, since the computation, at run-time, of a function to be applied is rare.

The atom following a left parenthesis holds the function symbol of the RA. For primitive FFP functions, this symbol is always deleted and for the functional forms (proposed so far), care with the implementation can prevent this atom having significant idioms associated with it. Thus, the classification of symbols is context dependent; those atoms immediately following a left parenthesis can be classified as attachable. This allows the significant instructions used for compaction to avoid causing left parentheses to be classified as solitary. Templates may contain two solitary registers in this way so long as no FFP expression places two such solitary symbols in the same L cell.

The solitary versus attachable distinction seems to follow a semantic versus syntactic distinction. Brackets merely supply the structure to the atoms that convey the real information; parentheses, in indicating computations, fall in between. The choice in previous FFP machines of having one symbol per L cell is modified to having one atom per L cell, that is, a symbol that conveys value rather than structure. It is unclear whether there is a causal relationship between the classification of FFP symbols by associated significant idioms and the classification of FFP symbols by information content.

8.2.2 *Organising appropriate symbol registers*

FFP symbol registers are classified as solitary or attachable according to the classification of the symbols they hold. Attachable symbol registers allow the corresponding symbols to be created without storage allocation, which saves both time and space.

Packed symbol registers can contain many identical symbols. Packed registers must be assumed to be large enough to prevent overflow, or enough work may be added testing for overflow and then allocating space that such registers should be made solitary. This limits packed registers to holding various patterns of braces. The packed brackets registers, '$<^*$' and '$>^*$', provide most of the space savings found in this research. As important as the reduction in absolute space requirements, the reduction in the variation in space requirements during execution avoids storage allocation costs. Packed registers (usually) obviate specifying the exact position of the symbols being created or deleted. Incrementing or decrementing the register adds or removes an unspecified member of the group, as can be seen in the implementation of *transpose* where row brackets are deleted and created without reference to any relative level numbers. Of the packed parentheses registers, '$(^*$' and '$)^*$', the latter one is very useful for the *Compose* function, which by itself is a frequent part of FFP program expressions. The former register is suggested for symmetry, but would be as effective if limited to a single parenthesis.

If compactness is to be maintained entirely by microcode segments, an L cell must have access to the brackets that surround the RA. *Outer brackets registers* are brackets registers placed in the template so as to hold brackets adjacent to the parentheses (that is, brackets corresponding to sequence nodes immediately above an application node in the expression tree). Every function, on erasing the parentheses of the RA, checks whether either of the '$(^*$' or '$)^*$' registers has become empty. If so, the symbols in the adjacent outer brackets register must be moved to an L cell holding the result. This requires communication if that L cell differs from the L cell holding the outer brackets, which suggests that outer brackets registers are associated with solitary parentheses registers.

8.2.3 A catalog of templates and their consequences

Choosing a template involves assembling a sequence of symbol registers. A template may contain one solitary symbol register and as many attachable symbol registers as seem useful. The interactions among symbol registers can best be seen by considering the following templates.

8.2.3.1 '$(^* <^* \mathbf{A} >^*)^*$'

Parentheses have been defined as attachable, so this template favors *Compose* over the functional forms like *Insert*, *ApplyToAll* and *Construct*. The lack of outer brackets registers in this template prevents microcode segments maintaining compactness entirely on their own. This choice concerning compaction is appropriate since the parentheses are attachable.

8.2.3.2 ‘$(^* <^* \mathbf{R} >^*$’

R may contain either an atom or “$)^i$”. Right parentheses have been reclassified as solitary, which does not favor *Compose* over other functional forms, such as *Insert*, that may need to separate groups of consecutive parentheses. Whereas solitary symbols usually require storage allocation for their creation, *Compose* avoids this cost for creating closing parentheses because it creates new right parentheses at the site of the initial one, which can use the packed nature of the **R** register when that is behaving as ‘$)^*$’. *Compose* would not avoid these costs were **R** defined to contain either an atom or “)”. Functions that create parentheses at sites other than that of the initial right parenthesis do incur this added storage allocation. For example, *ApplyToAll* must allocate one more cell for each copy of “)([f]” under this template than under the one above.

The closing brackets register is an outer brackets register in the necessary circumstances, that is, when the **R** register contains a right parenthesis. Adding another ‘$<^*$’ register at the left of this template yields ‘$<^* \; (^* <^* \mathbf{R} >^*$’ which supports microcode segments maintaining compactness. This template avoids the difficulty shown in Section 6.3 of unavoidable scattering occurring with incorrectly placed parentheses, so long as the FFP program does no computation on function expressions. That is, the function expression in every application is a constant expression.

8.2.3.3 ‘$<^* \; (^* <^* \mathbf{A} >^*)^* >^*$’

This template is the first one above augmented with outer brackets registers. A minor factor for this template is that it contains seven registers, four of which are usually vacant: the parentheses registers and outer brackets registers will be empty in all L cells in the middle of RAs. Section 6.3 showed that microcode segments operating under this template can undo scattering associated with brackets but not parentheses. This extension of ‘$(^* <^* \mathbf{A} >^*)^*$’, intended to allow function implementations to perform compaction in the region of the RA’s parentheses, leads to the parentheses not being classified as attachable.

8.2.3.4 ‘$<^* \mathbf{P} >^*$’

P may contain either an atom, “$(^i$” or “$)^i$”. Only brackets have been classified as attachable. This template, considered with those above, demonstrates the contrasts between solitary and attachable registers. This template is relatively less useful for *Compose*. Although storage allocation is not needed to create the n right parentheses, despite their being solitary symbols, because **P** is a packed register when holding parentheses, creating n left parentheses, one before each $[\mathbf{f}_i]$, does require storage allocation. The difference is that the right parentheses are created at a point where a right parenthesis already occurs, so the packed parentheses register allows subsequent parentheses to be added. The left parentheses are added where none already exist, in the middle of the RA, and there is no attachable parenthesis register in the template of those L cells. An RA involving the function “< **ApplyToAll** + >” must allocate three times as many new cells to create copies of “)(+” as it would under ‘$(^* <^* \mathbf{A} >^*)^*$’.

This template would be appropriate for experimenting with mechanisms where arbitrary manipulations occurred around parentheses [Middleton and Smith 86]. Whether this template or ‘$(^* <^* \mathbf{A} >^*)^*$’ is more appropriate for implementing non-innermost reduction rules, such as normal-order evaluation, requires more study of the specific rule being considered. For standard FFP however, the only manipulations of parentheses (other than deletion) occur in the small fixed set of functional forms. More than the absolute space increase used to represent expressions with this template, its cost is the increased use of storage allocation that was avoided by other templates in the same situation.

Both brackets registers are outside ones and the parentheses are solitary, so this template is a good choice if microcode segments must maintain compactness. A minor advantage of this template is that the symbol registers are often occupied.

8.2.3.5 ‘$<^* (^* \mathbf{A})^* >^*$’

This template contains the same registers as the first one above, but in a different order. Whenever the function or operand of an RA is a sequence, the adjacent parenthesis will reside in a separate L cell. This template leads to layouts that are often identical with those resulting from the choice of ‘$<^* \mathbf{P} >^*$’ as template. The extra two registers are of little advantage, demonstrating that the assembly of the chosen symbol registers must reflect the structure of FFP expressions.

8.2.3.6 '$(^*\{^*<^*$ **A** $>^*\}^*)^*$'

The braces register, '$\{^*$', counts consecutive occurrences of "<(". The seven registers in this template will tend to be occupied sparsely. This appears to handle some needs of FFP well, but may be too specialised for broader uses. This template is particularly useful for the *Insert* functions which can create several consecutive copies of ")>" by adding the number to the '$\}^*$' register, as well as simplifying *ApplyToAll* and *Construct*. The manipulation of braces is relatively complex in comparison with manipulation of braces in previous templates, but it is still much simpler than the significant instructions. For example, erasing the closing parenthesis of an RA requires first checking the '$\}^*$' register. If it is empty, the '$)^*$' register is decremented; otherwise, the '$\}^*$' register is decremented and the '$>^*$' register incremented. On the other hand, the creation of a right bracket after the right parenthesis, an operation used by *ApplyToAll* among other functional forms, is difficult if the parenthesis is in the '}' register. This suggests a template of '$(^*\{^*<^*$ **B**' where '**B**' may contain either "**A** $>^i$" or "$\}^j)^k$". Implementing functional forms, such as *Insert*, *ApplyToAll* and *Construct*, is likely to be complicated by the need to handle multiple cases.

8.2.3.7 '$(^*$ **F** $<^*$ **A** $>^*)^*$'

In light of the discussion in Section 8.2.1 about the first atom in an RA being attachable, this template includes an attachable register for that symbol. Functions should avoid placing symbols in this register unless their result involves function applications, which restricts its use to the implementations of the functional forms. The FFP functions that create RAs, that is, the functional forms, must determine which register should contain the function symbol for the particular applications they are creating. The effects of this choice on the definition of expression trees are not difficult to handle.

The particular use for this template would be enabling *ApplyToAll* to avoid storage allocation in certain cases, namely those in which the parameter function is a single symbol. Large improvements can be gained if this function does not require storage allocation, as is demonstrated by considering a Navier-Stokes computation in which each element in a grid is updated to contain the average of its four neighbors. Specially defined atoms containing four numbers are initialised to contain the neighbors of each grid point using virtual machines in special ways [Middleton and Smith 86]. These values can be assembled in time proportional to the grid's width rather than the number of points. The straightforward *ApplyToAll* implementation wastes this advantage with its storage allocation costs likely being proportional to the number of elements in the grid. The modified *ApplyToAll* exploiting this template can create the computations to be run in single L cells and so achieve large improvements in this kind of situation.

8.2.4 Some remarks on significant instructions

Significant actions are an unstable criterion on which to base design decisions about the symbol registers that constitute a template. Adding two bracket registers to **S**, the template of the generic FFP machine, yields the template '$< \mathbf{A} >$', which leads to an FFP machine in which the L cells can incur more significant idioms and so are of larger granularity. Adding further storage to the template for brackets yields the template '$<^* \mathbf{A} >^*$', which is more fine-grained than either of the other two.

Even as the significance of the idioms that can be attached to them affects what registers can be placed in the template, the types of symbol registers available affect whether an idiom requires significant instructions in its implementation.

A similar effect occurs in the question of scattering. Even as the ways in which scattering can occur depend on the types of registers in the template, the approaches to scattering affect the choice of template registers.

Further study of the definition of *solitary* is necessary, since it fails to classify brackets adequately. On the one hand, brackets should be attachable because such templates displayed uniformly positive results. In particular, compressed brackets removed the need for significant actions to implement some frequent idioms, so that even as symbols were being compressed, significant actions were being removed. On the other hand, there remain idioms, such as duplicating one expression before another, which continue to use significant actions and are associated with the first symbol of an expression, which is often a bracket. This suggests that a bracket should be classified as solitary by the above definition. Perhaps there is some correlation which prevents different significant actions being associated with both a bracket and an adjacent atom, in the realm of reasonable FFP functions, or perhaps the brackets, having allowed the *firstL* and *lastL* addresses to be generated, are no longer important as the site of an idiom.

8.3 A program-related definition of fine granularity

This definition assumes a distributed multiprocessor model of parallel computers in which each processor has its own program and private memory (there is no global memory) and a computation consists of tasks that are performed by different processors. For the FFP machine, this multiprocessor corresponds to the virtual machine.

There are three factors that relate to the performance of a processor. The *responsibility* of a processor is the amount of the computation that it is to perform. *Responsibility* is measured by the significant operations the processor must perform for its task, significant operations being the ones that incur the majority of the time and complexity costs and which usually involve cooperation with other processors. For the FFP machine, these operations are indicated by the *fork* and *SendReceive* instructions in the microcode segments. The *view* of a processor is the amount of initial information available in its own memory; it can use this information to fulfill its *responsibility* without disturbing other processors. The initial information of an RA is just its symbols, so the *view* of an L cell is the FFP symbols it contains. The *capability* of a processor is its ability to generate its part of the result without affecting other processors. For example, the amount of local storage will affect the processor's need for secondary storage to hold intermediate results. The *capability* of an L cell relates to the attachable registers which allow it to generate symbols of the result without invoking storage allocation. Similarities exist between the *view*, the *capability* and the *responsibility* of processors, and the abilities to read, to create and to modify objects in the context of operating systems.

These three factors affect the performance of a processor in the following ways. In order to maximise the parallelism of a computation, it should be divided into several small tasks, that is, the *responsibility* of processors should be made small. This allows the processor to become smaller and simpler. To reduce the amount of communication a processor uses, the *view* should be made large, that is, the processor should be given a large amount of information about the computation. Also, the *capability* should be made large enough that the processor can represent its part of intermediate and final expressions without acquiring more resources.

These three factors are interdependent. A processor is responsible for all those tasks that use the information it contains. As it gains more information, that is, as its *view* increases, so does its *responsibility*. For the FFP machine, the *responsibility* of the L cells includes all the idioms associated with the symbols it contains, its *view*, so that increasing its *view* and *capability* will increase its *responsibility*. This tradeoff between maximising the *view* and *capability* and minimising the *responsibility* is the basis for choosing the granularity of a parallel computer.

Fine granularity is a property of parallel computers for which design decisions placed more importance on minimising their *responsibility* than on maximising their *view* or *capability*.

Early parallel computers suffered from serious bottlenecks in the communication between processors. It is possible that more recent designs have, as a result, stressed maximising the *view* of the individual processors in order to reduce the communication necessary between the processors. That recent designs often involve powerful microprocessors containing many tens of kilobytes of memory supports this conjecture. The discussion above suggests that such designs must impose a large *responsibility* on the individual processors, and this might explain the recurring difficulty that parallel computers have in achieving a high degree of parallelism over a wide range of computations. Furthermore, since a single computation might not use the full *responsibility* available in a processor, multiprocessing is added to the processor incurring the costs of the scheduling overhead.

Chapter 9
Conclusions and suggestions for further work

Language-directed design is an approach to building parallel computers that emphasizes programmability. It is a response to the widespread difficulty in using parallel computers designed for different primary goals. The FFP machine is one such parallel computer which has furthermore been designed around the run-time characteristics of individual programs as well the characteristics of a particular language. Flynn and Hennessy argue that the question of program representation becomes more important and complex with the transition from sequential computers to parallel ones because of the need to provide enough information that available parallelism can be recognised, while withholding information that overconstrains machine operation and so limits the parallelism that can be exploited [Flynn and Hennessy 79]. It is not surprising that language-directed parallel computers in general, and the FFP machine as a program-directed parallel computer design in particular, would be especially sensitive to the choice of program representation.

This research has investigated different ways of representing programs in the FFP machine. Even under constraints that restrict the possible representations to a narrow range, the consequences of the different representations are far-reaching. The number of machine cycles used to implement several common FFP functions was halved, the system operations that contribute to the length of the basic machine cycle were greatly simplified and the size of the processors (predominantly, their local storage) has been reduced. The performance of this parallel computer was shown to be extremely sensitive to the choice of program representation.

This research provided advantages at several levels. At the lowest level, several competing variants of the FFP machine were examined. Each variant FFP machine corresponds to a different representation in which the broad needs of machine operations may be accomplished in different ways. Individual machines outperform others in specific aspects of machine operation. For example, machines that include a ')*' register implement *Compose* better, and *ApplyToAll* slightly worse, than those without such a register. Several representations were designed in which large numbers of functions could be evaluated in fewer machine cycles; by avoiding storage allocation, these functions can be accomplished in a single cycle in contrast with the minimim of two cycles needed in previous FFP machines. Representations were designed which required significantly fewer L cells to hold program expressions, without contravening the fine-grained nature of the L cell design. The factors affecting performance of machines using particular representations have been organised into a procedure that can be used to aid the design of future FFP machines.

At an intermediate level, the repeated redesign of the same fundamental machine has shown the differences between the essential and the superficial aspects of various mechanisms supporting general machine operation. This better understanding is evident in improvements which have reduced the complexity of system operations for all FFP machines (in contrast with improvements that apply to specific variants). In particular, the T cell hardware associated with message processing has been reduced by a factor of four, the system algorithm that performs the *partitioning* operation has been significantly simplified, and the first correct executable implementation of the *directory* algorithm was constructed for these versions of the FFP machine.

This research has also suggested an approach to considering the granularity of the processors forming a parallel computer in relation to characteristics of programs and languages rather than in relation to hardware characteristics. In a distributed computer, there is a conflict between giving individual processors a large amount of information in order that they may perform more work autonomously and so reduce communication, and giving them a small amount to increase the parallelism being exploited. In this tradeoff, fine granularity chooses in favor of requiring minimal responsibility of the processor at the expense of providing little information.

A set of necessary characteristics for representations was developed that provides an accurate predictor of their utility. Being derived from the fundamental fine-grained and reconfigurable nature of the FFP machine design, this set of characteristics can be relied on to indicate the advantages and problems with new representations as they are proposed. To the extent that other parallel computers may share this nature with the FFP machine, these characteristics can also be used to direct the design of program representation in those computers.

At the highest level, in demonstrating one general-purpose parallel computer that is extremely sensitive to choices about program representation, this research provides a first piece of support for the conjecture that general-purpose parallel computer designs may be intrinsically very sensitive to choices of program representation. This is an important issue since parallel computer designs often do not consider the representation of programs among the initial design choices.

This research has raised several questions that form the basis for future research. The constraints on the representations limited them to a narrow range. Even over this narrow range, machine performance was shown to vary widely with respect to changes in program representation. This sensitivity suggests that further relaxing the constraints on allowed representations might lead to a wider variety of language-directed parallel computers with further improvements in performance.

The aspect of the research that was least amenable to investigation was that of unique representations in general, and scattering in particular. Since this appears to be the only potential disadvantage of these representations, the problem should be investigated further. In particular, further work might be done on its relation to the view, capability and responsibility of a processor and on improving algorithms to perform compaction by exploiting characteristics of the scattering process.

The concept of significant instructions is ill-defined, relying on a coarse attempt at quantifying costs in the FFP machine. As the factors contributing to time, space and complexity costs become better understood for the FFP machine, the definition of significant instructions and so in turn the concepts of responsibility and capability should also be refined.

Bibliography

S. Abramsky and R. Sykes "Secd-m: a Virtual Machine for Applicative Programming" Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag, LNCS 201. pp81-98. September 1985. Nancy, France.

T. Agerwala and Arvind "Data Flow Systems" IEEE Computer, Volume 15 No. 2. pp10-14. February 1982.

J. Backus "Programming language semantics and closed applicative languages" ACM Symposium on Principles of Programming Languages. pp71-86. October 1973. Boston.

J. Backus "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs" Communications of the ACM, Volume 21 No. 8. pp613-641. August 1978.

K.E. Batcher "Bit serial parallel processing systems" IEEE Transactions on Computers, Volume C31 No. 5. pp377-384. May 1982.

K.J. Berkling "Reduction Languages for Reduction Machines" Second Annual Symposium on Computer Architecture. pp133-140. January 1975. Houston, Texas.

K.J. Berkling "Computer Architecture for Correct Programming" Fifth Annual Symposium on Computer Architecture. pp78-84. April 1978.

P. Chen "Implementations of FFP functions on the Magó Machine" Internal document. 1981.

T.S. Clark "S-K Reduction Engine for an Applicative Language" Master's Thesis. University of Illinois at Urbana-Champaign. 1982.

T.J.W. Clarke, P.J.S. Gladstone, C.D. MacLean and A.C. Norman "SKIM - The S, K, I Reduction Machine" Proceedings of the 1980 LISP Conference. pp128-139. August 1980. Palo Alto, California.

D.P. Christman "Programming the Connection Machine" Thesis, Massachusetts Institute of Technology. April 1984.

S.H. Danforth "DOT, a distributed operating system model of a tree-structured multiprocessor" Ph.D. Thesis, University of North Carolina at Chapel Hill. 1983.

J. Darlington and M. Reeve "ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages" Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture. pp65-76. October 1981. Portsmouth, New Hampshire.

J.B. Dennis, G.A. Boughton and C.K. Leung "Building Blocks for Data Flow Prototypes" Seventh Annual Symposium on Computer Architecture. pp1-8. May 1980. France.

K.R. Dybvig (Implementing SCHEME on the FFP Machine) Ph.D. Thesis in preparation, University of North Carolina at Chapel Hill.

M.J. Flynn and J.L.Hennessy "Parallelism and representation problems in distributed systems" First International Conference on Distributed Computer Systems. pp124-130. October 1979. Huntsville, Alabama.

G.A. Frank "Virtual Memory Systems for Closed Applicative Language Interpreters" Ph.D. Thesis, University of North Carolina at Chapel Hill. 1979.

G.A. Frank, W.E. Siddall and D.F. Stanat "Virtual Memory Schemes for an FFP Machine" Proceedings of the International Workshop on High-Level Computer Architecture. pp8.37-8.45. May 1984. Los Angeles, California.

D. Gajski, D.J. Kuck, D. Lawrie and A. Sameh "Cedar - a Large Scale Multiprocessor" Proceedings of the 1983 International Conference on Parallel Processing. pp524-529. 1983.

L. Gilman and A.J. Rose "APL: An Interactive Approach" Third Edition, John Wiley and Sons. 1984.

H. Glaser, C. Hankin and D. Till "Principles of Functional Programming" Prentice/Hall International, 1984.

B. Goldberg and P. Hudak "Serial Combinators: Optimal Grains of Parallelism" Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag, LNCS 201. pp382-399. September 1985. Nancy, France.

A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer" IEEE Transactions on Computers, Volume C32 No. 2. pp175-189. February 1983.

J.R. Gurd, C.C. Kirkham and I. Watson "The Manchester Prototype Dataflow Computer" Communications of the ACM, Volume 28 No. 1. pp34-52. January 1985.

W.D. Hillis "The Connection Machine" MIT Press. 1985.

P. Hudak and B. Goldberg "Distributed Execution of functional programs using Serial Combinators" IEEE Transactions on Computers, Volume C34 No. 10. pp881-891. October 1985.

D. Kehs "A Routing Network for a Machine to Execute Reduction Languages" Ph.D. Thesis, University of North Carolina at Chapel Hill. 1978.

R.M. Keller, G. Lindstrom and S. Patil "A loosely-coupled applicative multiprocessing system" AFIPS Conference Proceedings. pp613-622. June 1979.

R.M. Keller and F.C.H. Lin "Simulated Performance of a Reduction-based Multiprocessor" IEEE Computer, Volume 17 No. 7. pp70-82. July 1984.

J.N. Kellman "Parallel Execution of Functional Programs" Master's Thesis. University of California at Los Angeles. 1983.

R.B. Kieburtz "The G Machine: A fast, graph-reduction evaluator" Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag, LNCS 201. pp400-413. September 1985. Nancy, France.

W.E. Kluge "Cooperating Reduction Machines" IEEE Transactions on Computers, Volume C32 No. 11. pp1002-1012. November 1983.

D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure and M. Wolfe "Dependence Graphs and Compiler Optimizations" ACM Symposium on Principles of Programming Languages. pp207-218. 1981.

J.T. Kuehn, T. Schwederski and H.J.Siegel "Design of a 1024-Processor PASM System" First IEEE International Conference on Supercomputing Systems. pp603-612. December 1985. St. Petersburg, Florida.

R.E. Ladner and M.J. Fischer "Parallel Prefix Computation" Journal of the ACM, Volume 27 No. 4. pp831-838. October 1980.

P.J. Landin "The mechanical evaluation of expressions" Computer Journal 6, p308. January 1964.

G.A. Magó "A network of microprocessors to execute reduction languages" International Journal of Computer and Information Sciences, Volume 8 Nos. 5 and 6. pp349-385 and 435-471. 1979.

G.A. Magó "A cellular computer architecture for functional programming" IEEE COMPCON. pp179-187. Spring 1980.

G.A. Magó "Copying operands versus copying results: a solution to the problem of large operands in FFP's" Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture. pp93-97. October 1981. Portsmouth, New Hampshire.

G.A. Magó "Data sharing in an FFP machine" 1982 ACM Symposium on LISP and Functional Programming. pp201-207. August 1982. Pittsburgh, Pennsylvania.

G.A. Magó and D. Middleton "The FFP Machine - A Progress Report" Proceedings of the International Workshop on High-Level Computer Architecture. pp5.13-5.25. May 1984. Los Angeles, California.

D. Middleton "Alternate program representation for the FFP Machine" Eleventh EUROMICRO Symposium on Microprocessing and Microprogramming. pp85-93. September 1885. Brussels, Belgium.

D. Middleton and B.T. Smith "FFP Machine Support for Language Extension" Nineteenth Hawaiian International Conference on System Sciences. pp59-66. January 1986. Honolulu, Hawaii.

R.P. Pargas and H.A. Presnell "Communication along shortest paths in a tree machine" Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture. pp107-114. October 1981. Portsmouth, New Hampshire.

R.P. Pargas "Parallel Solution of Recurrences on a Tree Machine" International Journal of Computer and Information Sciences, Volume 13 No. 4. pp251-277. 1984.

D.A. Plaisted (a) "An Architecture for Fast Data Movement in the FFP Machine" Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag, LNCS 201. pp147-163. September 1985. Nancy, France.

D.A. Plaisted (b) "An Architecture for Functional Programming and Term Rewriting" IFIP TC10 Conference on Fifth Generation Computer Architecture. July 1985. University of Manchester.

M. Pozefsky "Programming in Reduction Languages" Ph.D. Thesis, University of North Carolina at Chapel Hill. 1977.

H.A. Presnell (New communication mechanisms in the FFP Machine network) Ph.D. Thesis in preparation, University of North Carolina at Chapel Hill.

S.F. Reddaway "DAP - a distributed array processor" First Annual Symposium on Computer Architecture. pp61-70. 1974. Florida.

C.L. Seitz "The Cosmic Cube" Communications of the ACM, Volume 28 No. 1. pp22-33. January 1985.

D.E. Shaw "The NON-VON Supercomputer" Technical Report. Columbia University. August, 1982.

H.J. Siegel, T. Schwederski, N.J. Davis and J.T. Kuehn "PASM: A Reconfigurable Parallel System for Image Processing" ACM Computer Architecture News, Volume 12 No. 4. pp7-19. September, 1984.

B.T. Smith "Logic Programming on an FFP Machine" 1984 International Symposium on Logic Programming. pp177-186. February 1984. Atlantic City, New Jersey.

B.T. Smith (Logic Programming on an FFP Machine) Ph.D. Thesis in preparation, University of North Carolina at Chapel Hill.

V.P. Srini "An Architectural Comparison of Dataflow Systems" IEEE Computer, Volume 19 No. 3. pp68-88. March 1986.

D.F. Stanat and G.A. Magó "Optimal Storage Management in a Cellular Computer" Technical Report. University of North Carolina. 1981.

S.J. Stolfo and D.P. Miranker "DADO: A Parallel Processor for Expert Systems" Proceedings of the 1984 International Conference on Parallel Processing. pp74-82. August 1984.

W.R. Stoye, T.J.W. Clarke and A.C. Norman "Some Practical Methods for Rapid Combinator Reduction" 1984 ACM Symposium on LISP and Functional Programming. pp159-166. August 1984. Austin, Texas.

D.M. Tolle "Coordination of Computation in a Binary Tree of Processors: an Architectural Proposal" Ph.D. Thesis, University of North Carolina at Chapel Hill. 1981.

P.C. Treleaven, D.R Brownbridge and R.P. Hopkins "Data-Driven and Demand-Driven Computer Architecture" ACM Computing Surveys, Volume 14 No. 1. pp93-143. March 1982.

D. Turner "Combinator Reduction Machines" Proceedings of the International Workshop on High-Level Computer Architecture. pp5.26-5.38. May 1984. Los Angeles, California.

S.R. Vegdahl "A Survey of Proposed Architectures for the Execution of Functional Languages" IEEE Transactions on Computers, Volume C33 No. 12. pp1050-1071. December 1984.

E.H. Williams "Analysis of FFP Algorithms for Associative Searching" Ph.D. Thesis, University of North Carolina at Chapel Hill. 1981.

J. Williams "Notes on the FP style of Functional Programming" in "Functional Programming and its Applications". Edited by J. Darlington, P. Henderson and D.A. Turner, Cambridge University Press. pp73-101. 1982.

W.T. Wilner "Recursive Machine Principles of Operation" Xerox Palo Alto Research Center. LSI Systems Area Memo. September 1978.

W.T. Wilner "Recursive Machines" Xerox Palo Alto Research Center. Submitted to 1980 IFIP.