The FFP Machine

Technical Report 87-014

1987

Gyula A. Magó Donald F. Stanat

The University of North Carolina at Chapel Hill
Department of Computer Science
New West Hall 035 A
Chapel Hill, N.C. 27514

(This appears as a chapter in 'Topics in High-Level Language Computer Architecture,' edited by V. Milutinovic, published by Computer Science Press)

THE FFP MACHINE

Gyula Magó*
Donald F. Stanat
Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina

ABSTRACT

The FFP machine directly executes programs written in a general purpose high level functional programming language. It is a small-grain reduction machine which dynamically creates independent submachines of varying size consisting of conglomerations of processors for each available subcomputation. Because the machine performs problem decomposition and resource allocation without explicit programmer or software control, it is appropriate for dynamic and irregular computations. We describe an implementation that consists of a binary tree of only two kinds of processors; this implementation is extensible and well-suited to VLSI implementation.

I. INTRODUCTION

A highly parallel machine architecture has been developed and extensively studied at the University of North Carolina at Chapel Hill [Mago79,MagMi84]. The project has as its goal the construction of a machine with the following characteristics:

1. general purpose,
2. well-suited to VLSI technology,
3. extensible,
4. high performance.

The FFP machine holds promise for satisfying all these criteria. Detailed simulations have shown the design to be sound, and recent advances in microelectronic fabrication technology and asynchronous design make a successful implementation feasible. We discuss each of the above goals as it relates to the FFP machine.

1. **General purpose.** To achieve ease and generality of programming, the FFP machine will be programmed in (and will directly execute) the high-level general purpose FFP (Formal Functional Programming) language proposed by Backus [Backu78]. The machine design reflects the requirements of an FFP language interpreter rather than the requirements of a class of problems or a particular application area. FFP is a functional language, highly regular in its syntax, and with a simple fixed-point semantics. Its simplicity makes it well-suited to machine execution and it is an appropriate target language for translation from a variety of user languages. FFP, being a functional language, provides automatic synchronization among subcomputations; thus eliminating one of the major sources of error in parallel programs written in languages which require programmer-controlled synchronization.

* This was written while the first author was on leave at the Institute for Biomedical Computing, Washington University, St. Louis, Missouri. He gratefully acknowledges their support.

2. **Well-suited to VLSI technology.** Because chip design is costly but chip manufacture is not, VLSI technology is ideal for the production of many instances of a few designs. An FFP machine consists of thousands of cells (processors) of only two types, used in approximately equal numbers. The cells are small enough to allow many to be implemented on a single chip.
3. **Extensible.** A machine architecture is *extensible* (or *scalable*) if the design is insensitive to the size of the implementation. Ideally, a multiprocessor's computing capacity is directly proportional to the quantity of hardware; the FFP machine comes close to achieving this goal.
4. **High performance.** The design aims at high performance through parallelism rather than hardware speed. (That high performance can be achieved is demonstrable only with a prototype, but all forms of parallelism that we describe below are clearly present in the design we describe.) The machine exhibits MIMD parallelism at three distinct levels:
   a. *Multiprogramming.* The machine simultaneously and independently runs as many distinct programs as will fit in its memory.
   b. *Language-level parallelism.* In FFP, as in any functional language, the programmer does not specify parallelism explicitly; only data dependencies constrain parallelism, and the coordination of subcomputations is implicit in a program's syntactic structure. Because the FFP machine executes the language directly, initiation and termination of execution paths is automatic; there is no need for prior compilation or optimization for parallel execution. Because the machine initiates every computation for which all operands are available, parallelism within an FFP program is limited only by the size of the machine and by the data dependencies of the computation. (Operations cannot be performed until their operands are fully known.)
   c. *Parallelism within FFP primitives.* Considerable parallelism can occur in the execution of a single primitive operation in the FFP machine. For example, the machine can add a constant to each element of a vector in time independent of the size of the vector, and can sum the entries of a vector by adding partial sums pairwise in parallel.

Small granularity is a critical factor in the ability of the FFP machine to utilize simultaneously of all three kinds of parallelism listed above. Every data item involved in the computation is close to a processor (if only a simple one), and thus no scheduling problems arise. Small granularity also makes possible a solution to a persistent problem of parallel programming, the decomposition of a program into independently executable subprograms. (This 'program decomposition problem' is, in general, extremely difficult. We will describe later how it is solved in the FFP machine.) Decomposition is constrained in that a submachine (processor and memory) of adequate size must be assigned to each subproblem; if submachines are of fixed size, they must be sufficiently powerful to handle the largest subproblems that will occur. A small grain system has the potential to form submachines of an appropriate size from a number of small processors; this is the mechanism used by the FFP machine. The processors of the FFP machine are too limited in power to solve problems by themselves, but they can cooperate with (any number of) neighboring processors. Thus program execution involves the formation of an arbitrarily large (up to the size of the machine) conglomeration of cooperating cells for each subproblem that arises in a computation.

II. FFP LANGUAGES

Because an FFP language is the machine language of the FFP machine, an understanding of the machine requires some knowledge of these languages. This section informally describes a specific FFP language only to the extent necessary to understand this chapter; the reader should consult the reference [Backu78] for a complete treatment.

II.1 Language description

John Backus proposed the FP and FFP programming languages in his Turing Award Lecture [Backu78]. FP languages are designed for human use, and thus contain considerable "syntactic sugaring"; FFP languages, in contrast, have a highly regular syntax well suited to machine execution.

An FFP language is a language of expressions that represent functions and values. A program is an expression that represents a function; it produces an output (the function value) when applied to an input (the function argument). Syntactically, a program expression f is combined with a data expression x to form an expression $(f : x)$ called an *application*; f is called the *operator* and x the *operand* of the application. The value of $(f : x)$ is the output of the program f run on the input x.

Executing the program f on the input x consists of evaluating the expression $(f : x)$. Program execution proceeds by *rewriting* (or *reducing*) the expression. This is done by selecting one or more disjoint subexpressions and replacing them with new subexpressions that have the same value, much as one would do in finding the value of an arithmetic expression involving operators such as + and *. FFP restricts the rewriting process so that only innermost applications are rewritten (this corresponds to a *call-by-value* semantics). The reduction process terminates when there are no more subexpressions that can be rewritten; the resulting expression, which contains no applications, is called an *object*.

The simplest FFP programs consist of an atom that represents a language primitive operator; an example is the symbol +, which is an atom (as is the numeral 6 and the boolean constant TRUE), but also, when it appears as an operator in an application, denotes a function that will sum the entries of a numeric sequence. The program + is run on the input sequence $< 4,6,8 >$ by evaluating the expression $(+ :< 4,6,8 >)$. This expression can be rewritten (according to the language definition, and in particular, the definition of +) as the sum of 4, 6 and 8; that is, the language definition states that the value of $(+ :< 4,6,8 >)$ is the same as the value of 18. We show a single reduction with an arrow $\rightarrow$, and a sequence of 0 or more reductions with $\stackrel{*}{\rightarrow}$; thus,

$$(+ :< 4,6,8 >) \rightarrow 18$$

The operator +, like all functions in FFP, is monadic; every FFP function takes exactly one operand, which can be an atom (such as the integer 4, the real number 3.425, or the character string 'cat') or a finite sequence, such as the empty sequence (denoted by ϕ) or the sequence of two elements $< 3, < 5, 'cat' >>$. Other primitive functions that will be used in this paper are described in Table 1.

(We will often not distinguish between an atom such as + and the function associated with that atom, in this case, variadic addition. Thus we will speak of 'the function +' when no confusion will arise.)

If only primitive functions were allowed, FFP languages would be very weak; one also needs a way to combine small programs into larger ones. In FP, program-forming operations are given as functional forms; in FFP, they are given as sequences whose meaning is defined by *metacomposition*. When the operator of an application is a nonempty sequence

object (a sequence that contains no applications), the expression is rewritten using the metacomposition rule as follows:

$$(< x, y_1, y_2, ..., y_n >: z) \rightarrow (x :<< x, y_1, y_2, ..., y_n >, z >)$$

The metacomposition rule rewrites an application as a new application. The operator of the new application is the first element of the original operator sequence; the operand of the new application is the pair consisting of the original operator and the original operand. Metacomposition makes possible powerful program-combining operations including the functional forms that Backus described for FP. Examples of the use of metacomposition are given in Table 2.

A subroutine facility is provided in FFP by user-supplied *definitions*. A programmer can assign to any atom a definition, which must be an object. When a defined atom appears in the operator position of an innermost expression, it is replaced by its definition. An example of a defined atom is *IP* (for 'inner product'):

$$IP \equiv < CMP, +, < ATA, * >, TR >$$

The following is an example of the application of IP:

$(IP :<< 1,2,3 >, < 3,4,5 >>) \rightarrow (< CMP, +, < ATA, * >, TR >:<< 1,2,3 >, < 3,4,5 >>)$
by definition of IP
$\xrightarrow{*}(+ : (< ATA, * >: (TR :<< 1,2,3 >, < 3,4,5 >>)))$ by definition of CMP (Table 2)
$\rightarrow (+ : (< ATA, * >:<< 1,3 >, < 2,4 >, < 3,5 >>))$ by definition of TR (Table 1)
$\xrightarrow{*}(+ :< (* :< 1,3 >), (* :< 2,4 >), (* :< 3,5 >) >)$ by definition of ATA (Table 2)
$\rightarrow (+ :< 3,8,15 >)$ by definition of $*$ (Table 1)
$\rightarrow 26$ by definition of $+$ (Table 1)

Because innermost applications are the only ones rewritten by the reduction process, they are called *reducible applications* (RAs). Note that

i. distinct RAs are disjoint,

ii. each RA contains all its operands, and

iii. rewriting an RA has no side effects.

It follows that all RAs of an expression can be rewritten concurrently and independently. Nesting of applications reflects data dependencies; an application can be rewritten if and only if none of its subexpressions is an application. Consequently, the number of RAs in an expression is equal to the level of parallelism at the FFP level that can be used in rewriting the expression.

Giving a careful description of an operational semantics (how FFP expressions are evaluated on the FFP machine) requires that we add a few details to the foregoing. A particular FFP language is completely characterized by specifying a set of *atoms* and a set of *primitive functions*, each of which is associated with a unique 'primitive' atom. Each primitive function is a map from the set of objects to the set of FFP expressions.

There are four kinds of FFP expressions: atoms, sequences, applications and bottom ($\perp$). The expression $\perp$ represents a value that is undefined, or it represents a nonterminating computation. It is an object but not an atom, sequence, or application. Any sequence expression in which $\perp$ occurs as an element is defined to be equal to $\perp$; such a sequence is considered to be simply an alternative notation for $\perp$.

If $x_1, \ldots, x_n$ are expressions other than $\perp$, then $< x_1 \ldots x_n >$ is a *sequence*; the *length* of the sequence is n, and x_i is the i^{th} *element* of the sequence. The *empty sequence* (of length 0) is denoted by ϕ; ϕ is both an atom and a sequence.

The *value*, or *meaning*, of any object is itself. The value of a sequence $< x_1, x_2, \ldots, x_n >$ is the sequence of values of its elements, unless the value of some of its entries is $\perp$, in which case the value of the expression is $\perp$.

The preceding specifies the value of all expressions except applications. The value of $(f : x)$ can be given denotationally with a fixed-point semantics [Backu78], but we will describe an operational semantics.

Operationally, evaluation of an expression e proceeds by successively reducing (rewriting) the expression e as another expression e' which has the same value. Each step of the rewriting process replaces one or more RAs by expressions with the same meaning. If this process terminates, then the final expression is an object o, and the value of e is defined to be o. If the process does not terminate, then the value of the expression is defined to be $\perp$.

The way in which an RA $(f : x)$ is rewritten can be described by cases. Since an RA does not contain any proper subexpressions that are applications, f and x are both objects; hence f is either an atom, or a sequence, or $\perp$. An RA is rewritten by applying the first applicable clause of the following sequence.

1. If f is a defined atom, then there is a (programmer-specified) definition in the form of an FFP object associated with f. (Recall that such definitions are analogous to subroutines and are not part of the language definition.) If f is defined by the expression e, then $(f : x)$ is rewritten as $(e : x)$.
2. If f is a primitive atom, then there is a primitive (built-in) function denoted by f. (A primitive function is part of the language definition; examples of such functions include the arithmetic and boolean functions as well as some that manipulate sequences.) In this case, $(f : x)$ is rewritten as the value of the function denoted by f applied to x.
3. If f is an atom that is neither defined nor primitive, or if f is $\perp$, then the expression $(f : x)$ is rewritten as $\perp$.
4. If $f =< f_1 \ldots f_n >$, where $n \geq 1$ (and no $f_i = \perp$), then the RA is rewritten according to the metacomposition rule:

$$(< f_1 \ldots f_n > : x) \rightarrow (f_1 :<< f_1 \ldots f_n >, x >).$$

With an appropriate set of primitive functions, an FFP language can specify any computable function. It follows that the program expression may expand as well as contract during execution, and may, in fact, grow without bound. The existence of the *apply* primitive function (see Table 1) means that data can be made executable; thus higher-order functions can be defined.

II.2 FFP on the FFP machine

In the sequel we will use the FFP language described above in Tables 1 and 2; it is nearly identical to that described by Backus [Backu78] except some of our primitive functions permit more efficient computations than would be possible otherwise.

We will sometimes distinguish between language primitive operators (which can be anything that can be effectively defined) and machine primitive operators (which must be implementable on the FFP machine). We will refer to these as language primitives and machine primitives respectively.

Section III.5 describes how FFP operations are implemented on the FFP machine.

Selector functions
For any positive integer s,
$(s:x) \equiv x =< x_1, x_2, \ldots, x_n > \& \ 1 \leq s \leq n \Rightarrow x_s; \perp$

Tail
$(TL:x) \equiv x =< x_1 > \Rightarrow \phi; x =< x_1, x_2, \ldots, x_n > \Rightarrow < x_2, \ldots, x_n >; \perp$

Identity
$(ID:x) \equiv x$

Atom
$(ATOM:x) \equiv x$ is an atom $\Rightarrow$ TRUE; $x \neq \perp \Rightarrow$ FALSE; $\perp$

Equals
$(EQ:x) \equiv x =< y, z > \& y = z \Rightarrow$ TRUE; $x =< y, z > \& y \neq z \Rightarrow$ FALSE; $\perp$

Null
$(NULL:x) \equiv x = \phi \Rightarrow$ TRUE; $x \neq \perp \Rightarrow$ FALSE; $\perp$

Length
$(LENGTH:x) \equiv x = \phi \Rightarrow 0; x =< x_1, x_2, \ldots, x_n > \Rightarrow n; \perp$

Reverse
$(REV:x) \equiv x = \phi \Rightarrow \phi; x =< x_1, x_2, \ldots, x_n > \Rightarrow < x_n, \ldots, x_2, x_1 >; \perp$

Rotate-left, rotate-right
$(ROTL:x) \equiv x = \phi \Rightarrow \phi; x =< x_1, x_2, \ldots, x_n > \Rightarrow < x_2, \ldots, x_{n-1}, x_n, x_1 >; \perp$
$(ROTR:x) \equiv x = \phi \Rightarrow \phi; x =< x_1, x_2, \ldots, x_n > \Rightarrow < x_n, x_1, \ldots, x_{n-1} >; \perp$

Distribute-from-left, distribute-from-right
$(DISTL:x) \equiv x =< y, \phi > \Rightarrow \phi; x =< y, < x_1, x_2, \ldots, x_n >> \Rightarrow << y, x_1 >, < y, x_2 >, \ldots, < y, x_n >>; \perp$
$(DISTR:x) \equiv x =< \phi, y > \Rightarrow \phi; x =<< x_1, x_2, \ldots, x_n >, y > \Rightarrow << x_1, y >, < x_2, y >, \ldots, < x_n, y >>; \perp$

Append-to-the-left, append-to-the-right
$(APNDL:x) \equiv x =< y, \phi > \Rightarrow < y >; x =< y, < x_1, x_2, \ldots, x_n >> \Rightarrow < y, x_1, x_2, \ldots, x_n >; \perp$
$(APNDR:x) \equiv x =< \phi, y > \Rightarrow < y >; x =<< x_1, x_2, \ldots, x_n >, y > \Rightarrow < x_1, x_2, \ldots, x_n, y >; \perp$

Transpose
$(TR:x) \equiv x =< \phi, \phi, \ldots, \phi > = \phi; x =< x_1, x_2, \ldots, x_n > \Rightarrow < y_1, y_2, \ldots, y_m >; \perp$
where
$x_i =< x_{i,1}, x_{i,2}, \ldots, x_{i,m} >$ and $y_j =< x_{1,j}, x_{2,j}, \ldots, x_{n,j} >, 1 \leq i \leq n$ and $1 \leq j \leq m$.

Apply
$(AP:x) \equiv x =< y, z > \Rightarrow (y:z); \perp$

Arithmetic operators
$(+:x) \equiv x = \phi \Rightarrow 0; x =< x_1, x_2, \ldots, x_n >$ & every x_i is a number $\Rightarrow x_1 + x_2 + \ldots + x_n; \perp$
$(*:x) \equiv x = \phi \Rightarrow 1; x =< x_1, x_2, \ldots, x_n >$ & every x_i is a number $\Rightarrow x_1 * x_2 * \ldots * x_n; \perp$

Table 1: Definitions of selected primitive FFP functions. The definitions are given as a case statment on the argument x. In each definition, first condition that holds for the argument x specifies (with a $\Rightarrow$) the applicable clause of the definition; the last clause (usually $\perp$) specifies the value if none of the conditions holds. All the functions listed are implementable as machine primitives.

Composition:
$(< CMP >: x) \rightarrow x$
$(< CMP, f_1, f_2, \ldots, f_n >: x) \rightarrow (f_1 : (f_2 : (\ldots (f_n : x) \ldots)))$

Construction:
$(< CON >: x) \rightarrow \phi$
$(< CON, f_1, f_2, \ldots, f_n >: x) \rightarrow < (f_1 : x), (f_2 : x), \ldots, (f_n : x) >$

Apply-to-all
$(< ATA, f >: \phi) \rightarrow \phi$
$(< ATA, f >:< x_1, x_2, \ldots, x_n >) \rightarrow < (f : x_1), (f : x_2), \ldots, (f : x_n) >$

Conditional
$(< COND, p, f, g >: x) \rightarrow (< CN, (p : x), f, g >: x)$
$(< CN, TRUE, f, g >: x) \rightarrow (f : x)$
$(< CN, FALSE, f, g >: x) \rightarrow (g : x)$

Insert (Operand must be a nonempty sequence.)
$(< INSERT, f >:< x_1 >) \rightarrow x_1$
$(< INSERT, f >:< x_1, x_2, \ldots, x_n >) \rightarrow (f :< x_1, (f :< x_2, \ldots, (f :< x_{n-1}, x_n >) \ldots >) >)$

Constant
$(< CONST, c >: x) \rightarrow c$

Binary-to-unary
$(< BU, f, x >: y) \rightarrow (f :< x, y >)$

Apply-to-the-right (Operand must be a nonempty sequence.)
$(< AR, f >:< x_1, x_2, \ldots, x_n >) \rightarrow < x_1, x_2, \ldots, (f : x_n) >$

Table 2: Rewriting of selected functional forms using composite functions. Although the metacomposition notation is used, expressions will be rewritten in a single step as shown above (rather than using the metacomposition rewriting rule) if the composite function operator has been implemented as a machine primitive. Not explicit in the above is that any application which does not match one of the templates or in which the argument x is $\perp$ will be re-written as $\perp$.

III. THE MACHINE DESIGN

The FFP machine is designed for the direct, reduction-style execution of programs written in the FFP language. Reduction is a simple and straightforward process when viewed in the abstract, but it raises many problems when it is to be implemented on a multiprocessor. For example, the number of reducible applications (RAs) can vary rapidly, creating problems of control (e.g., which RAs are to be evaluated first) and resource allocation (e.g., how to allocate newly created RAs to processors). The expression can also shrink and expand unpredictably, creating problems of storage management.

If one defines the parts of the expression within RAs as *active*, and those parts outside RAs as *inactive*, then the ratio of the sizes of the active and inactive parts of the expression can also change rapidly during execution. Since each RA must be rewritten, the active parts of the expression require processing. In the extreme case, nearly all of the expression can be active, which suggests a multiprocessor organization in which processing capabilities are distributed over memory rather than being separated from it, as with collections of von Neumann computers. Klaus Berkling may well have been the first to recognize this when he observed in 1975: "The reduction concept seems particularly suited for a 'processing in memory.' But for the time being, appropriate hardware concepts are not available" [Berkl75].

The FFP machine project seeks to demonstrate that reduction style execution can be done efficiently via processing in memory. A system capable of processing in memory is usually called a *logic-in-memory* system, a *smart memory* or a *small-grain multiprocessor*. We shall use the last term for the FFP machine.

III.1 A small-grain multiprocessor

The FFP machine as shown on Figure 1 consists of a linear array of cells and an interconnection network, which also performs a variety of processing functions. The cells are called L cells (for *linear* or *leaf* cells). Each L cell corresponds to a location in a conventional memory (its capacity may be several bytes) except that it also contains some processing capability, e.g., a simple ALU and several registers. Therefore, a useful machine must contain at least thousands of L cells.

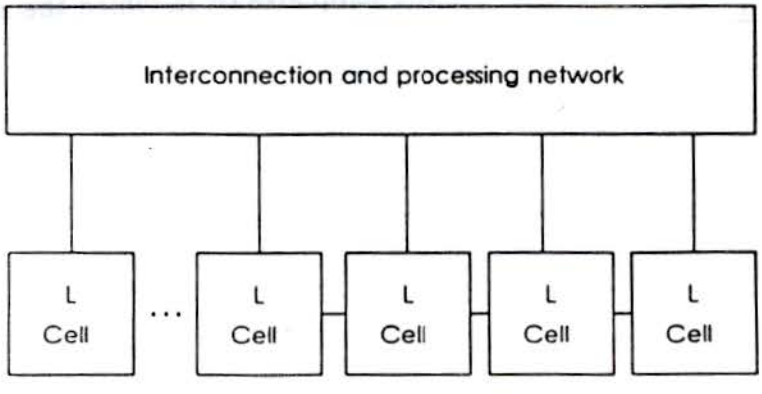


Figure 1: Block diagram of the FFP machine. The L cells are both storage and processing elements; the interconnection network provides both communication and processing.

III.2 Representation of program and data in the FFP machine

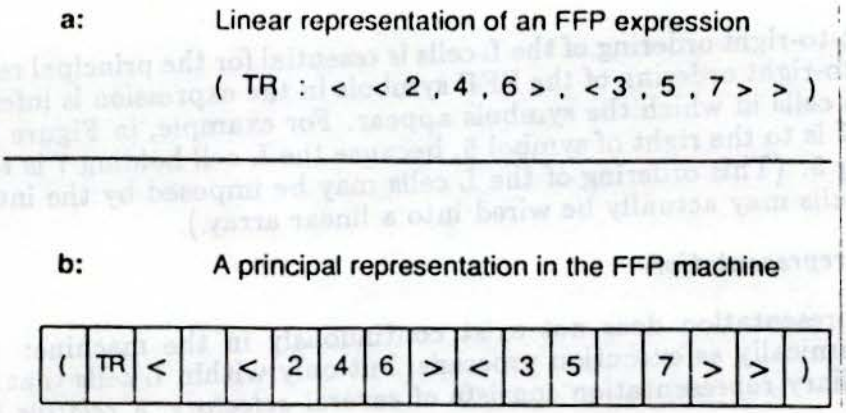


Figure 2: The linear representation of an FFP expression and a typical principal representation of the expression in the FFP machine.

The representation of FFP expressions in the FFP machine consists of a *principal* representation and an *auxiliary* representation. The principal representation exists continuously, and changes only when the expression changes. The auxiliary representation is transient: it is constructed for a subexpression only when it becomes an RA, and is erased when the reduction of that RA is complete.

III.2.1 Principal representation

The principal representation of an FFP expression is essentially the linear, parenthesized representation of the FFP expression normally used to write it on paper. (Other linear representations could also be used, such as those obtained by traversing the tree form of the expression in various orders.)

This linear representation contains information about the structure of the expression both in the syntactic markers (i.e., brackets and parentheses) and in the left-to-right order of the symbols. For example, a two-dimensional array in this representation is most commonly laid out in row-major order with brackets around the rows and the whole array. Figure 2.a shows an FFP expression consisting of an application of the transpose operator to a two-by-three array of integers given in row-major order.

The principal representation is obtained by mapping the linear representation of the FFP expression onto the L array, maintaining the left-to-right ordering of the symbols. A variety of principal representations are possible, but they all share the following characteristics:

a. The left-to-right ordering of the symbols in the FFP expression is preserved.

b. Commas and colons, used as separators between FFP symbols when the expression is written on paper, are omitted since the cell boundaries act as separators between symbols.

c. Blank L cells may appear outside or within the image of an FFP expression; thus the image of an expression may be located anywhere in the L array, and it need not occupy contiguous cells.

It follows from (c) that the principal representation of an expression is not unique.

Principal representations differ in what sequences of FFP symbols can be held by a single L cell. Most of our discussion of the machine will be based on the particularly simple representation that allows only one FFP symbol in each L cell, as shown in Figure 2.b. Some alternative principal representations will be discussed in Section III.2.3.

Note that the left-to-right ordering of the L cells is essential for the principal representation, because the left-to-right ordering of the FFP symbols in the expression is inferred from the ordering of the L cells in which the symbols appear. For example, in Figure 2.b, we know that the symbol 7 is to the right of symbol 5, because the L cell holding 7 is to the right of the L cell holding 5. (This ordering of the L cells may be imposed by the interconnection network, or the cells may actually be wired into a linear array.)

III.2.2 Auxiliary representation

The auxiliary representation does not exist continuously in the machine: the machine constructs it dynamically as execution proceeds, but only within L cells that hold part of an RA. The auxiliary representation consists of several *selectors*, a *relative level number* and an *index*.

Selectors are explained with the help of Figure 3, which shows the expression of Figure 2.a represented as an ordered tree. Each node represents a unique well-formed subexpression; the root represents the entire expression, and leaves represent atoms. The children of each internal node are ordered; the i^{th} child of a node represents the i^{th} maximal well-formed proper subexpression of the expression represented by that node. Internal nodes of the tree are labelled with the (pair of) syntactic markers that enclose the expression; leaves are labelled with the atomic expression they represent. In our diagrams the branches from internal nodes are labelled with integers; the branch from a node to its i^{th} child is labelled i. Each subexpression is uniquely specified by the sequence of labels of the edges of the path from the root to the node that represents the subexpression. Thus, for example, the operator TR corresponds to the path labelled (1), the sequence subexpression $< 3, 5, 7 >$ corresponds to the path labelled (2,2) and the atom 4 corresponds to the path labelled (2,1,2). These branch labels, called *selectors*, form the first part of the auxiliary representation.

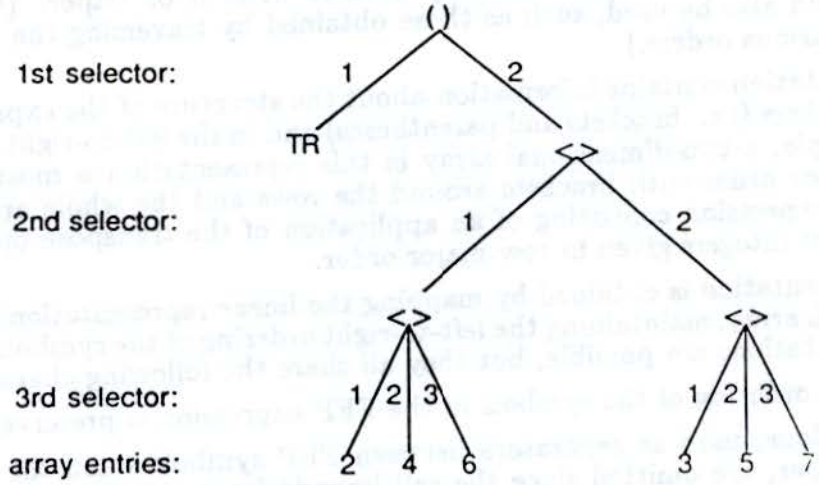


Figure 3: The FFP expression of Figure 2 in tree form. Edges of the tree are labelled with selectors values.

Figure 4 shows a principal and auxiliary representation in the FFP machine of the RA expression in Figure 3. Note that every symbol of the RA is associated with a selector sequence of length three specifying the first steps along the path from the root of the RA expression to the FFP symbol held by the L cell. The occurrence of 0 in a selector sequence denotes the end of the path; thus, the selector sequence (2,0,0) denotes the path (2) of length one.

PRINCIPAL:	(	TR	<	<	2	4	6	>	<	3	5	7	>	>	)
AUXILIARY 1st selector:	0	1	2	2	2	2	2	2	2	2	2	2	2	2	0
AUXILIARY 2nd selector:	0	0	0	1	1	1	1	1	2	2	2	2	2	0	0
AUXILIARY 3rd selector:	0	0	0	0	1	2	3	0	0	1	2	3	0	0	0
AUXILIARY rln:	0	1	1	2	3	3	3	2	2	3	3	3	2	1	0
AUXILIARY index:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 4: Principal and auxiliary representation of the FFP expression given in Figure 2. The representation shown uses three selectors.

Storing only k selectors in each L cell ($k = 3$ in Figure 4) means that the machine can use selectors to identify unique subexpressions only on the top $k+1$ levels of the RA expression tree, but none of the subexpressions further below. For example, if in Figure 4 the FFP atom 7 was replaced with a sequence of FFP symbols, then each symbol of that sequence would have the same selector sequence. Since it is not feasible to store an unbounded number of selectors in an L cell, it follows that selectors alone can not distinguish and manipulate the individual elements of deeply nested expressions. This deficiency (resulting from having only a limited number of selectors) is largely overcome by the other two parts of the auxiliary representation. These two fields, shown in Figure 4, are called the *relative level number* and the *index*.

The index provides a left-to-right numbering (starting at the left end of the RA) of all the symbols of the RA. The relative level number (RLN) of a symbol is its nesting level assuming the application parentheses have zero as their nesting level.

III.2.3 Compressed representations

The representation of FFP expressions used in Figures 2.b and 4 is not the only possible principal representation. One alternative, shown in Figure 5, permits more than one FFP symbol in an L cell, and is called a *compressed* representation.

	PRINCIPAL:	(TR	<<2	4	6>	<3	5	7>>)
AUXILIARY	1st selector:	1	2	2	2	2	2	2
	2nd selector:	0	1	1	1	2	2	2
	3rd selector:	0	1	2	3	1	2	3
	rln:	1	3	3	3	3	3	3
	index:	1	2	3	4	5	6	7

Figure 5: A compressed principal and auxiliary representation of the FFP expression of Figure 2.

Tolle was the first use a compressed representation [Tolle81a,Tolle81b] and Middleton has studied a variety of them and described the far-reaching effects they have on the whole FFP machine [Middl86]. Middleton found that a good compressed representation should have no more than one atomic symbol in an L cell, because their processing requirements are much higher than those of syntactic markers.

Figure 5 uses a compressed representation (one among many possibilities) in which an arbitrary L cell contains information according to the following pattern:

$$< * (* < * \mathit{ATOM} > *)* > *$$

where the asterisk (the Kleene star operator) indicates zero or more occurrences of the preceding symbol, and ATOM denotes either no symbol or any atomic FFP symbol. Thus, the symbol string " < (< < *TAIL*" can then be stored in a single L cell, and would be represented by the seven tuple 1,1,2,*TAIL*,0,0,0.

In compressed representations, there will be one selector sequence, relative level number and index per occupied L cell; it belongs to the atomic symbol held by the L cell if one is present, and otherwise (somewhat arbitrarily) to the innermost bracket or parenthesis held by the cell.

III.2.4 The FFP machine as memory

The FFP machine is a logic-in-memory system designed to do most of its processing in memory; its L cells correspond to memory locations, each with built-in processing capability. One may, therefore, ask what properties the FFP machine has as a memory device (in addition to being viewed as a parallel processing device).

Since the FFP machine does not use physical addresses to identify its L cells (which are its memory locations), it is legitimately viewed as a content-addressable (or associative) memory. However, the representation used in the machine differs substantially from those in other associative machines. Maintaining the left-to-right order of FFP symbols of the principal representation can be viewed as a limited use of physical addresses, because the

order of FFP symbols is determined by the order of the L cells that hold them. In other words, although addresses are not used to identify the L cells, the linear ordering of the L cells is used in the operation of the machine. This is in contrast with usual associative memories, in which all locations are completely interchangeable.

Maintaining left-to-right order of FFP symbols also means that the contents of a location (L cell) can be relocated only to certain other locations without changing the existing left-to-right order and thereby changing the structure of the expression. The fact that information is not as freely relocatable in this machine as in other associative memories may also be expressed by saying that the contents of the FFP machine are not fully "self-describing" as the contents of associative memories usually are. (The left-to-right order of symbols carries information not otherwise represented.)

III.3 Overall machine structure

The four major components of a complete FFP machine are shown in Figure 6: the linear array of L cells, the interconnection network (which also performs certain processing functions), a front-end machine and auxiliary memory.

The representation of FFP expressions in a linear array of small-grain processing elements is the cornerstone of the FFP machine; a variety of computing machines could be constructed on this foundation. This chapter will describe the variant currently pursued by the FFP machine project, which aims to be as simple as possible while still demonstrating the advantages of machines of this kind. Possible alternatives for some of the design choices will be pointed out throughout this chapter.

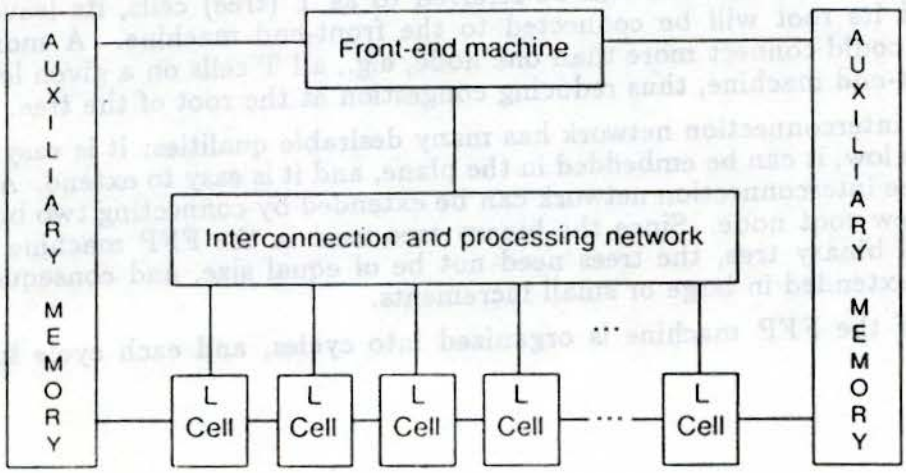


Figure 6: Block diagram of the FFP machine showing its connections with auxiliary memory and the front-end machine.

The linear array of L cells of the "logic-in-memory" system not only holds the FFP expression (program and data together), but also carries out most of the processing done by the machine. Thus, an L cell serves not only as a memory location, but also as a processing element (PE). Adjacent L cells in Figure 6 are connected to form a linear array. These connections are used only for purposes of storage management, described in Section III.6.

The front-end machine is in overall control. It contains the definitions of the FFP primitive operations that the FFP machine uses, controls the auxiliary memory devices, and manages I/O, which is done by shifting expressions in and out of the ends of the L array. (A more

refined implementation could allow for I/O at selected internal points of the L array in addition to its endpoints.)

The auxiliary memory devices (typically disks with large RAM buffers) are connected to the two ends of the L array, and they extend the memory capacity of the machine in a manner transparent to the user. A variety of schemes for providing virtual memory are possible and have been studied [FrSiS84]. In the simplest scheme, the L array acts as a "sliding window" on a much larger expression, and only the part of the expression within the window is available for processing.

The machine tries to keep the L array nearly full at all times. When the machine is started, the L array is empty, and the machine starts its operation by moving information in from auxiliary memory. When the contents of the L array expand beyond the array's capacity, it simply overflows into auxiliary memory.

Communication among the L cells is through the interconnection network. This network also performs certain processing functions, as will be seen later. All multiprocessors must provide for communication among PEs in the system, but small-grain ones must rely on communication more heavily than large-grain ones because their PEs can do so little on their own. Small-grain multiprocessors favor simple interconnection networks, i.e., ones with a low growth rate, since having larger numbers of PEs, they need larger interconnection networks than large-grain multiprocessors.

This chapter describes an implementation of the FFP machine that uses a binary tree interconnection network network, which is one of the simplest possible. (Note, however, that some preliminary work has been done in the area of using a richer interconnection network to increase the performance of the machine [Kellm82, Plais85].) The internal nodes of this binary tree network will be referred to as T (tree) cells, its leaves will be the L cells, and its root will be connected to the front-end machine. A more refined implementation could connect more than one node, e.g., all T cells on a given level of the tree, to the front-end machine, thus reducing congestion at the root of the tree.

The binary tree interconnection network has many desirable qualities: it is easy to build, its growth rate is low, it can be embedded in the plane, and it is easy to extend. A machine with a binary tree interconnection network can be extended by connecting two binary tree machines to a new root node. Since the binary tree used in the FFP machine does not have to be a full binary tree, the trees need not be of equal size, and consequently the machine can be extended in large or small increments.

The operation of the FFP machine is organized into cycles, and each cycle into three phases:

1. partitioning;
2. execution;
3. storage management.

The partitioning phase, which is global to the machine, creates a separate submachine for each RA. During the execution phase, each of these submachines operates independently, rewriting its RA. The storage management phase is again global to the machine. The following sections explain each of these phases.

III.4 Partitioning phase

The partitioning phase of the machine cycle does the following:

a. it locates all the RAs within the FFP expression contained by the machine, i.e., it determines which computations can be done next;

b. it partitions, or reconfigures, the machine, into a collection of independent *subma-*

chines, including one for each RA. The submachine for an RA will consist of the contiguous set of L cells that hold the symbols of the RA and a tree of communication paths and message processors embedded in the interconnection network.

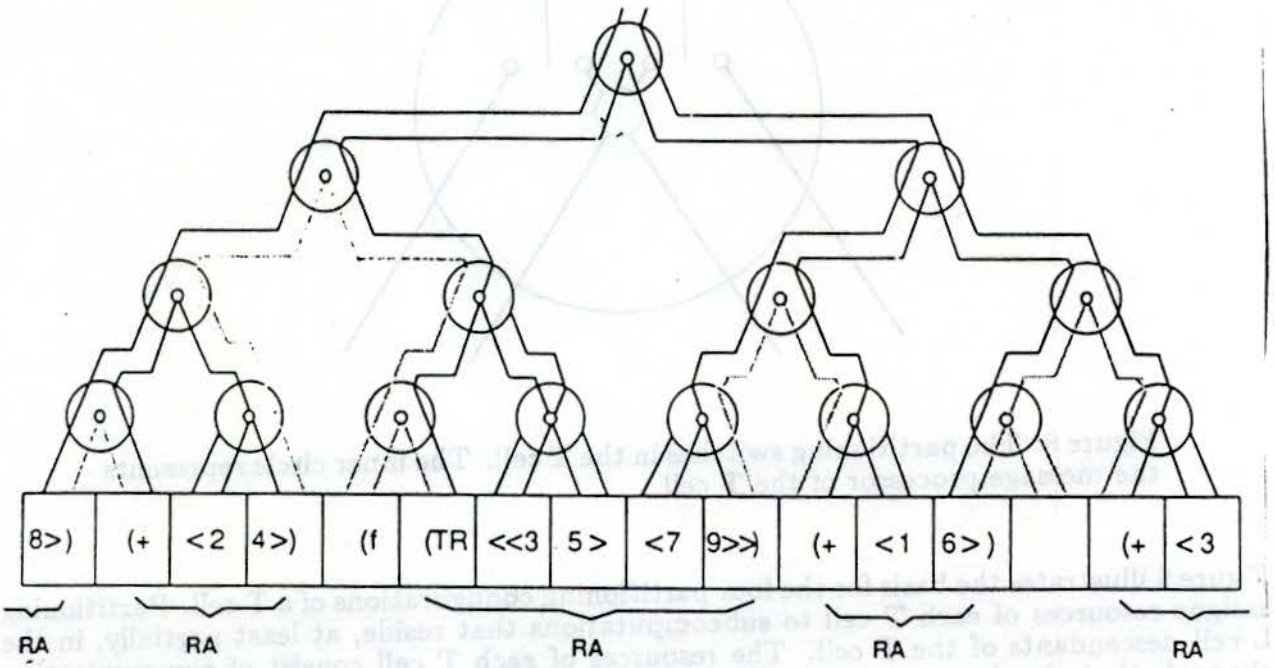


Figure 7: A fragment of an FFP machine showing the results of partitioning. Dark lines represent communication paths of submachines with RAs; gray lines represent communication paths for submachines that do not have RAs. Subtrees without any parentheses in the leaves result in some paths being redundant; these appear as lines that end at a T cell.

Because partitioning is governed by the expression contained in the L array, it may be viewed as a way of fitting or adapting the hardware of the machine to the program that is in memory. The details of how this is done depend on the representation of the FFP expression. This section describes a method due to Middleton that works with the representations discussed in Section III.2.

Figure 7 shows a part of a partitioned machine. In it, five RAs can be seen (two of them only partially), and the submachines constructed for them are shown in dark lines. Gray lines indicate submachines that were created according to the (strictly local) rules of partitioning, but which do not contain RAs.

Partitioning constructs a submachine for each RA, and the submachines of distinct RAs are disjoint. This has several important consequences. Congestion is kept low in the interconnection network, because contention can occur only among the messages of a single RA. Communication packet formats are simple because there is no need to distinguish packets of one RA from those of another; submachines need not deal with alien packets. The routing of packets is simple because packets cannot get out of the submachine; e.g., broadcasting is used instead of specifying path information or using destination addresses.

Partitioning allocates the resources of each T cell by setting its two *partitioning switches* (Figure 8) to one of four possible configurations, all of which occur in the example illustrated in Figure 7.

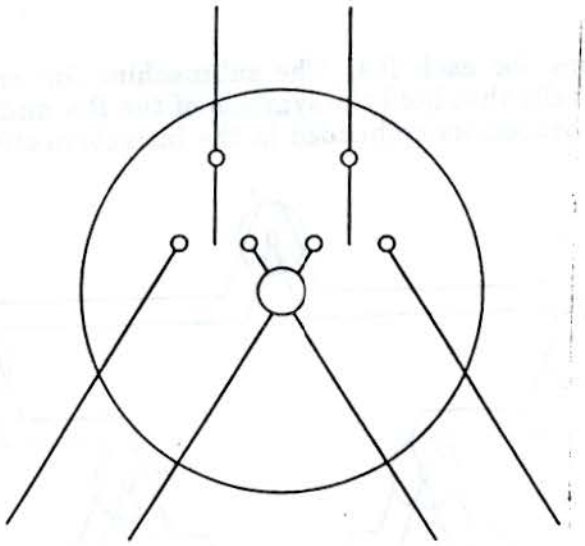


Figure 8: The partitioning switches in the T cell. The inner circle represents the message processor of the T cell.

Figure 9 illustrates the basis for the four partitioning configurations of a T cell. Partitioning assigns resources of each T cell to subcomputations that reside, at least partially, in the L cell descendants of the T cell. The resources of each T cell consist of communication channels that simply route messages through the T cell and a single message processor; the message processor is represented by the inner circle of a T cell in Figure 9.

In case (a) of Figure 9, the T cell shown is part of an edge of the submachine of RA1, which is contained partly in subtree S1 but extends into subtree S0. This T cell simply provides a communication channel for the submachine of RA1; no processing is performed on the messages of that machine. The same T cell provides a root for the submachine of RA2, which is in S1 and S2. The root cell of an RA always processes the messages of that RA, and the message processor of this cell participates in the evaluation of RA2. Additionally, the same T cell is part of an edge of the component machine for RA3, which is partly in S2 but extends into (and possibly beyond) S3; the cell provides only a communication channel for the submachine of RA3.

In case (b), the T cell shown provides a only a communication channel for RA1, which is contained partly in subtree S1 but extends into subtree S0. Because RA2 is in S1, S2 and extends into S3, both children of the T cell see part of RA2, and consequently the message processor will participate in the evaluation of RA2. Case (c) is the mirror image of case (b).

Finally, in case (d), the T cell provides an internal node for the submachine of RA1, which extends beyond both S1 and S2; the message processor of the node will participate in the evaluation of RA1.

If there are any RAs between those explicitly shown in cases (a) (b) and (c), their submachines are fully contained in S1 or S2, and do not involve the T cell in question.

We noted before that while the L cells provide most of the processing power of the FFP machine, the T cells contribute to some degree. The processing power of a T cell lies solely in its message processor. Although each physical T cell may provide communication paths for up to three submachines, messages from at most one RA use the message processor; thus there is no contention between RAs for the processing capacity of a T cell. Since

different RAs use physically distinct communication channels, there is also no contention between RAs for communication channels.

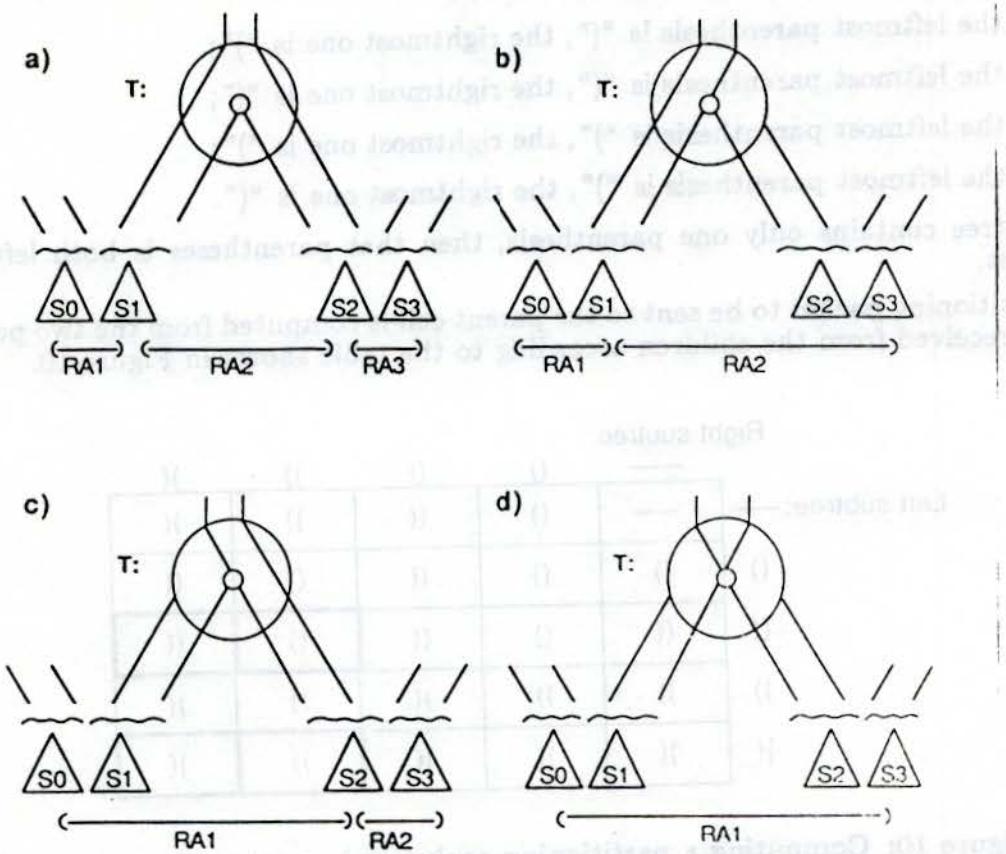


Figure 9: Examples of occurrences of the four partitioning configurations of a T cell. The inner circle of each node represents the message processor of the T cell. The message processor is the only part of the T cell that can contribute to the evaluation of an RA; partitioning assigns each message processor to at most one RA.

Partitioning is a single, distributed operation in which all cells of the machine participate. Partitioning must not only physically partition the machine (by setting the partitioning switches) but must also identify the root nodes of submachines assigned to RAs. Partitioning is done as follows: certain information about parentheses (which delimit applications) is passed from all L cells towards the root of the tree interconnection network. Each T cell receives a partitioning packet from each of its children. In response to the information in these two packets, the T cell sets its partitioning switches and sends a partitioning packet to its parent cell. By the time a partitioning packet is produced by the root of the tree, the partitioning of the machine has been completed and the root nodes of submachines of RAs can be identified.

Partitioning switches are set according to whether the subtrees contain parentheses: if the left (right) subtree contains no parentheses, the left (right) partitioning switch of the T cell is set to the inward position.

To identify the T cells that are the roots of RA submachines, it is sufficient to know about the leftmost and the rightmost parentheses in each subtree of the machine. Accordingly, the

partitioning packet emerging from an each node carries the following information (which can be encoded in three bits) about its L cell descendants.

– : there are no parentheses in the subtree;

() : the leftmost parenthesis is "(", the rightmost one is ")";

((: the leftmost parenthesis is "(", the rightmost one is "(";

)) : the leftmost parenthesis is ")", the rightmost one is ")";

)(: the leftmost parenthesis is ")", the rightmost one is "(".

If a subtree contains only one parenthesis, then that parentheses is both leftmost and rightmost.

The partitioning packet to be sent to the parent cell is computed from the two partitioning packets received from the children according to the table shown in Figure 10.

Left subtree: \ Right subtree:	——	()	((	))	)(
——	——	()	((	))	)(
()	()	()	((	()	((
((	((	()	((	()	((
))	))	))	)(	))	)(
)(	)(	))	)(	))	)(

Figure 10: Computing a partitioning packet to be sent to the parent cell. Highlighted entries correspond to conditions in which the T cell is the root node of a submachine that holds an RA.

The identification of the root nodes of submachines assigned to RAs can be done based on the partitioning packet as follows: if the rightmost parenthesis in the left subtree is "(", and the leftmost parenthesis in the right subtree is ")", then the T cell contains the root node of a submachine that holds an RA.

The method just described locates all innermost applications (by identifying the root nodes of the submachines that will process them) in the FFP expression and constructs submachines to process the RAs. Evaluation methods other than innermost can be obtained by slight modifications of it. For example, if two kinds of parentheses are introduced – say red and green – and partitioning of the machine is done based on the innermost green parentheses, then outermost evaluation can be done as follows: start with all parentheses colored red except the outermost ones. Locate all innermost green parentheses, execute the corresponding applications, and make the applications responsible for creating new green and red parentheses.

III.5 Execution phase

Computation in the FFP machine proceeds by reducing innermost applications; these reductions are always performed by submachines. The larger the RA, the larger its submachine tends to be. (This relationship is complicated by two factors: empty L cells within

the RA increase the size of the submachine, whereas using a compressed representation decreases it.) However, since an L cell holds at most one atomic symbol, even the smallest RA involves at least two L cells, one holding the operator and another one holding the operand. As a result, the reduction of an RA within a submachine is always a distributed computation.

Reduction of an RA takes place during the second phase of the machine cycle. The following subphases can be distinguished:

2.1: request L cell program;

2.2: create the auxiliary representation;

2.3: receive L cell program;

2.4: execute local code in L cell;

2.5: participate in communication (message waves);

2.6: request extra space and suspend execution.

Subphases 2.1 through 2.3 occur only once for each RA and are completed within a single machine cycle; they prepare the RA for execution, and are collectively called *initialization*. Initialization brings a program that will reduce the RA into the L cells of the submachine and constructs the auxiliary representation, which is needed for execution of this program.

The program brought into the L cells of a submachine is determined by the FFP operator of the RA; a program is available for each machine primitive. The program specifies an algorithm to be executed by each L cell that holds a symbol of the RA; execution of the algorithm by all the L cells of the submachine results in the reduction of the RA. These programs are written in a language called the *L cell programming language*, which is a lower level language than FFP. Programs written in the L cell programming language are executed during phases 2.4 through 2.6.

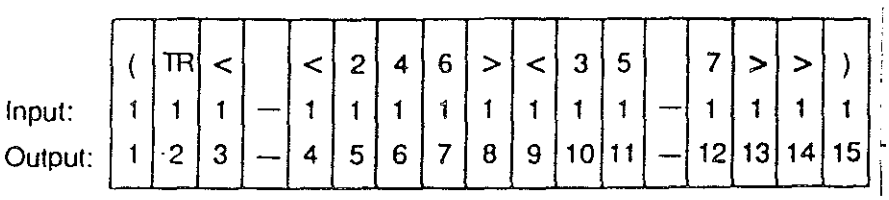
A resident program in the L cells manages initialization; this program can be viewed as the kernel of a simple operating system. Even if an RA takes more than one machine cycle to execute, it goes through initialization only once, during the first cycle. (The information provided by initialization is retained until the reduction of the RA is complete.) If a reduction requires more than one cycle, it resumes execution in one of the subphases 2.4 through 2.6, at whatever step was not completed during the previous cycle.

III.5.1 Constructing the auxiliary representation

As explained in Section III.2, the auxiliary representation includes a certain number of selectors (typically three or four), the relative level number and the index. Computing each of these involves computing cumulative sums of values as follows: The input to the computation is a sequence of values laid out left to right in the L array: $a_1, a_2, \ldots a_n$ (at most one value per L cell, even in compressed representations). The result of the computation is another sequence $b_1, b_2, \ldots b_n$, such that the L cell that held a_i holds b_i. The result is computed as follows: $b_1 = a_1$, and $b_i = a_1 + \ldots + a_i = b_{i-1} + a_i$ for $2 \leq i \leq n$.

We shall explain each of these computations with the help of an example. For simplicity, we shall use the non-compressed representation in these examples. Figure 11 shows the computation of the index for an RA expression. For an L cell that is not empty, the input is $a_i = 1$. Empty L cells do not participate in this computation. (We shall not discuss how this is done.) The result of the cumulative sum computation is the required index value.

Figure 12 shows the computation of the relative level number. In this computation, the input is $a_i = 1$ if the L cell holds an opening bracket or parenthesis, $a_i = -1$ if the L cell holds a closing bracket or parenthesis, and $a_i = 0$ for all other occupied L cells. The cumulative sum gives the correct value for all cells except those that hold opening brackets



	(	TR	<		<	2	4	6	>	<	3	5		7	>	>	)
Input:	1	1	1	—	1	1	1	1	1	1	1	1	—	1	1	1	1
Output:	1	2	3	—	4	5	6	7	8	9	10	11	—	12	13	14	15

Figure 11: Computation of the index

and parentheses, which receive an integer one larger than they should. These cells simply subtract 1 from the value they receive to produce the correct relative level number values.

	(	TR	<		<	2	4	6	>	<	3	5		7	>	>	)
Input:	1	0	1	—	1	0	0	0	-1	1	0	0	—	0	-1	-1	-1
Output:	1	1	2	—	3	3	3	3	2	3	3	3	—	3	2	1	0
RLN:	0	1	1	—	2	3	3	3	2	2	3	3	—	3	2	1	0

Figure 12: Computation of the relative level number (RLN)

Figure 13 shows the computation of three selectors for each L cell. The input to this computation is derived from a modified relative level number (RLN'), which is defined as follows: $RLN' = 0$ for L cells holding closing brackets or parentheses, and $RLN' = RLN$ otherwise. The input to the computation in each L cell is a triple of Boolean values $< RLN' = 1, RLN' = 2, RLN' = 3 >$, with 1 and 0 representing "true" and "false" respectively. The result of the computation in each L cell is a triple of selectors $< S1', S2', S3' >$. These selectors are again computed as modified cumulative sums. $S1'$ is the cumulative sum of the first components of the input triples, while $S2'$ and $S3'$ are the cumulative sums of the second and third components respectively, except that a one in the first component restarts the summing of the second components, and a one in the second component restarts the summing of the third components. The selector value Si for a symbol x is computed from Si' as follows: If the RLN of x is less than i, then Si is set to 0; otherwise, Si is set equal to Si'.

For example, in Figure 13 the L cell holding the FFP symbol 7 has received the selectors < 2, 2, 3 >, and the components of the input triples that contributed to these values have been highlighted. (A zero value in the selector triple should be interpreted as being undefined.)

The values of cumulative sums are computed in the T network. This task, like many others, is done by processing messages that originate in the L cells and are sent up through the T network in an *upsweep* and then broadcast back down from the root of the RA submachine in a *downsweep*. Using this mechanism, the values of the cumulative sums for any expression are computed in time logarithmic in the number of L cells that hold the expression.

Figure 14 shows what an arbitrary node of the submachine does when it participates in the computation of a cumulative sum. On the upsweep, the node receives two integers, a and b. It computes $a + b$ and sends it to its parent node, and also saves the value a. Later, during the downsweep, the node receives a value d from the parent, which it sends to the

Principal repr.	(	TR	<		<	2	4	6	>	<	3	5		7	>	>	)
RLN'	0	1	1	—	2	3	3	3	0	2	3	3	—	3	0	0	0
RLN' = 1	0	1	1	—	0	0	0	0	0	0	0	0	—	0	0	0	0
RLN' = 2	0	0	0	—	1	0	0	0	0	1	0	0	—	0	0	0	0
RLN' = 3	0	0	0	—	0	1	1	1	0	0	1	1		1	0	0	0
S1	0	1	2	—	2	2	2	2	2	2	2	2	—	2	2	2	0
S2	0	0	0	—	1	1	1	1	1	2	2	2	—	2	2	0	0
S3	0	0	0	—	0	1	2	3	0	0	1	2	—	3	0	0	0

Figure 13: Computation of the selectors

left child, while sending $a + d$ to the right child. The root node uses $d = 0$ to start the downsweep.

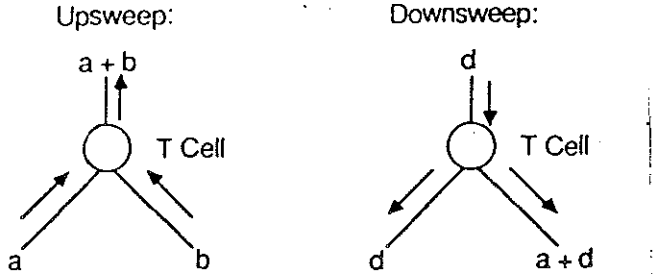


Figure 14: Computing a cumulative sum

The computation of the selectors is somewhat more complicated because of the need to restart the summation repeatedly, but this is easily handled with the modest processing power of a T cell. Danforth's dissertation explains this computation in more detail [Danfo83a].

III.5.2 The L cell programming language

L cells execute programs written in the L cell programming language, a language much simpler than the FFP language. Executing such programs in the L cells reduces the RA. Since the RA is distributed over a number of L cells, reducing an RA is always a distributed computation, and the L cell programs interact, when necessary, by passing messages to each other through the interconnection network.

L cell programs are not kept in the L cells so that the L cell can be kept small while allowing implementation of a rich set of primitive operations. The L cell program for an RA is brought into the machine after the RA is located and the three leftmost FFP symbols within the RA have been found. The first such symbol is a left parenthesis. If the second symbol is an atom, then that atom is the operator and the appropriate L cell

program is brought in. (If no L cell program exists, then a program error has occurred.) If the second symbol is an opening bracket, then the language definition specifies that the RA is to be rewritten using the metacomposition rule (this rule was explained in Section II). In fact, for common functional forms, such as those in Table 2, an L cell program exists which avoids use of the metacomposition rewrite rule and proceeds directly to a subsequent expression. (An example will be given later in this Section.) If the second symbol is an opening bracket but no appropriate program exists for the third symbol, then the metacomposition rule is invoked. The second and third symbols of the RA are used as the request for the L cell program, and they are also broadcast from the root of the submachine so that all the L cells of the RA will know what reduction is to take place.

In the simplest case, all L cells holding an RA may receive the same L cell program. This program is typically a sequence of conditional statements that ascertains which part of the expression the L cell holds, and then performs the required actions. A more economical solution [Mago79,Danfo83a,Danfo83b] is to break each L cell program into segments, and for each L cell to accept only the segment it needs. The L cell chooses the appropriate segment based on the principal and auxiliary representation contained in the L cell.

The L cell programs rewrite the RA by performing the following types of actions: (1) ascertain what part of the RA the L cell holds (this is done by inspecting the part of the principal and auxiliary representation held within the L cell); (2) perform local actions in the L cell (e.g., erase the FFP symbol, or overwrite it with another symbol contained in the L cell program); (3) communicate with other L cells via messages sent through the interconnection network; (4) request extra space and suspend execution.

We will describe three example L cell programs. The L cell programming language presents rich opportunities for detecting improper operands, but we will not address this problem here; the illustrative L cell programs we give will only handle operands of the proper form.

Figure 15 shows the L cell program for APNDL (append-to-the-left), and the effect of applying it to a specific argument. Implementation of this function requires no additional L cells and no communication between L cells and will always be done in a single machine cycle.

RA expression:

	(		apndl			<	<	a	b	>		<	d	f	>	>	)
S1	0		1			2	2	2	2	2		2	2	2	2	2	0
S2	0		1			0	1	1	1	1		2	2	2	2	0	0
S3	0		0			0	0	1	2	0		0	1	2	0	0	0

Result expression:

					<	<	a	b	>			d	f		>	

Figure 15: L cell program for the machine primitive append-to-the-left (APNDL)

1. If ($S_1 < 2$) or ($S_2 = 2$ and $S_3 = 0$) then erase FFP symbol.

Figure 16 shows the L cell program for the FFP operator IP (innerproduct) and the effect of applying it to a given expression. This example illustrates message sending and the use of message waves. The operand is a pair of vectors of atoms, which first must be multiplied componentwise. The corresponding components of the two vectors are brought

together by broadcasting each element of the first vector as a pair of the form < i, i^{th} vector element>, and causing the L cell holding the i^{th} element of the second vector to accept the i^{th} element of the first vector. Figure 16 shows how the intermediate result of these products has replaced the second vector. Finally, these products are summed, and one cell is programmed to accept and hold this sum when the reduction is completed. The summation is done by the tree network: each L cell that computed a product sends a message of the form < +,product> into the tree network. When a node of the submachine receives messages of the form $< +, x >$ from its children, it adds the second components. Nodes other than the root of the submachine then send a message of the form $< +, \text{sum} >$ to their parent; the root node broadcasts the sum.

RA expression:

	(	IP	<	<	1	2	3	4	>	<	11	12	13	14	>	>	)
S1	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	0
S2	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	0	0
S3	0	0	0	0	1	2	3	4	0	0	1	2	3	4	0	0	0

Intermediate result:

(	IP	<	<	1	2	3	4	>	<	11	24	39	56	>	>	)

Final result:

	130															

Figure 16: L cell program for the machine primitive innerproduct (IP)

1. if $S_1 = 2$ and $S_2 = 1$ and $S_3 \neq 0$ then send the pair < S_3, FFP symbol>.
2. if $S_1 = 2$ and $S_2 = 2$ then
   a. accept any message (of message wave 1) of the form $< x, y >$ if $x = S_3$;
   b. store y in BV;
   c. multiply BV by FFP symbol and store in P;
   d. broadcast the pair $< +, P >$.
3. if $S_1 = 1$ then accept any message (of message wave 2). Make the symbol received in the message the new FFP symbol.

Note that messages can be sent sequentially. In IP, for example, broadcasting the elements of one vector to the other must be completed before the summation is begun. Messages are kept apart by putting them into separate "message waves," which are separated from each other by end-of-wave markers.

Each L cell sends an end-of-wave marker to end its participation in each message wave (whether or not it sent a message). Nodes of the submachine wait for one such marker on each of their inputs before sending one to their parent node. As a result, messages of different waves cannot meet each other. The root node of the submachine broadcasts the end-of-wave marker, signalling to the L cells that they can start sending messages in the

next wave.

When many messages participate in a message wave and the messages are not combined, or when an L cell program sends a sequence of message waves, more than one machine cycle may be required for all messages to be processed. It must, therefore, be possible to interrupt a message wave and resume it in the next machine cycle; this requires that each L cell know whether the messages it has sent have been processed. Since messages that are not combined (such as those of the form <S3,FFP symbol>, sent in the first wave of IP) are broadcast from the root, if an L cell receives its own message, that message has been successfully broadcast, and it need not be resent. Messages that are combined (such as the second wave of messages of the form $< +, P >$ in IP) need not be resent if the L cell receives the (combined) result of the message wave.

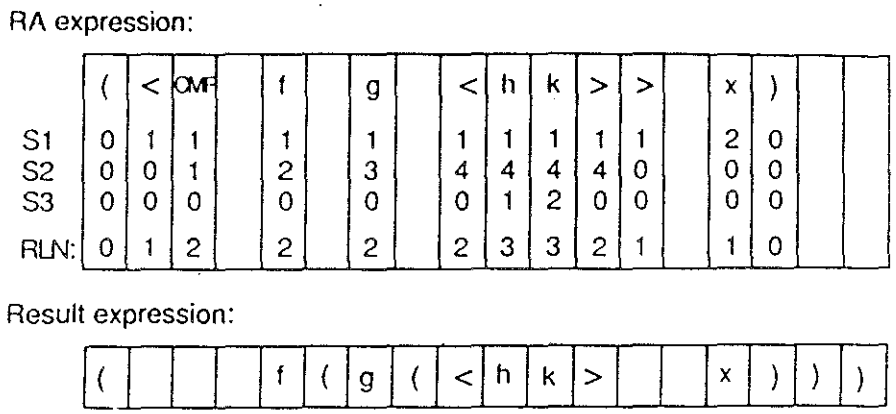


Figure 17: L cell program for the functional form composition (CMP). The FFP expression of the example shown is (<CMP,f,g,<h,k>>:x), which is to be rewritten to (f:(g:(<h,k>:x))).

(1) If $S_1 = 1$ then broadcast $< max, S_2 >$. (Determine the length of the operator expression; broadcast values are compared on the upsweep and only the largest sent upward; the maximum is broadcast by the root.)

(2) If FFP symbol = ")" then accept the broadcast value and store in BV.

(3) If $S_1 = 1$ and $S_2 > 2$ and $RLN = 2$ and FFP symbol $\neq$ ">" then request one empty L cell. If FFP symbol = ")" then request $BV - 2$ empty L cells. Suspend execution until storage management has provided the necessary cells.

(4) If $S_1 = 1$ and $S_2 > 2$ and $RLN = 2$ and FFP symbol $\neq$ ">" then insert "(" to the left of the FFP symbol. (This will require moving the FFP symbol to the right if the empty L cell is provided to the right of the L cell holding the symbol.)

(5) If FFP symbol = ")" then place $BV - 2$ copies of ")" in the L cells provided (either to the right or left).

(6) If $S_1 = 1$ and $S_2 < 2$ then erase the FFP symbol. (The opening "(" is not erased.)

Figure 17 shows the L cell program for the composition operator, and the effect of applying

it to a given expression. This reduction has the property that its result cannot be held in the cells holding the original RA. (When using certain compressed representations, the result of this RA would not require additional space.) The FFP machine uses the mechanism of storage management (to be discussed in the following section) to provide empty L cells wherever they are needed: L cell programs request the required number of empty L cells, suspend execution and wait for storage management, and resume execution in the next machine cycle after the empty L cells have been made available. (An RA is prevented from executing in the next cycle only if storage management moved some part of it to auxiliary storage.)

For simplicity, Figure 17 shows empty L cells within the original RA wherever the result needs them.

III.6 Storage management phase

Storage management is the process of remapping an FFP expression onto the L array of the FFP machine, i.e., changing the principal representation of the expression. The need for storage management arises when the rewriting (or evaluation) of at least one RA requires L cells other than those that held the symbols of the original RA. This occurs most commonly when reduction of an RA results in a larger expression. Since such reduction steps are the basic computational steps of the FFP machine, the required empty L cells should be provided as soon as possible. Furthermore, the principal representation has the property that empty L cells may be found anywhere among the occupied ones, and thus the required empty L cells may be arbitrarily far from where they are needed (or may not be available in the L array at all).

This suggests the need for a frequently repeated, global remapping of the FFP expression so that reduction of RAs can begin soon after they are created. This need is met by organizing the machine operation into cycles and making storage management a part of each cycle.

Storage management can be viewed as moving empty L cells to where they are needed by shifting the contents of L cells in the machine while maintaining the left-to-right order of the FFP symbols. Here we assume that direct horizontal connections exist between the L cells, and they are used only for symbol movement during storage management. Alternatively, symbol movement could be done through the interconnection network.

Figure 18 shows schematically two snapshots of an L array (a blank square indicates an empty L cell, and a square with an x in it indicates an occupied L cell). Before storage management, 1,3 and 7 additional L cells are needed at the places shown by the vertical arrows. The figure shows one of the many possible ways that FFP symbols (or rather contents of occupied cells, which may include partially executed L cell programs) could be shifted to create the required spaces.

Figure 19 shows how the planning of storage management can be formulated as a problem of calculating flows in a linear graph. The situation shown in Figure 18 is depicted as a linear graph, in which the empty L cells needed are sources of flows (vertical arrows entering), empty L cells available are sinks of flows (vertical arrows leaving), and moving symbols correspond to flows (horizontal arrows) which originate in sources and end in sinks. The flows calculated must empty all sources, although they need not fill all the sinks. (Auxiliary memory, at each end of the L array, provides one or more arbitrarily large sinks that make it possible to empty all sources.) A request for n empty L cells is represented by the integer n in the L cell making the request. During storage management, n placeholders "flow out" of the L cell originating the request into adjacent L cells.

The shifting of the FFP symbols is planned for the whole machine using one upsweep and one downsweep in the T network in time proportional to the height of the machine.

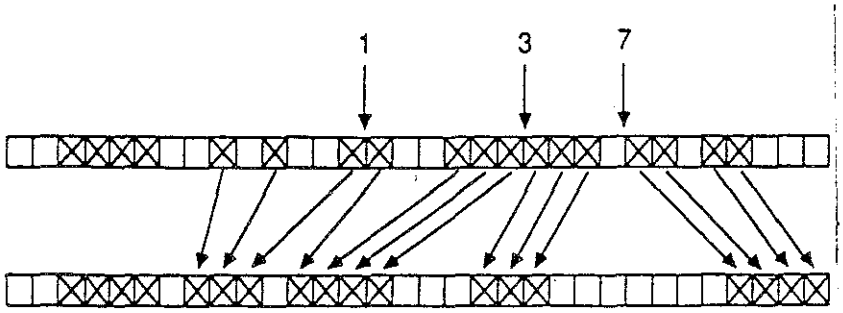


Figure 18: Storage management: an example of how requests can be satisfied by remapping the FFP expression onto the L array. A labelled vertical arrow indicates a request, a blank square indicates an empty L cell, and a square with an X in it indicates an occupied L cell.

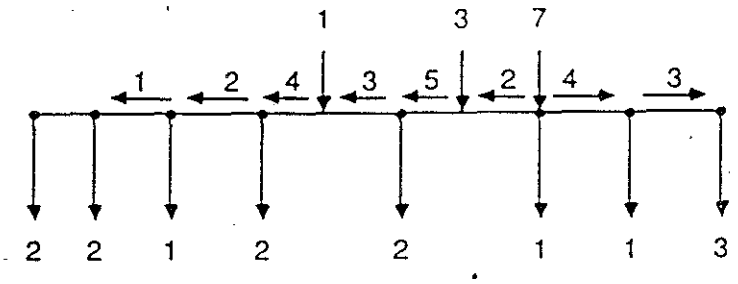


Figure 19: The storage management problem of Figure 18 is represented as a flow problem in a linear graph. Sources represent requests for L cells; sinks denote contiguous sequences of empty cells. Satisfying all requests requires that the contents of each source be emptied into one or more sinks.

Planning consists of determining which direction each FFP symbol should move in the L array, and how far.

After planning is complete, all symbols start moving at the same time, and (assuming that it takes unit time to move a symbol from one L cell to the next) the time needed to do storage management is proportional to the longest distance travelled by any symbol (FFP symbol or placeholder). Optimal storage management planning would therefore minimize the maximum distance travelled by any symbol in the machine.

A sequential planning algorithm for that minimizes the maximum distance travelled by any symbol has been found [StaMa79], but that algorithm is not well-suited to execution on the FFP machine. A planning algorithm suitable for execution on the FFP machine can be based on divide-and-conquer; each T cell computes how many symbols or place-holders will flow across the edge in the L array that connects its left subtree and right subtrees. An optimal algorithm of this type that requires $O(\log n)$ time, where n is the number of L cells, has been described [StaMa81].

The current machine uses a simpler algorithm, which results in symbols travelling no more

than twice as far as necessary. During the upsweep each L cell sends to its parent cell +1 if it is empty, and a negative integer equal to the number of L cells requested if it is not. Each T cell receives two such numbers. It stores both of them, and sends their sum to the parent cell; this sum is equal to the net number of empty L cells in the subtree. At the end of the upsweep, all T cells together have complete information about the distribution of the available and needed empty L cells in the L array. During the downsweep, this information is used by each T cell to compute the flow over the edge that connects the rightmost L cell of its left subtree with the leftmost L cell of its right subtree; this flow is equal to the number of FFP symbols that must be moved from one of its subtrees to the other. Figure 20 shows how this flow, denoted by M, is computed.

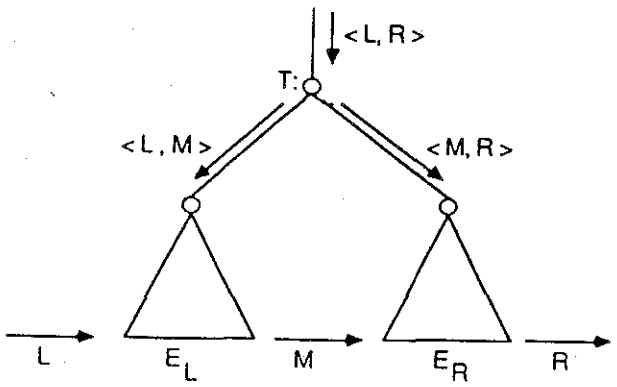


Figure 20: The downsweep of a simple algorithm for storage management preparation. E_L and E_R are the number of empty L cells in the left and right subtrees respectively; L is the flow into the left subtree, R is the flow out of the right subtree, and M is the flow from the left subtree into the right subtree. Each flow can be positive, zero, or negative. M is computed as follows:

M = if $L - E_L > 0$ then $L - E_L$;
if $R + E_R < 0$ then $R + E_R$;
else 0.

The (arbitrary) T cell receives two values, L and R, from its parent, where L is the number of symbols that must enter the tree from the left, and R is the number of symbols that must leave the tree to the right. The tree under the T cell contains E empty L cells (if $E < 0$ then $-E$ empty L cells are needed by the tree). After storage management, the tree will have $E' = E - L + R$ empty L cells. One of the following conditions must hold for E, L and R:

1. $E < 0$ and $E' = 0$ and $L - R = E$, i.e., the number of empty cells requested exceeds the number of available empty cells, and there is a net outward flow of symbols from this subtree to one or both adjacent subtrees. The flow between trees is made as small as possible; hence $E' = 0$. Note that all requests are satisfied in each subtree, even if this means that the subtree overflows into an adjacent subtree or into auxiliary storage.
2. $E = 0$ and $E' = 0$ and $L - R = 0$, i.e., the number of empty cells requested equals the number of available empty cells, and all requests of the subtree can be satisfied within it. The net flow of symbols between this subtree and adjacent subtrees is 0.
3. $E > 0$ and $E' = 0$ and $L - R = E$, i.e., the subtree contains more empty cells than requests, but the net flows into the subtree from adjacent trees cause all empty cells to be filled.
4. $E > 0$ and $E' > 0$ and $L \geq 0$ and $R \leq 0$, i.e., the subtree contains empty cells both before

and after storage management. Consequently, the tree is able to accommodate its own requests as well as any flows in from adjacent trees.

This relationship among L, R and E can be stated more concisely using a predicate p defined as follows:

$$p(L,R,E) = \{L - R = E\} \text{ or } \{L - R < E \text{ and } L \geq 0 \text{ and } R \leq 0\}.$$

It can be shown that computing M as given in Figure 20 preserves this property, i.e., if $p(L,R,E)$ then $\{ p(L,M,E_L)$ and $p(M,R,E_R) \}$.

At the root cell of the machine, the values of L and R are chosen to establish p initially at the beginning of the downsweep. Note that not only can these flows be set to allow the expression in the L array to overflow into the virtual memory (thus satisfying all requests); they can also be set to fill the machine completely or partially from auxiliary memory.

We note two advantages of machine-wide storage management that uses global information: (1) deadlocks are prevented, since there is no competition for any specific empty L cell (if more space is needed than is available, the machine simply overflows into auxiliary memory); (2) the movement of FFP symbols is conflict-free, and thus can be done with the maximum amount of parallelism (i.e., all symbols can be moved simultaneously).

III.7 The L and T cells

So far the FFP machine has been considered from a global point of view, i.e., as a collection of small processing elements that together act as an interpreter for the FFP language. This section will take a more local view, and briefly consider the two building blocks of the machine, the L and T cells.

Both kinds of cells are small-grain processing elements, too simple and too small to know the FFP language; a single L cell is not able to recognize, hold or reduce an RA, only a collection of cells can. The behavior of individual cells is most easily related to the phases of the machine cycle, and that is what we shall do here.

The L cell is an instruction-set processor. It executes a small resident kernel, which knows about the phases of the machine cycle, and the L cell program (brought into the machine during execution), which defines a reduction rule. The T cell is not programmable in the usual sense. It only performs some simple processing on packets as specified by information contained in them.

The T cell consists of two parts: G (global) and P (partitioned). G deals with packets that affect the whole machine irrespective of its present partitioning, e.g., the packets that do partitioning, request L cell programs, or prepare for storage management. P is the part of the T cell that is partitioned in each machine cycle, and it deals, for example, with submachine initialization, including generation of the auxiliary representation, and the message packets generated by the L cell programs. The cells of the L array are connected to their parent cells in the T network through both G and P channels, and thus one may consider the whole tree machine to be composed of two trees: the G tree, which is never partitioned, and the P tree, which is partitioned once each machine cycle into submachines as discussed in Section III.4.

The L cell kernels together drive the machine cycle by emitting a variety of message packets in well-defined sequences. (The T cells treat packets received on their P channels uniformly, whether they were sent by the kernel or by L cell programs.) The packets are of variable length, and are preceded by a type field indicating the nature of the packet (e.g., a partitioning packet, or a message from an L cell program). The packets are pipelined on narrow (typically bit-serial) channels.

The L cell begins the machine cycle by sending a partitioning packet on its G channel. If the L cell contains an L cell program (that is, if the L cell contains part of an RA that was

interrupted before completing its reduction) then it immediately continues its execution, which will typically involve sending and receiving messages (on its P channel) generated by the L cell program. Otherwise, the L cell begins initialization by sending a series of packets to determine which L cell program to request, and to compute the auxiliary representation. The L cell will at some later time receive on its P channel a packet stating whether the submachine it is in contains an RA, and if so, another packet stating which L cell program to accept. After these, packets bringing components of the auxiliary representation are received. If the L cell receives a program, it begins execution immediately, otherwise it remains idle for the rest of the machine cycle.

The duration of the machine cycle is determined by an interrupt packet, which is broadcast from the root of the machine on the G channel. When an L cell receives this interrupt packet, it continues any RA processing in progress but begins preparation for storage management by sending its request for L cells in a packet on the G channel. These packets together form the upsweep of planning for storage management. When the downsweep of storage management planning reaches (on the G channel) a T cell that contains the root node of a submachine, the root node finishes broadcasting (on its P channel) any message that is currently passing through it, but does not allow any succeeding messages to pass through it. All other messages that have entered into the tree network (even if only partially) are discarded. When the storage management planning downsweep packets have been received by an L cell, and all messages have been emptied out of the T network above it, the L cell begins storage management.

IV. EXAMPLE

In this section we discuss execution of a matrix multiplication program on the FFP machine. Our program trace will treat the case of a pair of $n \times n$ matrix operands, although the program will multiply any pair of conforming matrices. Multiplication of matrices of atoms can be done with an FFP machine primitive, but space constraints prohibit our discussing the implementation of that primitive.

A variety of matrix multiplication programs can be written in FFP, with widely varying performances when executed on the FFP machine. Backus gave an easily understood highly parallel program [Backu78], but the parallelism requires $O(n^3)$ space on the FFP machine, and hence $O(n^3)$ time. The program that follows computes the entries of each row of the result matrix in parallel, but finds the rows sequentially. This program exhibits less parallelism than Backus's, but requires only $O(n^2)$ space and time.

The comments about space requirements given below apply to a noncompressed representation; a compressed representation would require less space and less storage management (at the cost of more complex L cell programs). Except for the step that rewrites the functional form INSERT, each step of the execution of the example program requires at most a constant number of additional L cells and will therefore require at most two machine cycles to complete.

The definition of the program MM (matrix multiply) is the following:

MM ≡<CMP,2,<INSERT,ROWOP>,APNDR,<AR,<CMP,<BU,ROTL,ϕ >,TR>>>

Application of the function MM to a pair of square matrices $< A, B >$ produces an expression consisting of a single RA. Because of its operand, the expression requires $O(n^2)$ space. The first step in evaluation causes MM to be replaced by its definition.

(MM:<A,B>) → (<CMP,2,<INSERT,ROWOP>,APNDR,<AR,<CMP,<BU,ROTL,ϕ >,TR>>>:<A,B>)

The resulting expression has a single application (which is therefore an RA) with CMP (denoting composition) as the controlling operator of a composite function. Rewriting this produces:

(2:(<INSERT,ROWOP>:(APNDR:(<AR,<CMP,<BU,ROTL,ϕ >,TR>>:<A,B>))))

The resulting expression has several applications but only one RA, whose operator is a composite function with AR (Apply-to-the-right) as its controlling operator. AR applies its parameter to the rightmost entry B of its operand, resulting in:

(2:(<INSERT,ROWOP>:(APNDR:<A,(<CMP,<BU,ROTL,ϕ >,TR>:B)>)))

The sole RA in the new expression has CMP as its controlling operator; the next rewrite produces:

(2:(<INSERT,ROWOP>:(APNDR:<A,(<BU,ROTL,ϕ >:(TR:B)) >)))

Next the transpose function TR transposes B. Transposing a matrix of atoms can be done in place (i.e., without additional L cells) by a machine primitive in $O(n^2)$ time. Since there is no danger of confusion, we will represent the transposed matrix by B as well, giving:

(2:(<INSERT,ROWOP>:(APNDR:<A,(<BU,ROTL,ϕ >:B) >)))

MM operates by finding the result matrix a row at a time. The initial approximation to the result matrix is the empty matrix ϕ, which is created using the BU (Binary-to-unary) functional form. BU pairs ϕ with the matrix B producing:

(2:(<INSERT,ROWOP>:(APNDR:<A, (ROTL:< ϕ,B>) >)))

and ROTL (Rotate-left) interchanges the elements of this pair, giving:

(2:(<INSERT,ROWOP>:(APNDR:<A,<B,ϕ >>)))

APNDR (Append-to-the-right) then produces the following, where we have expanded the matrix A:

$$(2{:}(<\text{INSERT,ROWOP}>{:}< a_{,1}, a_2, \ldots a_n, <\text{B},\phi >>))$$

Rewriting the INSERT functional form requires O(n) additional space to create n new applications, with ROWOP the operator in each:

$$(2{:}(\text{ROWOP}{:}< a_1,(\text{ROWOP}{:}< a_2, \ldots (\text{ROWOP}{:}< a_n,<\text{B},\phi >>) \ldots >)>))$$

ROWOP is then executed n times (sequentially, from right to left), once for each row of the matrix A. ROWOP takes an argument of the form $< a_i, <\text{B},\text{C}' >>$, where a_i is a row of A and C′ is a matrix consisting of the first $n - i$ rows of the result matrix. The value of ROWOP operating on this argument is of the form <B,C″ >, where C″ is C′ augmented with another row of the result matrix. Thus each execution of ROWOP multiplies a row of A with the entire matrix B and produces another row of the result matrix. We will not describe in detail how ROWOP works, but it is a machine primitive that is similar to IP (innerproduct), described in Section III.5.2. ROWOP broadcasts the row a_i to the matrix B, where scalar multiplications are performed in the L cells, and the n inner products are formed in the T cells. Each execution of ROWOP requires $O(n)$ time; the n executions of ROWOP require $O(n^2)$ time. The result of the sequence of executions of ROWOP is a pair of matrices consisting of B and the (now complete) partial result C. In the final step of the computation, the selector function 2 selects the result:

$$(2{:}<\text{B},\text{C}>) \rightarrow \text{C}.$$

V. CONCLUSIONS

Although reduction machines have thus far not been investigated thoroughly, they have considerable promise as easily programmable parallel computers. But their implementation poses many problems, some of which are poorly understood. The FFP machine design differs from both conventional architectures and other reduction machines on several counts, and offers a new approach to problems that have been obstacles to the programmability of parallel computers.

The program decomposition problem is handled automatically and gracefully: the hardware is reconfigured at run-time to fit the needs of the program running on the machine.

The machine is programmable in a high level language such as FFP, with parallelism invoked naturally and without explicit programmer specification. FFP operators specify transformations of data objects, while message passing is encapsulated in programs of the L cell language and does not concern the FFP programmer.

The parallelism of the FFP machine is not limited to that in a user program, but occurs also in the execution of individual language operations and the operating system functions, including partitioning and storage management.

Finally, although the FFP language has guided and inspired many of the design decisions in the machine, the machine is not constrained to that language. It can execute lambda calculus based languages, and we expect that the machine will also do well on logic programming languages [Smith84].

Because the design represents a radical departure from machines that are commonly available, the ultimate utility of such a machine is difficult to assess. Moreover, the massive asynchronous parallelism of the design makes it difficult to predict or simulate performance on problems that are not either small or based on highly regular computations. It appears that a thorough and realistic evaluation will only be feasible with a carefully designed prototype.

Acknowledgements

The authors wish to thank the many graduate students, and especially Edoardo Biagioni and William Partain, who read drafts and offered suggestions.

References

[Backu78] Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21**, 8 (1978), pp. 613–641.

[Berkl75] Berkling, K.J., Reduction languages for reduction machines. *Proceedings of the Second Annual Symposium on Computer Architecture, 1975*, pp. 133–140.

[Danfo83a] Danforth, S., DOT, a distributed operating system model of a tree-structured multiprocessor. Ph.D. dissertation, University of North Carolina at Chapel Hill, 1983.

[Danfo83b] Danforth, S., DOT, a distributed operating system model of a tree-structured multiprocessor. *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 194–201.

[FrSiS84] Frank, G.A., W.E. Siddall, and D.F. Stanat, Virtual Memory Schemes for an FFP Machine. *International Workshop on High-Level Computer Architecture 84* (Los Angeles, California, May 23–25, 1984).

[Kellm82] Kellman, J.N., Parallel Execution of Functional Programs. Technical Report No. CSD 830114, 1982, UCLA Computer Science Dept.

[Mago79] Magó, G.A., A network of microprocessors to execute reduction languages. Two parts. *International Journal of Computer and Information Sciences* 8, 5 (1979), 349-385, 8, 6 (1979), pp. 435–471.

[Mago80] Magó, G.A., A cellular computer architecture for functional programming. Digest of Papers, *IEEE Computer Society COMPCON* (Spring 1980), pp. 179–187.

[Mago85] Magó, G.A., Making parallel computation simple: the FFP machine. Digest of Papers, *IEEE Computer Society COMPCON* (Spring 1985), pp. 424–428.

[MagMi84] Magó, G.A. and D. Middleton, The FFP Machine—A Progress Report. *International Workshop on High-Level Computer Architecture 84* (Los Angeles, California, May 23–25, 1984).

[Middl86] Middleton, D., Alternative program representations in the FFP machine, Ph.D. dissertation, University of North Carolina at Chapel Hill, 1986.

[Plais85] Plaisted, D.A., An Architecture for Fast Data Movement in the FFP Machine. *Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science 201, pp. 147–163. Springer-Verlag 1985.

[Smith84] Smith, B., Logic programming on an FFP machine. *Proceedings of the 1984 International Symposium on Logic Programming* (Febr. 6-9, 1984, Atlantic City, New Jersey), pp. 177–186.

[StaMa79] Stanat, D.F. and G.A. Magó, Minimizing maximum flows in linear graphs. *Networks* 9, 4 (1979), pp. 333–361.

[StaMa81] Stanat, D.F. and G.A. Magó, Optimal storage management in a cellular computer. Technical Report 81-006, Department of Computer Science, University of North Carolina at Chapel Hill, 1981.

[Tolle81a] Tolle, D.M., Coordination of computation in a binary tree of processors: an architectural proposal. Ph.D. dissertation, University of North Carolina at Chapel Hill, 1981.

[Tolle81b] Tolle, D.M., Implanting FFP trees in binary trees: an architectural proposal. Proceedings of the 1981 ACM Conference on Functional Programming and Computer Architecture (Oct. 18–22, 1981, Portsmouth, New Hampshire), pp. 115–122.