Information and Computer Science

im P. Gostelow
obert E. Thomas

Technical Report #127

Department of Information and Computer Scienc
University of California, Irvine
Irvine, CA 92717

March 1979

UNIVERSITY OF CALIFORNIA
IRVINE

Performance of a Dataflow Computer*

by

Kim P. Gostelow
Robert E. Thomas

Technical Report #127

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

March 1979

*This work was supported by NSF Grant MCS76-12460: The UCI Dataflow Architecture Project.

Performance of a Dataflow Computer*

by

Kim P. Gostelow,
University of California, Irvine

and

Robert E. Thomas,
University of California, Irvine

Abstract

Our goal is to devise a computer comprising large numbers of cooperating processors (LSI). In doing so we reject the sequential and memory cell semantics of the von Neumann model, and instead adopt the asynchronous and functional semantics of dataflow. We describe a dataflow base machine language and unfolding interpreter that generates large numbers of asynchronous tasks for execution. Also presented is the high-level dataflow programming language Id, as well as an initial design for a dataflow machine and the results of detailed, deterministic simulation experiments on a part of that machine. For example, we show that a dataflow machine can automatically unfold the nested loops of n-by-n matrix multiply to reduce its time complexity from $O(n^3)$ to $O(n)$ so long as sufficient processors and communication capacity is available. Similarly, quicksort executes with average $O(n)$ time demanding $O(n)$ processors. Also discussed are the use of processor and communication time complexity analysis, and "flow analysis", as aids in understanding the behavior of the machine.

Index terms: dataflow, multiprocessor architecture, large-scale integration, asynchronous execution, parallel computer, distributed computer, concurrency, functionality, locality

Phone: (714) 833-5517 or 5233

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

*This work was supported by NSF Grant MCS76-12460: The UCI Dataflow Architecture Project.

Abstract

Our goal is to devise a computer comprising large numbers of cooperating processors (LSI). In doing so we reject the sequential and memory cell semantics of the von Neumann model, and instead adopt the asynchronous and functional semantics of dataflow. We describe a dataflow base machine language and unfolding interpreter that generates large numbers of asynchronous tasks for execution. Also presented is the high-level dataflow programming language Id, as well as an initial design for a dataflow machine and the results of detailed, deterministic simulation experiments on a part of that machine. For example, we show that a dataflow machine can automatically unfold the nested loops of an n-by-n matrix multiply to reduce its time complexity from O(n³) to O(n), so long as sufficient processors and communication capacity is available. Similarly, quicksort executes with average O(n) time demanding O(n) processors. Also discussed are the use of processor and communication time complexity analysis, the "flow analysis", as aids in understanding the behavior of the machine.

Index terms: dataflow, multiprocessor architecture, large-scale integration, asynchronous execution, parallel computer, distributed computer, concurrency, functionality, locality

1.0 INTRODUCTION

The ability of LSI technology to inexpensively produce large numbers of identical, small, yet complex devices should make possible a general-purpose computer comprising hundreds, perhaps thousands, of asynchronously operating processors. Within such a machine each processor accepts and performs a small task generated by the program, produces partial results, and sends these results on to other processors in the system. Many processors thus cooperate, asynchronously, to complete the overall computation. A natural consequence of such behavior should be decreased time for problem solution as new processor modules are added to the machine.

This paper describes results of simulation experiments done at Irvine on an initial design for such a machine. Our approach has been to reject the von Neumann computer as a model of computation, and instead, to adopt a model based on dataflow. This has necessitated the development of a new dataflow base (machine) language and high-level programming language, as well as the machine design itself.

1.1 Background

Several computers have been devised in attempts to synthesize a single large machine from a collection of smaller processors, e.g., Illiac IV [10], Cm* [17], and C.mmp [33]. So far, however, multiprocessor machines have not achieved the ease of programming and level of performance expected. Concerning ease of programming, for example, the programmer should not be concerned with how a program is

partitioned into concurrently executable pieces or how these pieces coordinate; nor should the programmer be concerned with the number or physical arrangement of the processors comprising the system. Instead, a program should automatically break apart into small pieces that are executed asynchronously, with minimum interference from one another. Several researchers [4,8,15,19] have concluded that this can be achieved only with significant changes to the widely accepted "von Neumann model".

For the machine we have in mind, two particularly troublesome attributes of the von Neumann model are

1. (centralized) sequential control
2. memory cells

Sequential, one-instruction-at-a-time control is inappropriate because it prohibits the asynchronous behavior and distributed control we consider essential. It also burdens the programmer with the need to explicitly specify (or employ an analyzer to determine) exactly where concurrency is to occur. The second attribute, the memory cell, presents a more subtle difficulty which we illustrate by an extreme case: the global variable. Imagine a situation in which several otherwise asynchronous modules are busy executing tasks and that these tasks require coordination through a common cell. This may call for complex synchronization controls to ensure orderly use of the global variable. Such controls are difficult to design into a highly distributed machine, may be costly in execution time, and are tedious for programmers to use. Finally, memory cells present particularly thorny problems for program verification largely because there is no

simple mathematical characterization of [illegible] cell.

The semantics of nearly every programming language so far devised[1] is closely related to the von Neumann model. In contrast, our approach to multiprocessor machine design is to adopt a programming language semantics more appropriate to the hardware technology now available. Thus we view the fundamental problem of designing a general-purpose, asynchronous, multiprocessor not as simply the design of an appropriate bus and machine interconnection scheme, nor the design of a machine which can efficiently manipulate arrays or interchange numbers. The problem is instead one of avoiding complications due to the fundamental principles of the von Neumann model.

1.2 The Dataflow Model

The version of dataflow used here is a new and more asynchronous version of a semantic model which has been in existence for some time [9,15,20,21,27]. The distinctions between "data flow" semantics and von Neumann's principle of "control flow" are two: First, dataflow operations are executed asynchronously. Second, only values (not a memory cell that holds a value) are the results of computations, that is, all computations are functions. A dataflow program is a directed graph where the nodes are operators and the directed edges denote the path of an operand from the output of one node to the input of another

---

[1] Some exceptions are VAL [1], Id [6], LUCID [7], FFP [8], LAU [13], ISWIM [24], and pure LISP [25].

(or the same) node. Note that such a rule induces a minimum of precedence among operators. Operationally:

1. A dataflow operation executes when and only when all required operands become available (asynchrony).
2. A dataflow operation is purely functional and produces no side-effects (functionality).

By adopting these principles, it is possible to realize the asynchronous execution of programs without the need for special parallel programming constructs (e.g., parbegin-parend, fork-join) or parallel program analysis of any kind.

Other dataflow machines have been proposed [14,15,22,28,32], and one, the Davis machine, has been constructed though on a very small scale. Our work differs from each of the above machines in at least one of two important respects. First, the underlying interpretive mechanism is far more asynchronous; second, our high-level dataflow programming language, Id (for Irvine dataflow) [5,6], is a complete language and may be used to write operating systems and distributed database systems as well as the smaller kinds of problems we are able to describe here. Finally, there are important differences in goals. Our goal is not just to devise a machine that can execute programs rapidly. We are also interested in how dataflow can help in solving general system problems. For example, user protection [12] and exception handling facilities are also being integrated into the design.

2.0 DATAFLOW

In this section we briefly describe our base machine language and "unfolding" interpreter with the aid of example Id programs. Our purpose in discussing Id is to show that high-level dataflow languages can be defined, and that dataflow programs are no more difficult to write than programs in von Neumann languages. Furthermore, we assume that programs are never written directly in the base language, for then it would be possible to combine operations in ways which may directly affect other users or introduce undesired non-deterministic computation. These "illegal" constructions are not possible in Id.

2.1 Dataflow Values

As discussed earlier, the concept of an addressable memory is not part of the dataflow model. Instead, expressions simply produce output values for each given set of input values, and these output values in turn are input to succeeding expressions. There are two classes of values in our dataflow system[2]: elementary and structured. Elementary values are integers, reals, booleans, strings, etc., and need no further discussion. Structured values or structures are trees, defined recursively as either the distinguished empty symbol Λ, or a set of <selector:value> ordered pairs where selector (also called a subscript) is an integer or a string, and value is any elementary or structured value. Structures may be used to represent vectors,

---

[2] A third kind of value called a "stream" [6] is also defined, but is not discussed here.

matrices, etc. For example, Figure 2.1a shows two vector values, x and y, each of length 3; a 2×3 matrix in row major order is shown in Figure 2.1b .

2.2 Id Constructs and Corresponding Base Language Schemata

2.2.1 The Function and Predicate Schema -

A translation of the Id expression

```
( y ← 2*x ;
  x ← a+b ;
  z ← x↑b
  return y,z )
```

is shown in Figure 2.2 . Each box in the figure is an operator with ordered input and ordered output lines. A line may be named, in which case its name is written adjacent to it (e.g., line a). The function performed by each operator is written inside its box; the operator label, if any, appears adjacent to the box (e.g., operator s). Note that a constant value in Id is represented by a constant function in the base language. Assignment in Id is not an operator as it is in other languages, rather it is a specification to the compiler to name a line.

A value is carried by a token that flows along a line. A token is represented by a large black spot with the value it carries written adjacent to it. Execution of an operator is illustrated in Figure 2.3a . The first principle of dataflow (Section 1.2) states that an operator executes when and only when the required input operands have arrived, at which time the operator is said to be enabled. An enabled

operator may execute by absorbing those tokens as input, computing a result, and producing one or more output tokens that carry the result values. Note that an operator follows no particular "law of conservation of tokens". Exactly one token is removed from each enabling input and one result token is produced for each active output. Figure 2.3b shows that a token encountering a fork in a line is replicated so a value may follow all outbranches of a line. In this way a single result may be input asynchronously to many different operators.

The reader may carry out the evaluation of the program of Figure 2.2 by placing one token on each of the input lines a and b, and following the dataflow execution rules illustrated by Figures 2.3a and 2.3b . Note that the order of evaluation of enabled operators is unimportant since there are no races, i.e., computation is determinate [3,26]. This latter characteristic is in part guaranteed by a rule of dataflow program formation called the single-assignment rule which makes the connection shown in Figure 2.4 illegal. It is not possible for two tokens to arrive on the same input line intended for the same instance of execution of the same operator. In essence, once a value is defined it can never be changed. This makes the Id expression

```
( x ← a+b ;
  y ← a-b ;              ** illegal **
  x ← a
  return x,y )
```

illegal, since x is defined twice.

To further illustrate the functional nature of dataflow, the only operators defined on structured values are select and append. In the

following example, let the values carried by the tokens on lines x and y be the structures λ and μ, respectively, shown in Figure 2.1a . Also, let lines i, j, and z carry the respective values i, j, and z. Then select(x,i), written x[i] in Id, outputs the value λ_i if it exists, otherwise an error value is produced. Somewhat more complex is append(x,j,z), which creates a new structure value identical to λ except that selector j is made to refer to value z. We emphasize that the creation of this new structure does not modify the value on line x; rather, a new (logical) copy λ' of λ is first made, and then the value z is associated with selector j in λ'. Thus append(append(Λ,1,x),2,y) produces the structured value γ shown in Figure 2.1b .

The above is an introduction to simple dataflow expressions. More complex operators needed for procedure application, conditional expressions, and loops are briefly described below. Before continuing however, we will review how the principles of dataflow are reflected in what has already been described. First, all dataflow operations are asynchronous unless a sequential dependence is explicitly noted (by a path from the output of one operator to the input of another as shown in Figure 2.2). Second, dataflow is functional. There are no side-effects such as updating a global variable. This allows computation to proceed without concern for coordination with other asynchronously operating tasks.[3]

---

[3]Updating a database and other "history-sensitive" functions are handled by dataflow resource managers described in [5,6]. Resource managers are not discussed here.

2.2.2 The Conditional Schema -

The conditional schema (Figure 2.5) corresponds to the Id expression

(if p then f else g)

where p, f, and g stand for arbitrary Id expressions dependent upon some input x (p must produce a boolean result). Whenever a token arrives on line x the predicate p is evaluated. The SWITCH executes after the tokens on both lines x and b have arrived, and if the predicate is true then the token from x is sent by SWITCH to the schema f, otherwise to g. The result from f or g then goes to the ⊗ operator which simply passes what it receives on either input directly to its output. It is important to recognize that each execution of SWITCH causes only one token to reach ⊗, either from the T or F side, making the entire conditional schema an expression (function). Functional behavior is basic to Id schemata, even though some operators (such as ⊗ and SWITCH) do not individually exhibit such behavior.

2.2.3 The Procedure Application Schema -

A value of type procedure is an encoding of a dataflow procedure (subroutine). Since a procedure is a value, it may be carried by a token and input to an application schema. In Id we write f(x) to mean that the (procedure valued) token arriving on line f is to be applied to the argument token appearing on input line x. An application schema comprises the two operators ACTIVATE (denoted by A) and TERMINATE (denoted by A^{-1}) that work together as shown in Figure 2.6 .

The A operator creates[4] an instance of the procedure according to the description β carried by the procedure token, and sends a single token carrying the argument value x (which may actually be a structure value holding several arguments) to the procedure. Every procedure comprises a BEGIN operator that is initiated upon receipt of the input argument. BEGIN then passes its input through to its output line which may then fork to the inputs of the expression(s) comprising the body of the procedure. The result produced from the procedure body (which may also be a structured value holding several results) is sent to an END operator that sends the result on a single token back to the A^{-1} operator at the point of application. The A^{-1} operator simply distributes the result to those operators expecting the value $\beta(x)$. The procedure instance created by A is destroyed after its result has been sent to A^{-1} .

2.2.4 The Loop Schema -

The loop schema shown in Figure 2.7 corresponds to the Id expression

```
( initial x ← a
  while p do
    new x ← f
  return x )
```

where p and f stand for arbitrary Id expressions dependent upon x (p must produce a boolean result). Note that a loop is simply an expression that produces a value. On initiation of a loop, a token is

---

[4]Creation of a procedure instance will become clear when activity names are discussed in Section 2.3 .

placed on line x from line a, and then, on each successive iteration line x receives tokens that cycle through the loop with values f(x), f(f(x)), ..., until p(x) becomes false. A loop expression closely resembles a mathematical recurrence relation(s) but with the inclusion of a stopping condition and a specification of the particular values to be returned. (The D, D^{-1}, L, and L^{-1} operators in Figure 2.7 will be explained below. In any case these four operators always execute the identity function with respect to their input values.)

The above four types of expressions and schemata can be more complex than so far indicated. For example, a loop may actually comprise any number of lines (recurrence variables) that circulate values, asynchronously produce successor values, and return more than one result. A statement utilizing a loop with two recurrence variables is

```
m,n ← ( initial i ← 0 ; sum ← 0
        while i<10 do
          new i ← i+1 ;
          new sum ← sum+i↑(i+1)
        return sum,i )
```

This more complex Id statement produces the base language program shown in Figure 2.8 .

2.3 The Unfolding Interpreter

Section 2.2 discussed several operators as well as the four basic expression schema classes of the base language (function and predicate, conditional, procedure application, and loop). Also presented was a rather straightforward interpretive mechanism for

asynchronously executing a dataflow program by moving tokens along lines and executing operators. The asynchrony allowed by straightforward interpretation of a dataflow program graph (such as between * and ↑ in Figure 2.2) is called static parallelism. The following describes an unfolding interpreter that allows dynamic parallelism, which produces far more asynchrony than that achieved from static parallelism alone.

Consider operator s in Figure 2.8 (the ↑ function) as successive values are fed to its inputs. Since the rate at which those inputs are generated may be greater than the rate at which s executes, we can look at s at a time when, say, two complete sets of input values are present. Such a situation is shown in Figure 2.9a where the (logical or positional) order of arrival is denoted by a superscript. By the first principle of dataflow, asynchrony, the first execution of s can take place since both a^1 and b^1 are present (Figure 2.9b). For the same reason the second execution can take place immediately (Figure 2.9c). However, the second principle of dataflow, functionality, would also allow the second execution to precede the first, since functionality means that any result is dependent only on the function and its arguments. Let each distinct execution of an operator be termed an activity. If sufficient free processors are available, and if each activity is associated with one processor, then both activities shown in Figure 2.9 could be carried out asynchronously (concurrently). For example, the computations in Figure 2.8 that produce the arguments to operator s may be very fast relative to the speed of s itself. This means that many executions of s may go on simultaneously with the results being summed by the rest of the loop

as they become available. The effect is to "unfold" [4,21] the loop into many instances of each operator.

Figures 2.8 and 2.9 illustrate the basic idea behind the unfolding interpreter. However, the presence of loop and procedure application expressions adds yet another dimension to the unfolding that can be achieved. For example, let operator s in Figure 2.8 be the procedure application f(i) rather than the primitive function ↑ . Each execution of s would then give rise to all the activities within procedure f. Furthermore, concurrent executions of operator s imply that concurrent invocations of f may also take place. Identical remarks would hold true were ↑ a loop expression instead; that is, concurrent execution of distinct invocations of the same loop is also possible. This increased asynchrony is achieved by the appropriate interpretation of the repetitive execution (specified either by iteration or by recursion) of a dataflow operator or schema. When the number of repetitions is determined at execution time, any resulting concurrency is considered dynamic parallelism. Dynamic parallelism is especially important since it may affect the time complexity of an algorithm, or in other words, the potential speedup is a function of the problem size. (A brief introduction to complexity theory can be found in [11].)

A loop and a procedure are very similar in that both give rise to many internal activities. For this reason we say that the set of activities generated by a procedure or loop invocation not belonging to any inner procedure or loop invocation, is a logical domain. If a given logical domain contains an operator that invokes an inner

procedure or loop, then that inner invocation itself comprises a new logical domain. An activity belongs to exactly one logical domain.

The purpose of the unfolding interpreter is to generate large numbers of activities for execution by waiting processors. The more asynchronous the interpretation of a program, the greater the number of activities that might be generated and become ready for execution at any given time [3]. To keep track of the activities, each is given an activity name unique throughout the system. All tokens input to the same activity carry (along with a value) the same destination activity name which is used to group together the operands of a single activity for execution. The unfolding interpreter is the set of rules that, in conjunction with the particular program schema being executed, govern how activity names are generated. An activity name has the form u.i.α.s where[5]

1. each operator in a dataflow procedure is given a unique label s ;
2. each procedure α is encoded as a structure α, such that the value α[s] is the encoding of the operator labeled s ;
3. i.α.s identifies the i^{th} execution of operator s in procedure α ;
4. u is the unique name or context of the logical domain; all activities in logical domain u have the same domain context u .

We now give informal activity name manipulation rules for

---

[5]The notation for activity names used here is slightly different from the notation in [6].

function, procedure application, and loop schemata. Additional details can be found in [6]. In the following, s' denotes any successor operator of operator s (i.e., an output of s is connected to an input of s') and we require each operator to perform the work done by a fork by replicating and distributing output to each successor operator. Also, we assume that tokens are given port destination numbers so that the order of operands to each function is determined.

Functions and predicates: Operator s (Figure 2.8) executing the function $\uparrow$ as activity u.i.α.s typically demonstrates operators of this class. The determination of destination activity names is straightforward for functions and predicates since u, i, and α are unchanged. For each successor s' of s, only the label s' (and destination port number) need be copied from the program code to yield the destination activity name u.i.α.s' for the tokens output from s .

Procedure application: The elements of procedure application f(x) were demonstrated in Figure 2.6 . The A operator creates a new logical domain and passes the argument value to that domain. Let the activity name of an instance of A be u.i.α.s . The context u' of the new logical domain is generated by "stacking" the activity name of the A^{-1} mate operator of A so that u'=u.i.α.s' . Thus the destination activity name for initiation of the procedure is u'.1.β.b where the i field is initialized to the (arbitrary) value 1, and β is the procedure value (actually a pointer to memory holding the procedure code) which arrived on line f, and b is the standard name of the BEGIN operator in every

procedure. The BEGIN operator is an identity function and so was described above. The END operator reverses the effect of A and returns the result back to the calling logical domain by "unstacking" its activity name to reveal the logical return address $u.i.\alpha.s'$. Finally, A^{-1} (in the calling domain) receives the result value and acts as an identity function with respect to that result.

We note that the "stacking" of context within individual activity names implies an activity name of unbounded length. However, the implementation length of an activity name can be limited to reasonable size by use of a unique number generator (which can be distributed) as discussed in [19].

Loop initiation and execution: Loop initiation and termination (Figure 2.7) is similar to procedure application, where the L and L^{-1} operators change the context just as do the A and A^{-1} operators, respectively. Changing context creates a logical domain and allows independent executions of the same loop to go on concurrently. This commonly occurs with nested loops (i.e., the outer loop creates n initiations of the inner loop) and is a major source of parallelism in our machine. Every activity executed in a loop logical domain has u' in the context field of its activity name. Moreover, for each initiated loop, a unique activity name must be created for each repeated execution of the same operator. This is done by the D box which simply increments the i (or iteration count) field of the activity name (set to 1 by the L box at loop initiation). Finally, at loop exit the

token enters the D^{-1} box which simply sets the iteration count back to its original value, 1, and the L^{-1} box "destroys" the loop logical domain by recreating the original context.

2.4 A Program and Analysis of its Time Complexity

The following Id procedures perform matrix multiply on two matrices a and b, of dimension ℓ-by-m and m-by-n, respectively. These procedures are not the simplest way to write matrix multiply in Id, but this method was chosen to simplify complexity arguments as well as to reduce simulation costs.

```
procedure transpose (b, m, n)
   (initial trans ← Λ
    for i from 1 to n do
        new trans ← append(trans,i,(
            initial row ← Λ
            for j from 1 to m do
                new row ←append(row,j,b[j,i])
            return row))
    return trans) ;

procedure mmt (a, bt, ℓ, m, n)
   (initial c ← Λ
    for i from 1 to ℓ do
        rowa ← a[i] ;
        new c ←append(c,i,(
            initial d ← Λ
            for j from 1 to n do
                colb ← bt[j] ;
                new d ← append(d,j,(
                    initial innerprod ← 0
                    for k from 1 to m do
                        new innerprod ← innerprod + rowa[k]*colb[k]
                    return innerprod))
            return d))
    return c);
```

Matrix multiply is invoked by the call mmt(a,transpose(b,m,n),ℓ,m,n).

The following analysis concerns the procedure mmt that multiplies

matrix a and the transpose of matrix b. The purpose of the analysis is to describe how the unfolding interpreter is able to cause large numbers of activities to be created with corresponding reduction in time complexity. For the moment we assume unbounded resources, we also ignore communication and memory conflicts as these are considered in Sections 3 and 4 . Finally, select and append are assumed to require constant time (Section 3.2.4).

The i-loop of procedure mmt (the outermost loop) produces all values for i in time linearly proportional to ℓ, i.e., $O(\ell)$ time (see Figure 2.10a). However, once all values of i from 1 through ℓ have been computed (or even before then), instances of the j-loop (there are ℓ of them) can begin execution, each of which requires O(n) time (Figure 2.10b). The result of each j-loop having produced all values of j from 1 through n is that n instances of the k-loop will have been created by each j-loop; that is, a total of ℓn k-loop instances will have been created. Each k-loop instance computes an inner product in O(m) time. When a k-loop has completed, it returns its value (an inner product) to the j-loop that created it. Thus each j-loop instance receives one inner product from each of n completed k-loops. Each j-loop collects these n inner products in O(n) time to form one row, d, of the result matrix. Finally, each j-loop returns its row to the i-loop which collects the rows together in $O(\ell)$ time to form the result matrix c. Adding up time in the order described, the overall processing time complexity of procedure mmt is $O(\ell)+O(n)+O(m)+O(n)+O(\ell) = O(\ell+m+n)$, assuming that at least $O(\ell n)$ processors are available. The corresponding time complexity for a sequential machine is $O(\ell mn)$. This significant difference in time

complexity demonstrates the effect of dynamic parallelism, whereby the unfolding interpreter can generate large numbers of concurrent activities and thus demand processors in place of time.

3.0 THE ARCHITECTURE

3.1 Principles

Three basic principles have guided the design presented below. First is concurrency achieved through distribution. This is the most basic behavior we are trying to achieve in activity execution, token transmission, and structure access. In practice this means that although it is desirable to achieve, for example, a short access time for structures, it is more important to design for large numbers of slow but concurrent accesses than to design for a few accesses that are fast but sequential. Distribution, however, must be tempered by a second principle, locality, meaning that activities logically close together should be executed physically close together. We have selected the logical domain (Section 2.3) to be the unit of localization. That is, each logical domain is confined to some small sub-section of the machine since the activities within a logical domain are more likely to communicate with one another than with activities outside that domain. A third principle, redundancy, can affect both concurrency and locality. For example, the memory system may keep multiple copies of the same structure value in disjoint areas of the machine to allow concurrent access to local copies of information.

We wish to emphasize that the design discussed here is intended to help discover how dataflow programs behave, and to test some ideas for exploiting that behavior. It is not intended to be a final design. With that in mind, we mention two important design goals that we feel are more easily met in dataflow than with a von Neumann system but which we have not as yet attacked. These include:

- modularity: The machine should be constructed from only a few different component types, but internally these components will probably be quite complex (e.g., a processor).
- reliability and fault-tolerance: Components should be pooled, so removal of a failed component may lower speed and capacity but not the ability to complete a computation. New opportunity in this area is evidenced by the use of redundant values in the memory system which may prove useful in case a copy of data is lost through component failure.

3.2 Description of the Machine

3.2.1 Units of Measure -

We have experimented with various machine configurations and component speeds by detailed, deterministic simulation.[6] The following paragraphs describe the machine in detail according to a standard configuration. Unless otherwise stated for any particular experiment, all parameters assume their standard value. Time is referenced in terms of time units.[7] Physical capacities, such as

---

[6]The simulator itself comprises a 4500 line program written in SIMULA and runs on a PDP-10.

---

[7]When necessary, for example, to determine the feasibility of a device operating in x time units, we equate one time unit with 100 nanoseconds.

storage words or queue lengths, have no physical limit in the simulation. Although finite working storage at various points in an actual machine can lead to deadlock if exceeded, we feel that work on detailed deadlock avoidance schemes at the architecture level is premature. (Deadlock at the Id program level is impossible.)

3.2.2 A Physical Domain -

A physical domain (Figure 3.1) comprises an interconnection of processing elements (PE), memory controllers (MC), and memory boxes (M). All PEs within a physical domain are connected to a pair of shift-register token buses. The buses are connected at their ends (points A and A' in Figure 3.1) to form a pair of counter-rotating rings. Each ring is partitioned into a number of token slots and each slot is either empty or holds one fixed-length token. There is one token slot per ring per PE. Physically, each token carries its <value, destination activity name> pair, as well as a physical PE destination address (explained later). We assume this plus other control information totals to 100 bits per token. The rings shift together, so each shift brings two new token slots to the front of each PE. If the PE's address matches the physical PE destination address on a token facing it, the PE removes the token from the ring (this also produces an empty slot). A PE may fill any empty ring slot facing it with an output token. For the standard configuration we have assumed a token bus shift of 4.0 time units, or equivalently a maximum of one token in and one token out of each PE every 2.0 time units.

The basic unit of computation is the activity. When a result

token is produced by a PE, the PE evaluates an assignment function that maps the token's logical destination activity name onto a physical PE address. Any two PEs intending to send a token to the same activity must use the same assignment function. Given a fixed number of PEs and an unbounded number of activities, more than one activity may be assigned to the same PE for execution. Thus each PE must accept all tokens that are sent to it and sort those tokens into groups by activity name. When all input tokens for an activity have arrived, the PE may execute that activity. At the end of execution, the PE queues the activity's output tokens to await empty token bus slots.

Structure values are not explicitly carried by tokens but rather are kept in a memory, so a token need only carry a pointer to the structure that it logically transmits. (Since a given activity may manipulate only a small part of a structure, a great deal of communication load can thus be avoided, though we emphasize again that this memory system is not seen by the Id programmer.) In the standard configuration, four PEs are connected together and to a memory controller by a single-message-at-a-time local bus. Each memory controller is a fairly sophisticated machine that controls the random-access memory box (assumed to be interleaved arrays of 32-bit words) associated with it. All memory controllers in a physical domain are themselves interconnected by a single-message-at-a-time global bus so that every PE has (indirect) access to any structure value held in the machine. Although the memory of the dataflow machine is distributed over the memory boxes, it is organized into one unified address space to facilitate sharing. For example, say a PE is

to execute a select on structure α. The PE then asks the memory system to do the operation by sending a request over the local bus to the memory controller to which that PE is attached, called the local memory controller. If α is available in the local controller's memory box, the controller can carry out the request on α and then return a response to the requesting PE. If α is not local, then the local memory controller must forward the request to the proper distant memory controller for action. The distant controller then returns its response to the local controller. Both the request and response messages traveling between memory controllers move on the global bus. Finally, whether α was local or distant, when the local memory controller has the result it is returned to the PE that initiated the original request.

3.2.3 Inside a PE -

Figure 3.2 shows details of a PE organized as a pipelined processor.[8] Each box in the figure is a unit that performs work on one item at a time drawn from FIFO input queue(s). Logically, tokens enter the PE from rings at the top of the figure while new tokens are output to the rings at the bottom. The connection to the local bus is shown at the left.

Functional Operation of a PE

The function of the sorter is to group tokens by activity name.

---

[8]The PE architecture described here was pipelined to simplify coding the simulator. The degree of concurrency appropriate within a PE has not yet been determined.

The sorter requires 4.0 time units to process each token with the aid of an associative table keyed on activity names. When an activity name is presented, the table returns a pointer to the list of tokens gathered for that activity (kept in a fast local scratch pad memory). The token is then added to that list. Each token carries a number specifying the total number of input tokens required to complete an activity. If the newly arrived token is the last token, an activity item pointing to the list of input tokens is created and sent to the code fetch box.[9] Each successive box then adds more information to the activity item as it passes through the PE until processing is completed.

Upon receipt of an activity item with name u.i.α.s, the code fetch box is responsible for retrieving the operation code α[s]. To speed operation, the code fetch box employs a local cache to hold previously fetched dataflow code. If the needed code is already present in the cache, the code is immediately added to the activity item which then moves to the next stage in the PE. If the code is not present in the cache, a code structure select request is placed in the local bus input queue and the activity item is held in the code fetch box until the selected item is returned. (This does not prevent the code fetch box from initiating work on the next activity item in its input queue.) Responses from the memory system to the PE are returned over the local bus. These responses are queued and then serviced in FIFO order by the appropriate box within the PE. For the case at

---

[9] This description of code fetch corresponds to the simulator implementation. Alternatively, code fetch could be initiated as soon as the first token for an activity arrives, or in fact as soon as the assignment function for a given logical domain is known.

hand, a response to a code fetch request contains the activity's operation code and the information necessary for the PE to construct the output tokens' destination activity names. The code fetched is also entered into the PE cache with keys α and s . No delay is charged in the simulator for code fetch if the code is found in the cache; otherwise the time charged to the activity item is the memory response time for the associated request. Note that the order in which activity items leave the code fetch box is not necessarily the order in which they entered.

After code fetch, the activity item moves to one of two boxes. The data fetch box issues memory requests and receives memory responses for structure operations (select and append); the arithmetic/logical unit, or ALU, carries out all dataflow operations not requiring the memory system (such as SWITCH, +, etc.). The data fetch box operates just as code fetch, sending requests and receiving responses from the memory, except that there is no cache. Thus all select and append operations are actually carried out by the memory system.[10] The time an activity item remains in the data fetch box is determined solely by the response time of the memory to each request. On the other hand, each ALU operation is fixed at 10 units of time.

After proceeding through either the data fetch or ALU, the activity item (with the result of the particular operation attached to it) moves to the output box. Tokens are manufactured by the output box at a rate sufficient to match the token bus, which in the standard

---

[10]We expect that a data cache would have little effect on performance unless blocks of data were fetched instead of single values.

configuration is a maximum of two tokens every 4 units of time. During this time, the box must copy the result, assign a destination activity name (Section 2.3)[11], and map the activity name to a physical PE address by evaluating the assignment function. The output box then selects the token ring that gives the shortest distance path from the present PE and places the token in the appropriate output queue. From here tokens move in FIFO order into empty token slots as they appear on the ring in front of the PE.

The Assignment Function

The assignment of activities to physical PEs is very important. A good assignment function promotes concurrency and locality, while a poor one can destroy machine performance (in Section 4 we demonstrate some results on different assignment functions).

Concurrency is achieved by distributing the activity workload over the PEs of the physical domain. Locality is promoted by mapping all activities within a single logical domain onto the same (physical) sub-domain, defined to be the set of PEs attached to the same memory controller. A physical domain with 32 PEs and four PEs per memory controller has eight sub-domains. When the number of logical domains created exceeds the number of sub-domains, several logical domains will be assigned to the same physical sub-domain and compete for PE resources. (The competition is actually at the level of the activities within each logical domain.)

---

[11] This is true except when the context u is stacked or unstacked. In such cases, we have assigned this work to the ALU and charge 10 units of time.

The following is a simple assignment function which promotes both locality and concurrency. The j^{th} logical domain to be created, in time order[12], is assigned to sub-domain (j mod q) where q is the number of sub-domains in the physical domain. Within a sub-domain, activity u.i.α.s is mapped onto PE number (s mod 4) since there are four PEs (numbered Ø through 3) per sub-domain. Figure 3.3 shows the effect of this assignment function on program execution where tokens in adjacent sub-domains do not intermingle, save for the passing of arguments and returning of results. This promotes concurrency in execution and in token transmission since the sub-domains may be active at the same time.

---

[12]This implies the existence of a centralized memory cell or other resource to keep track of the latest value of j. We chose this method for ease of simulator programming though we envision using a distributed method (with similiar effect) in an actual machine.

3.2.4 Inside a Memory Controller -

To discuss the operation of a memory controller, we must first discuss data representation. Program code is a special case of structures so the following covers both data and program representation.

Representation of Structures in Memory

Each level of a structured value v may be represented in the system in either of two[13] distinct ways, where each representation implies different time complexity for select and append operations, and different space requirements. Table I gives these requirements for a structure with n selectors at a given level. The contiguous vector (or dope-vector) representation allows for quick access and is essentially the technique used in FORTRAN, ALGOL, and other von Neumann languages. An example of a structured value v in vector representation is shown in Figure 3.4a . Select is straightforward and requires only constant time. However, since dataflow values are never modified, append requires that the vector first be copied (although sub-structures, if any, need not be physically copied as shown in [16,19]) and the new value inserted at the correct position to create the result. An important exception occurs when there is only one token referencing v (which can be determined using a reference count scheme). Here the input value v will no longer be used after the append anyway, so v can be updated in place to give the

---

[13] Another representation we have investigated is termed selector vector [19]. This representation is useful for structures which contain string or sparse integer selectors. Since the test programs studied here do not manipulate such structures, we have omitted discussion of the selector vector representation.

result directly. In this case append requires only constant time (assuming sufficient contiguous space is available for the new value if a new selector is being appended).

The alternative representation is a 2-3 tree [2] shown in Figure 3.4b . Briefly, a 2-3 tree is a "balanced" tree (the length of any path from the root to any leaf is the same) where each interior node in the tree has either 2 or 3 out-branches. A select operation may be done by a recursive search that follows a path through the tree to the leaf that contains the selected value. Each interior node holds information about the selectors below it, so a path, if it exists, can always be found. Append involves copying a path and allows sharing of common sub-trees. An algorithm for select appears in [2] while append is discussed in [30]. Both append and select have O(log n) time complexity.

The standard configuration uses vector representation for program code. Vector representation is also used for each input data structure until the first append when it is automatically converted to a 2-3 tree. The structure then remains in 2-3 tree form in anticipation of further appends. However, in many algorithms the automatic conversion on append is best overridden to reduce the time complexity of subsequent structure accesses. For example, in procedure mmt of Section 2.4 only one pointer to a given row of the result matrix exists at any given time during its formation. Thus if enough contiguous space is allocated[14] when computation of a row

---

[14]This is analogous to dynamic allocation of vectors in ALGOL and could be specified by the programmer or perhaps by compiler analysis.

begins, then each append can be done in constant time resulting in a matrix in vector format. Hence, ignoring conflicts, memory access need not increase the time complexity of matrix multiply.

The Memory Cache System

To increase concurrency and locality, we have devised a cache system (independent of the PE cache) wherein each memory controller acts as a cache to the rest of the memory system. To show how this operates, assume that a memory controller receives a select(α,i) request from a PE and that α is not local. The local controller then requests the distant controller to send a copy of the entire top level of the structure α to the local controller, rather than have the distant controller do the select. When the copy is received by the local controller, it makes an entry with key α in an associative table which points to where the copy α' of α is locally held. Any subsequent operations on α can then be carried out on α' independent of those carried out on α. Recall that this can happen only because dataflow structures are never modified. Also, the internal representations of the structure at α and at α' need not be the same.

Functional Operation of a Memory Controller

Figure 3.5 shows a detailed view of a memory controller and associated memory box. The transmission of an entire level of a structure resulting from a copy request can require significant global bus time. For this reason, and to make sender-receiver coordination easier, there is a separate copy processor and memory port provided for copy data transmissions. Thus when a distant memory controller's request processor services a copy request, it gives the request to the copy processor which then transmits the structure over the global bus to the copy processor at the local controller. In the standard configuration, the sending copy processor transmits only the leaves of 2-3 trees which are then converted back to 2-3 format by the receiving copy processor. Structures in vector format are transmitted and stored directly.

Concerning rates and capacities, the time charged to a memory controller to process a request depends upon several factors including data representation, memory speed, and memory controller speed. In general, the simulator charges 6 units of overhead for each request message, plus the time to do the actual operation. Select and append require the number of operations specified in Table I multiplied by 1.5 time units per word. Each select and append request message is four words and requires 0.4 units of transmission time per word. Structure copy bus transmission time is as specified in Table I multiplied by 0.4 time units per word.

4.0 MACHINE PERFORMANCE

We envision a full scale machine to be an n-dimensional interconnection of some large number of physical domains (Section 3) although we have limited this initial study to a single physical domain. Our intent was to answer some simple questions: Does the unfolding interpreter actually provide for more asynchronous computation? If so, does it allow for increased speed of execution as more processors are added to the pool? And finally, to what extent do our working hypotheses -- the anticipated relationships between locality, distribution, concurrency, and redundancy -- actually operate?

The experiments involved running dataflow programs on a simulated machine which monitored the programs' executions. All programs[15] were written in Id and then machine compiled and loaded into the simulator for execution. Many experiments were repeated on more than one type of dataflow program, though due to cost not all experiments could be repeated on all programs. Also, only the mmt procedure part of matrix multiply, Section 2.4, was used in the matrix multiply experiments presented here. (The presence of procedure transpose in matrix multiply has no effect on the overall time complexity since transpose requires only O(m+n) time. This prediction was confirmed by test cases as was the distribution of results from transpose to ensure satisfaction of procedure mmt's input assumptions.) Finally, we have

---

[15] The programs used were matrix multiply (procedure mmt), optimal binary search tree generation, Gauss-Seidel linear equation solver, Gaussian elimination, recursive quicksort, and fast Fourier transform (both an iterative and a recursive version).

4.0 MACHINE PERFORMANCE

We envision a full scale machine to be an n-dimensional interconnection of some large number of physical domains (Section 3) although we have limited this initial study to a single physical domain. Our intent was to answer some simple questions: Does the unfolding interpreter actually provide for more asynchronous computation? If so, does it allow for increased speed of execution as more processors are added to the pool? And finally, to what extent do our working hypotheses -- the anticipated relationships between locality, distribution, concurrency, and redundancy -- actually operate?

The experiments involved running dataflow programs on a simulated machine which monitored the programs' executions. All programs[15] were written in Id and then machine compiled and loaded into the simulator for execution. Many experiments were repeated on more than one type of dataflow program, though due to cost not all experiments could be repeated on all programs. Also, only the mmt procedure part of matrix multiply, Section 2.4, was used in the matrix multiply experiments presented here. (The presence of procedure transpose in matrix multiply has no effect on the overall time complexity since transpose requires only O(m+n) time. This prediction was confirmed by test cases as was the distribution of results from transpose to ensure satisfaction of procedure mmt's input assumptions.) Finally, we have

---

[15] The programs used were matrix multiply (procedure mmt), optimal binary search tree generation, Gauss-Seidel linear equation solver, Gaussian elimination, recursive quicksort, and fast Fourier transform (both an iterative and a recursive version).

$$\text{efficiency} = \frac{\text{cummulative actual busy time of all ALUs}}{\text{cummulative potential busy time of all ALUs}} * 100$$

i.e., efficiency is directly related to the mean ALU duty cycle. Note that execution time was reduced by up to a factor of 13.7 for procedure mmt (43468 time units for a system with 1 PE, 3123 units for 60 PEs).

The second point shows that computation speed increases (up to a limit) as available processor resources increase. In the Introduction we noted such behavior would be desirable because it demonstrates independence of physical processor configuration (both size and shape) from the programs executed. We expect this to be important in easing problems in programming, scheduling, and reliability (fail-soft). Moreover, the existence of performance fall-off can be blamed on an unsophisticated assignment function that forces computations to be distributed over the physical domain even when such distribution is inappropriate. The result is under-utilized PEs and increased communication delays since there is an increase in mean token distance and a decrease in the probability that any given structure is local to the PE needing it.

The other curves in Figure 4.1 show similar, if not as dramatic, results for other programs except for the iterative fast Fourier transform (FFT) which did not do well at all. Although the behavior of iterative FFT is not completely understood, it appears to be a combination of several factors. These factors include scheduling anomalies and unwanted synchronization imposed by the append operation in constructing structured values. In some programs, this

synchronization can be removed by using a new pipelined (or stream[16]) append operation resulting in dramatically improved performance. However, we have so far been unable to devise an iterative FFT (with or without streams) that performs as well as recursive FFT. Recursive FFT performs well because it uses a "divide-and-conquer" method, and because the size of the data structures manipulated progressively decreases at each recursive call. Under dataflow, pairs of divide-and-conquer recursive calls are done asynchronously and therefore in parallel. This means that recursion is often faster than looping.

4.2 Complexity Experiments

Recall from the analysis of matrix multiply in Section 2.4 that the processor time complexity (ignoring communication complexity) was O(r) for procedure mmt on two r-by-r matrices. To determine the actual execution time complexity, we performed the speedup experiment for all problem sizes from r=2 through r=8 . The bottom curve in Figure 4.2 plots the minimum time for each speedup experiment against r and shows that the unfolding interpreter was indeed operating with O(r) execution time complexity. In addition, when processor efficiency (ALU duty cycle) is accounted for, processor utilization is $O(r^2)$ as predicted by the analysis. To explain the other curves in Figure 4.2, we must first discuss time complexity a bit more.

---

[16] Id provides for stream variables [6] in which a sequence of values may be pipelined through a program. However, the simulator system is not yet capable of handling streams.

Processors are just one of the resources being demanded by a program. We must also consider other resources -- memory controllers, the global memory bus, the token bus -- and their effect on actual execution time complexity. Exactly one copy of each row of the input matrices is assumed to exist initially and these rows are initially distributed over the available memory boxes. The analysis in Section 3 assumed that the number of PEs available, and thus memory controllers, is $O(r^2)$ while the number of input rows is $O(r)$. Thus there are plenty of controllers and memory boxes. However, procedure mmt requires access to all elements of each row r times since each row participates in r inner products. So each of as many as 2r memory controllers (the number of rows) sees $O(r^2)$ accesses. Thus the memory controller time complexity is $O(r^2)$. (Note that it is irrelevant to the memory controller complexity analysis whether a row is copied from a distant to a local memory controller or not.)

The global memory bus experiences an even heavier demand than the memory controllers. There are $O(r^2)$ accesses to each of the possible 2r memory controllers which must send these elements over a single fixed bus. Thus the global memory bus time complexity is $O(r^3)$. (This might not be the case if more than one physical domain were present.)

Finally we consider the bi-directional shift register token bus with its intensive intra-logical domain communication. To determine the token bus time complexity, note that both the number of logical

domains and the physical sub-domains to which they are assigned is $O(r^2)$ since for these experiments $O(r^2)$ PEs are available. Since the intra-logical domain communication on the bus is concurrent (Figure 3.3), then these domains are non-interfering and have essentially constant token communication time within each domain. (This agrees with experimental results where the overall mean token communication distance was always between one and two shifts when the standard assignment function was used.) However, all r^2 inner product domains were originally produced from a single initial domain (the outer i-loop). This means a chain of tokens must have passed from this initial domain to each of the r^2 inner product logical domains distributed along the bus. But the length of the bus is directly proportional to the number of processors -- $O(r^2)$. Thus the length of the longest token path from the initial logical domain to the last of the r^2 inner product logical domains is $O(r^2)$ -- the token bus time complexity.

By the above analysis the global memory bus is the limiting resource and constrains performance to $O(r^3)$. Nevertheless, the bottom line in Figure 4.2 is (almost) linear because the constants in the global memory bus time complexity term do not allow it to become dominant when $r \leq 8$. For $r>8$ the above analysis predicts that the apparently straight line will eventually become a cubic. Due to constraints in the simulator on the PDP-10 we were not able to go beyond $r=3$; but to verify the expected behavior, we instead unbalanced the machine and lowered the memory system speed by the factors indicated in the other two plots in Figure 4.2 . Lowering memory system speed increases the constants in the global bus and

memory controller time complexity, causing the machine to reach more quickly the predicted execution time complexity of $O(r^3)$.

Another example of a time complexity experiment is Figure 4.3a which shows measured execution time for the two versions of FFT, recursive and iterative, previously discussed. Processing time complexity analyses for these two programs are O(n) and O(n log n) respectively, as borne out by the slight curve in iterative FFT. These times are both O(n log n) on a sequential machine. Figure 4.3b shows time complexity graphs for a Gaussian elimination algorithm to solve simultaneous linear equations. The time complexity for a single processor is $O(n^3)$ while the processing time complexity for our dataflow system is $O(n^2)$ as demonstrated by the experimental results. Similarly, recursive quicksort has an average time complexity of O(n) on the dataflow machine as shown in Figure 4.3c ; on a sequential machine quicksort has O(n log n) average time complexity.

We have found time complexity analysis to be a useful tool in understanding dataflow machine behavior, as well as aiding in selection among design alternatives. We note that the overall time complexity has been shown to include token bus, memory bus, and memory controller time complexities which together represent a "communication time complexity" factor not explicitly present in algorithmic analysis on standard von Neumann systems. It is clear that communication time complexity is important, and we expect it may become the dominant term in future systems and algorithmic design. This conclusion is very similar to that reached by Sutherland & Mead in their speculative article [29].

4.3 Flow Analysis

A major difficulty in evaluating a system is devising adequate measures. Section 4.2 showed that time complexity can be a useful tool for understanding system behavior. The usual measures such as queue lengths, time to execute a program, and the duty cycle of various units are also helpful. However, flow analysis[17] was most useful in determining resource balance and the location of bottlenecks (imbalances).

Both activity flow analysis and memory request flow analysis are used. Let the term item refer to a token or to an activity item, and let the block diagram of a PE (Figure 3.2) represent a sequence of stations through which items must pass (each queue is also interpreted as a station distinct from the station it serves). Activity flow analysis consists of measuring the mean time spent by all items at each station, and interpreting this time as the time spent at a station by some hypothetical "mean" item. For instance, Table II shows an activity flow analysis for two runs of procedure mmt at r=7 where column (a) is the result for the standard configuration. The mean token time in the sorter queue over all PEs was measured as Ø.81 units. Thus we say that the hypothetical token spent Ø.81 time units waiting in the sorter queue. Similarly for the other measures, though of course no one real activity item passed through both a data fetch box and an ALU box. In these cases we say the hypothetical mean item spent $(d/n)*t_d$ of its time in a data fetch box (where d of the n total

---

[17] Flow analysis is related to "longest path" analysis discussed in [31].

activities passed through a data fetch box with measured mean time t_d), and $((n-d)/n)*t_{ALU}$ of its time in an ALU box. For example, in the case of Table IIa 15.6% of the activity items passed through data fetch boxes with mean service time of 21.79 units to yield a data fetch time of 3.4 . The sum of the times for all boxes and queues listed is the cycle time of a hypothetical activity. Table IIb shows a flow analysis for the same program but on a machine differing from the standard configuration only in the speeds of the local and global buses. These analyses pinpoint the system imbalances. In fact, our dataflow machine was initially set with local and global bus speeds corresponding to Table IIb . They were then changed to what is now the standard configuration to better balance the system.

Memory request flow analysis creates a hypothetical mean memory request and measures its time in the system from creation until its originating PE receives and processes the corresponding hypothetical response (see Figure 3.2 and 3.5). The fact that not all messages require the global bus or a distant memory controller is accounted for by proportioning the measured times by the fraction of requests that did access the global bus and some distant memory controller. Table III presents the memory request flow analyses for the same runs that produced the respective activity flow analyses of Table II. Again the imbalances in resources are immediately evident.

We consider the weighted means given by flow analysis to be more indicative of overall performance for an asynchronous (dataflow) machine than the corresponding raw means. In addition, flow analysis factors out "waiting time" such as the time between the arrival of the

first token of an activity and the last token. Although waiting time is important in considering buffer requirements, it appears otherwise to have little effect on average machine performance.

4.4 Locality, Concurrency, Distribution

A primary effect we wish to achieve is concurrency of execution, induced by distributing (more or less) independent activities over many processors. But activities should not be distributed indiscriminantly -- program locality should be considered. Locality is evidenced in token and memory communication distances and is determined largely by the assignment function used.

Figure 4.4 shows a speedup curve for procedure mmt with r=7 for four different assignment functions called A, B, C, and D. Following the principle illustrated by Figure 3.3, assignment functions A, B, and C map the j^{th} logical domain that is created onto physical sub-domain d by the formula

$$d = j \bmod q$$

where q is the number of physical sub-domains in the machine. Again, this confines all activities in logical domain u to physical sub-domain d, regardless of how large that logical domain might be. Assignment function A (described previously) then maps activity u.i.α.s onto PE p within physical sub-domain d by the formula

$$p = s \bmod 4$$

Consider what happens when recursive procedures or nested loops are

present in a program. In assignment function A, distinct initiations of the same procedure or loop are assigned identically within the boundaries of a physical sub-domain. The same PE then executes the same operators within those logical domains resulting in very effective use of the PE's code fetch cache.

Assignment function B is used for the standard configuration

$$p = (s+j) \underline{\text{mod}}\ 4$$

and is similar to A except that distinct initiations of the same procedure or loop assigned to the same physical sub-domain do not (necessarily) have their activities assigned identically within that physical sub-domain. The result is a "wider" distribution of activities, and a lessening in cache effectiveness, i.e., reduced locality. As evidence of locality reduction, the mean code fetch hit ratio in the PE cache was reduced from 0.93 for assignment function A to 0.82 for assignment function B, in the case of procedure mmt with 60 PEs.

A third assignment function C distributes activities within a sub-domain to a greater extent than either functions A or B by including the term i in mapping u.i.c.s to PE p within sub-domain d :

$$p = (s+j+i) \underline{\text{mod}}\ 4$$

The fourth assignment function D is present only to compare the three above assignment functions with one which distributes activites without regard to sub-domain. Function D is

$$p = (s+j) \underline{\text{mod}}\ (q*4)$$

where (q*4) is the number of PEs in a physical domain.

Another view of the comparison among the four assignment functions is offered by the activity flow analyses in Table IV for the point at 60 PEs from each of curves A-D (Figure 4.4), respectively. Note the effect of locality on code fetch, output, and token bus times. Performance differences for the locality-exploiting assignment functions A-C are slight compared to the clear performance loss of assignment function D. Although reasons for performance differences among A-C are interesting to hypothesize, we have not yet conducted sufficient experiments to explain these differences in more detail.

5.0 CONCLUSIONS

Our eventual goal is to design a system that exploits the full potential of LSI technology. To achieve this goal, we have adopted the semantics of dataflow as the basis for a programming language and machine since it allows us to avoid many of the problems that confront current multiprocessor systems.

This paper has outlined a base dataflow language and the unfolding interpreter that generates the asynchronous activities executed by the processors in our machine. Also shown was the high-level language Id and a detailed description of a part of an architecture for implementing the unfolding interpreter.

Our purpose in experimenting on this machine was not to show that it was fast in any absolute sense, but rather to answer some basic questions about dataflow and its feasibility as the basis of a

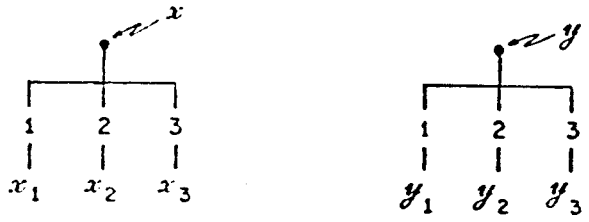
machine. In particular, we demonstrated that the unfolding interpreter can generate large numbers of activities, and that the independence of these activities allowed for increased execution speed (up to a point) with the addition of more processors to the system. We verified several expectations concerning locality, distribution, and redundancy, and their effects on the concurrency achieved in the machine. We also confirmed our analyses of program time complexity and concluded that communication time complexity is at least as important as processor time complexity. In general we feel complexity analysis is a useful tool for designers of such systems as is flow analysis for uncovering bottlenecks and resource imbalances.

Of course, much work remains to be done. In particular, we plan to revise several aspects of our initial design (e.g., the busing systems) and to extend the machine from one to many physical domains. Also planned is further research into the assignment and scheduling of activities and determination of the proper size or "grain" of an activity, aspects which are certain to have significant impact on machine performance. Other areas scheduled for investigation are the determination of the benefits of data redundancy and the incorporation of streams to avoid the unnecessary synchronization of dataflow structures.

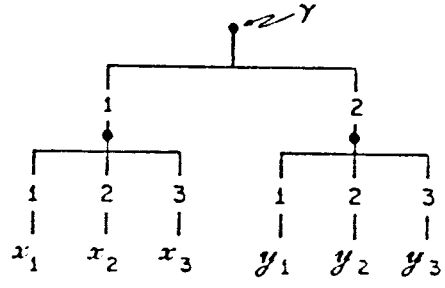
In summary, the results appear encouraging. The highly asynchronous behavior we hoped to observe was indeed found in many places to a degree suggesting that dataflow may be one way to utilize the power apparent in LSI technology, while also giving the programmer a clean and useful semantic basis [5,12].

ACKNOWLEDGEMENTS

We would like to acknowledge the work (reported elsewhere) of Arvind and Wil Plouffe on the design of Id, the base language, and the unfolding interpreter. We also acknowledge the excellent PDP-10 SIMULA system implemented by the Swedish National Defense Research Institute. We would like to thank Shirley Rasmussen for typing the manuscript, and the UCI Computer Facility for providing computer support.



(a)



(b)

Figure 2.1

Structured values

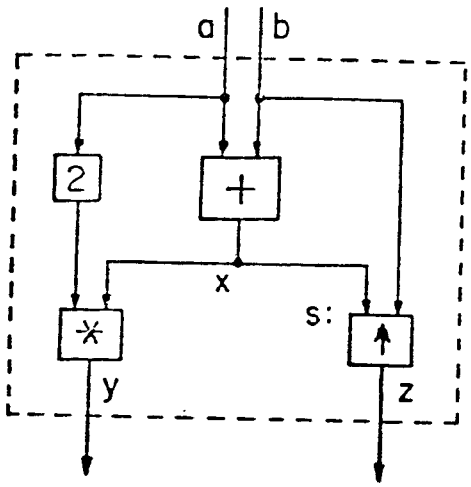
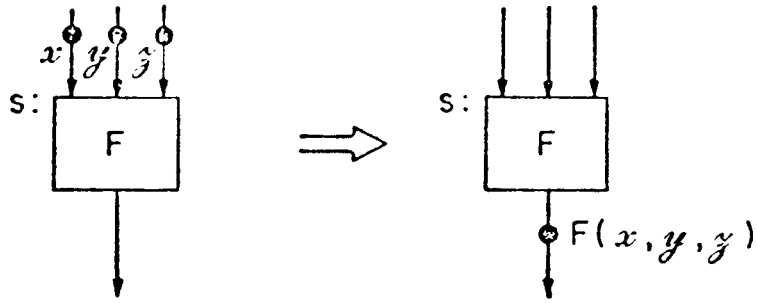
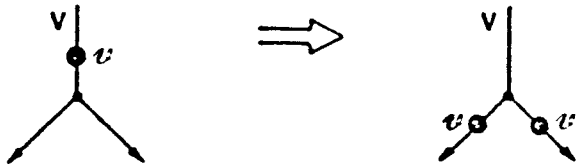


Figure 2.2

A simple Expression



(a)



(b)

Figure 2.3

(a) Execution of an operator
(b) A token encountering a fork in a line v

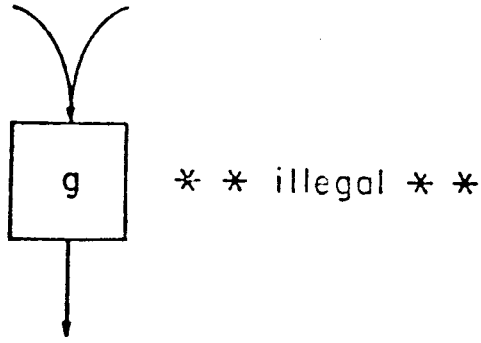


Figure 2.4

An illegal connection

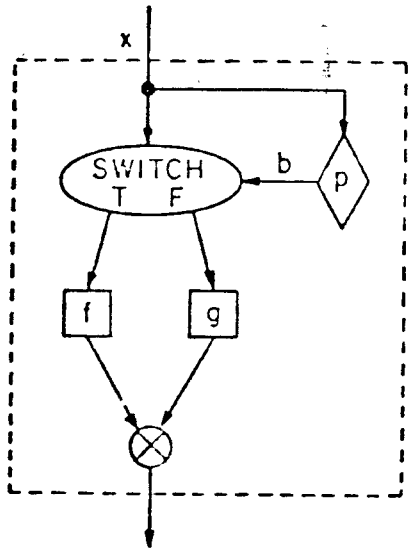


Figure 2.5

The conditional schema

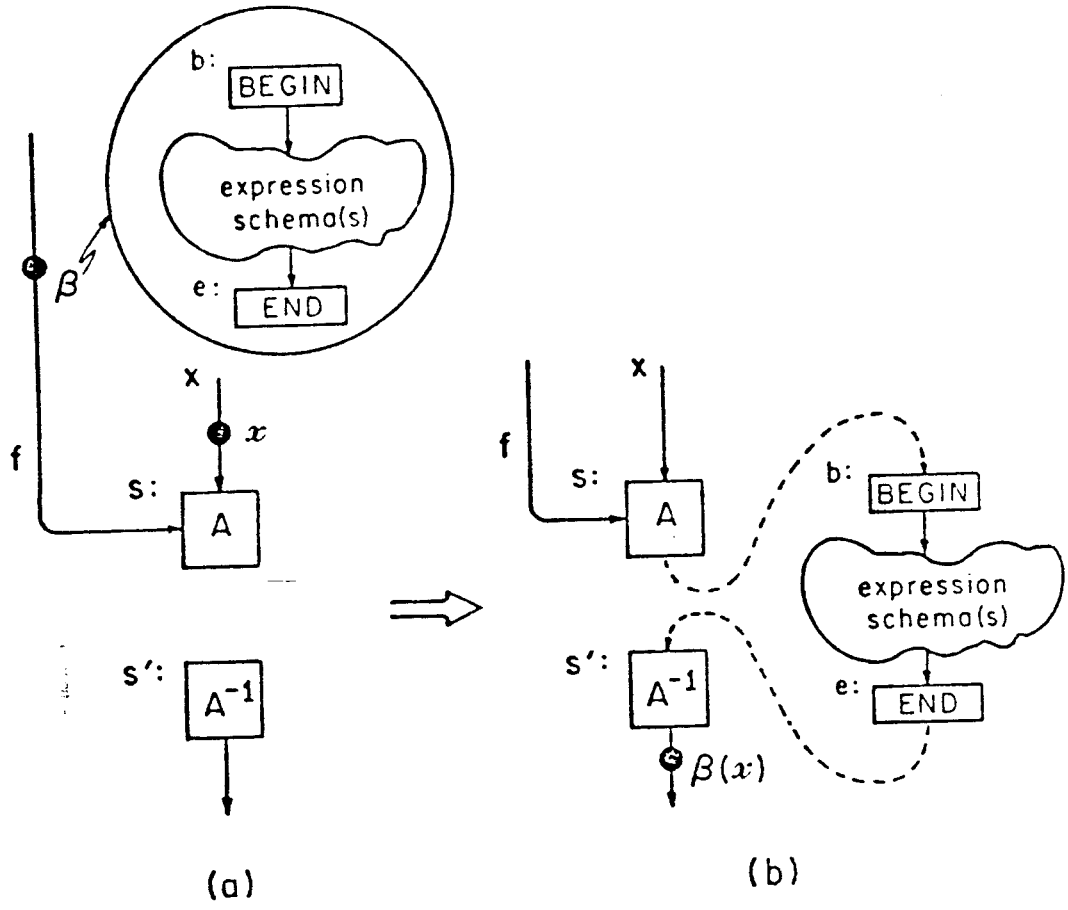


Figure 2.6

An application schema execution

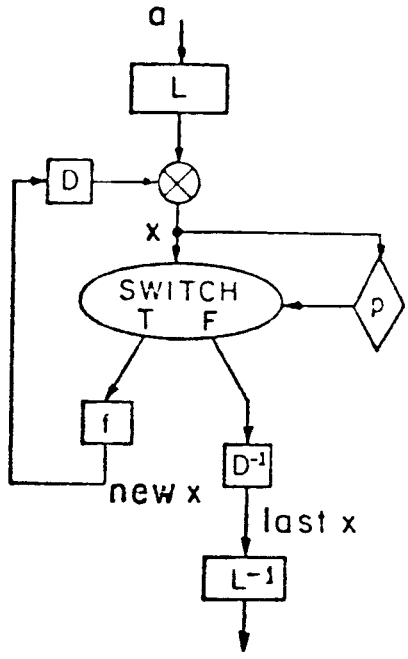


Figure 2.7

The loop schema

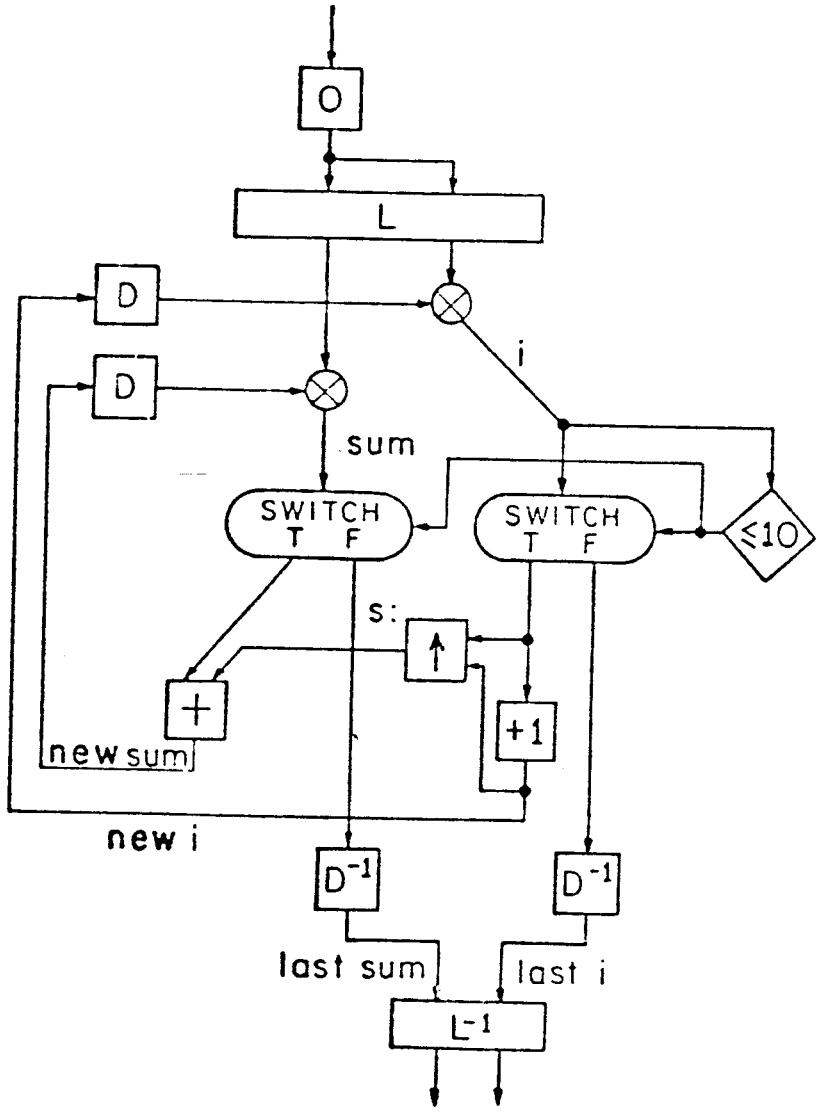


Figure 2.8

A loop over two variables

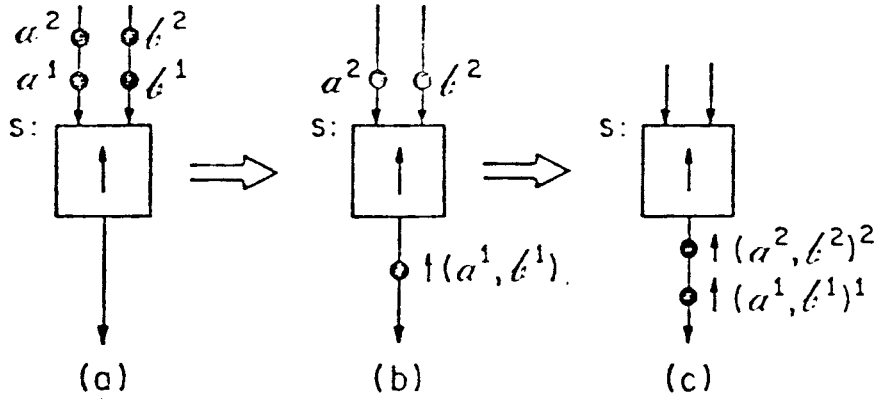


Figure 2.9

(a), (b), (c) show two executions
of a dataflow operator

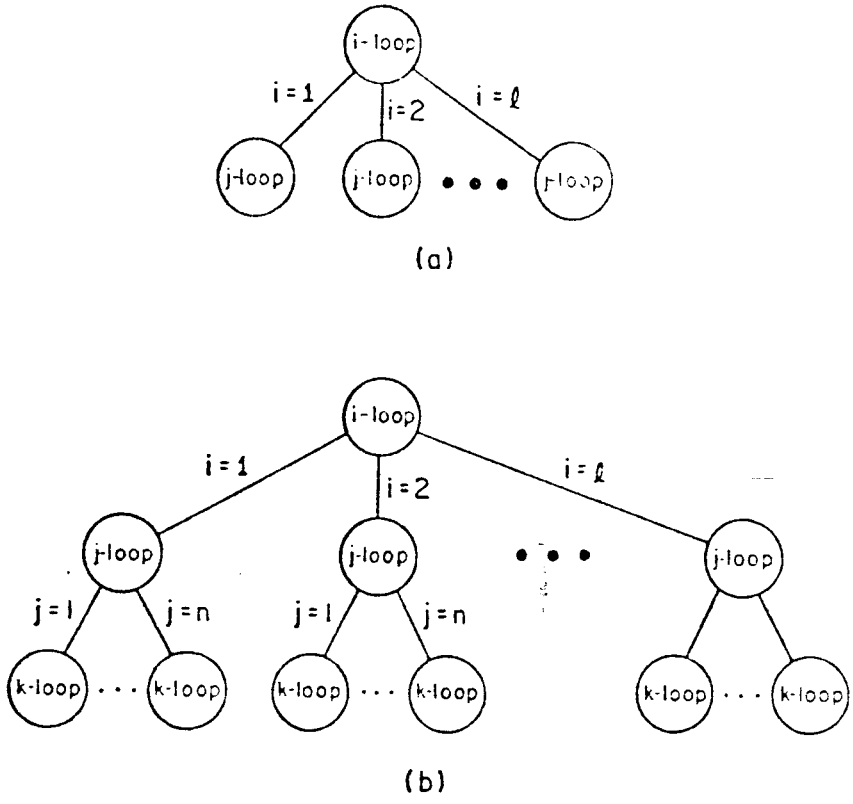


Figure 2.10

Matrix multiply complexity analysis

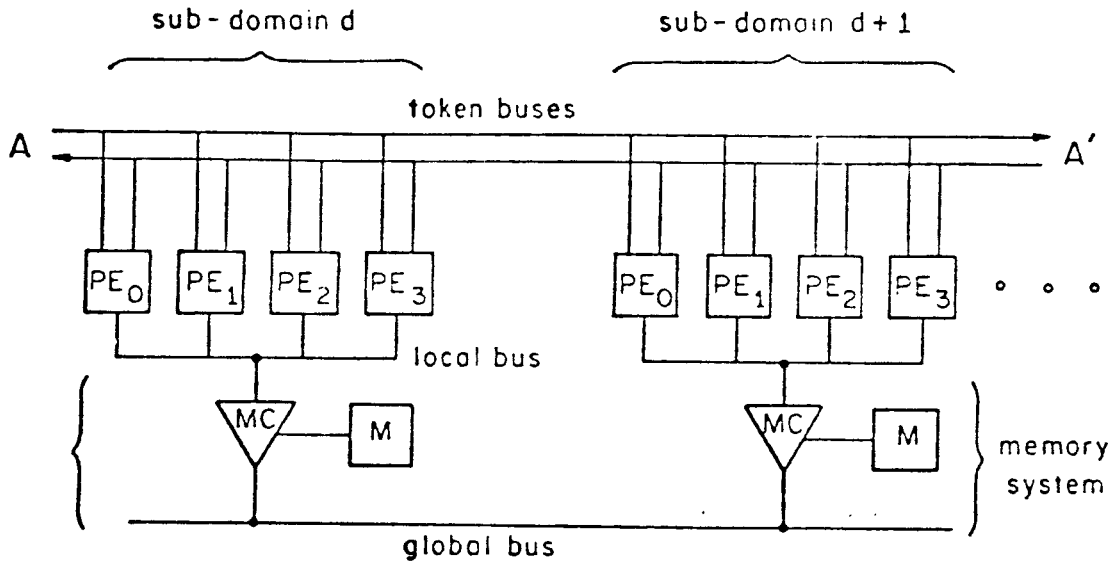


Figure 3.1

A physical domain

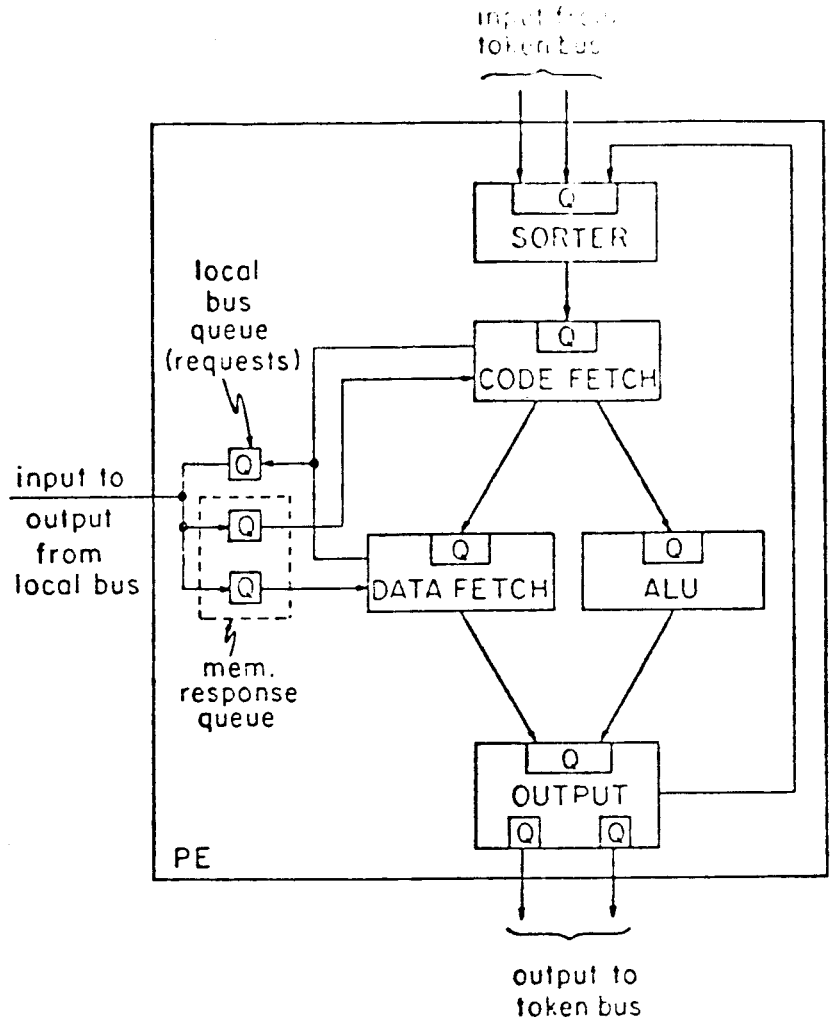


Figure 3.2

A processing element (PE)

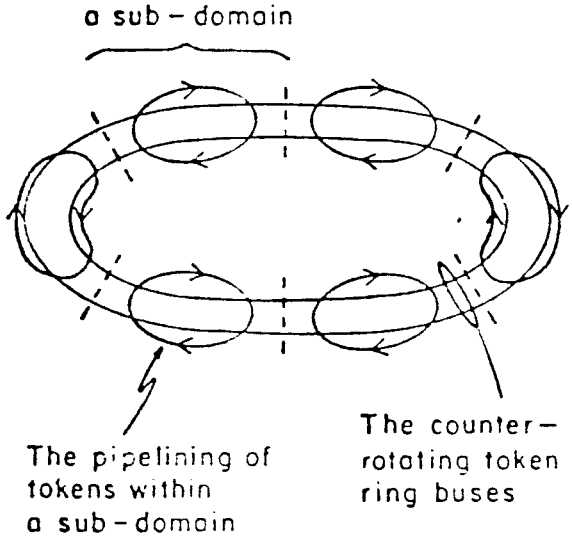


Figure 3.3

Sub-domains operating concurrently

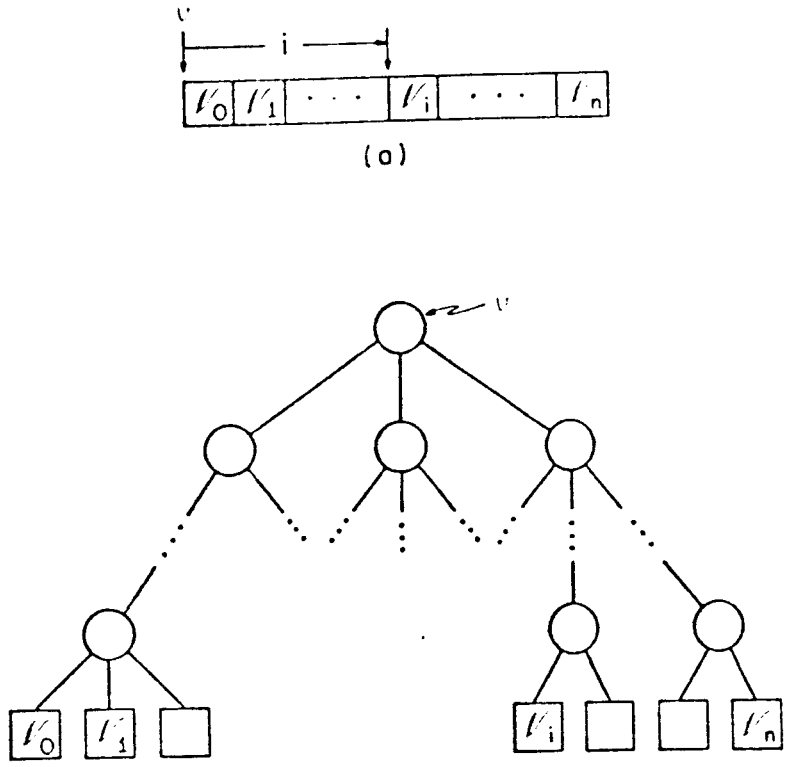


Figure 3.4

Representations of a structured value

(a) vector (b) 2-3 tree

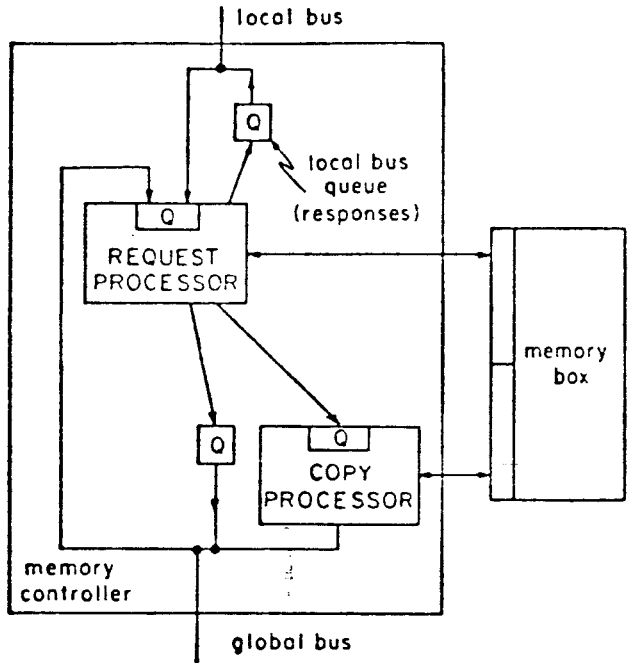


Figure 3.5

A memory controller and
attached memory box

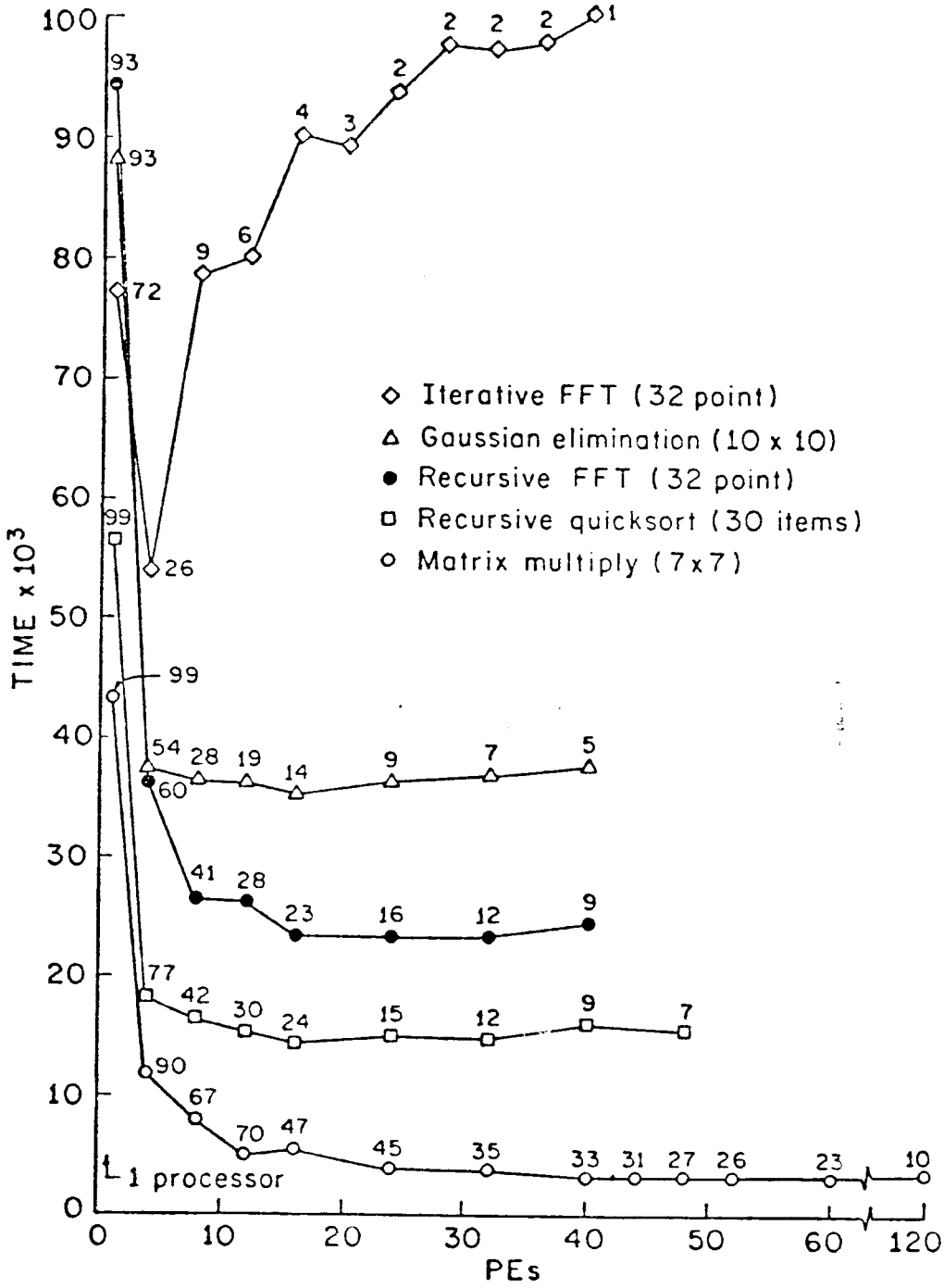


Figure 4.1

Speedup curves for five test programs, with ALU efficiency (per cent) indicated for each test point.

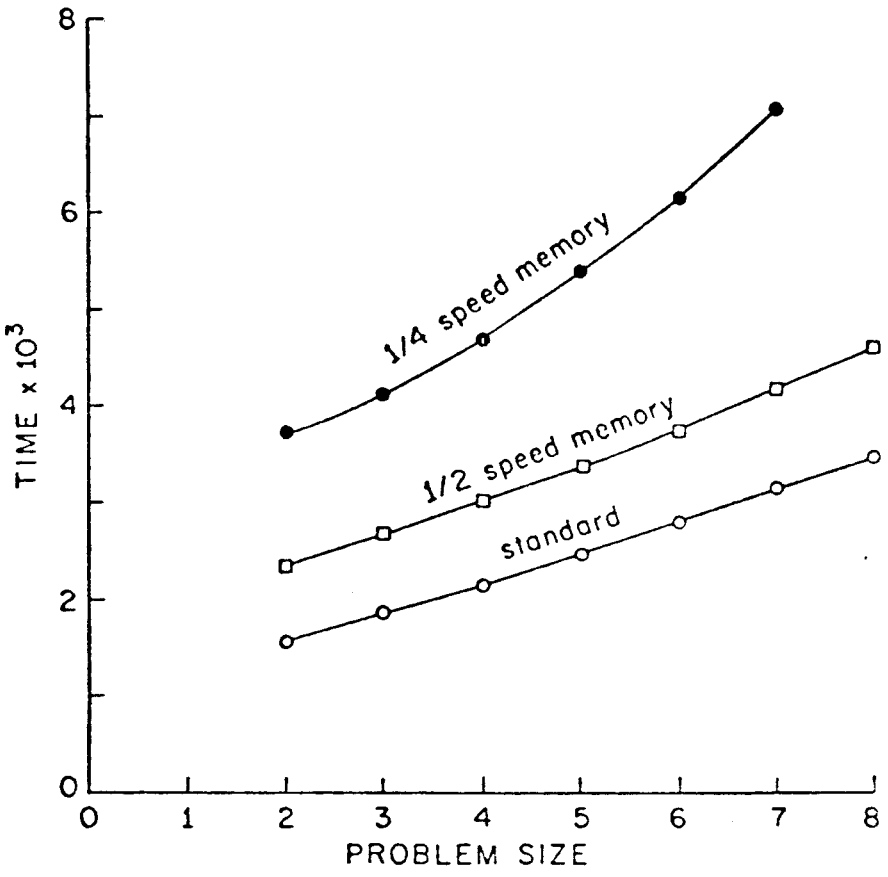


Figure 4.2

Execution time complexity curves for matrix multiply. The family of curves shows that as memory speed is slowed, communication time complexity effects begin to dominate.

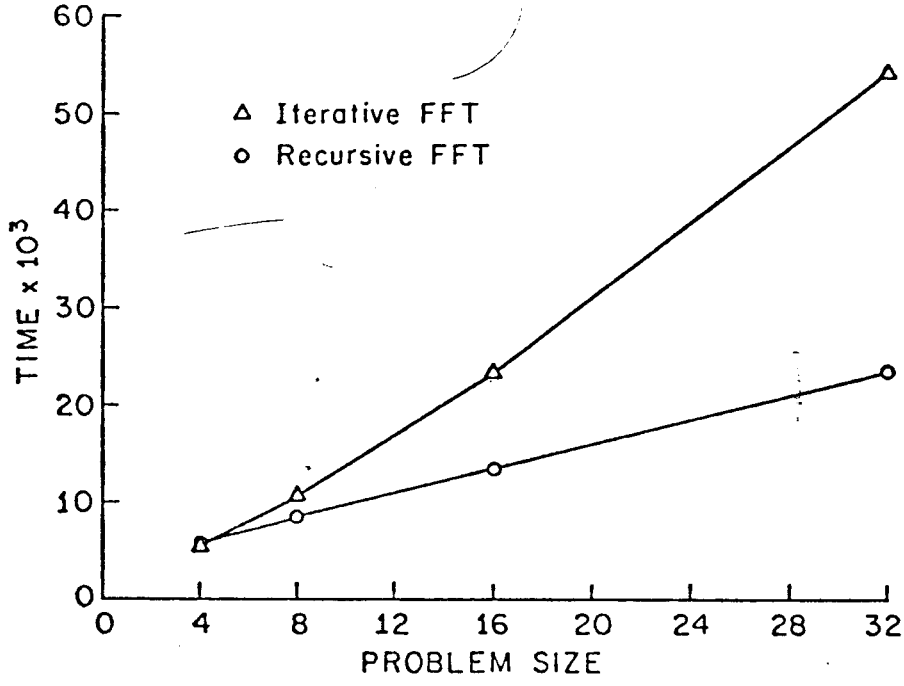


Figure 4.3a

Execution time complexity curves for recursive and iterative FFT: O(n) and O(n log n) with optimal number of PEs, respectively, while both are O(n log n) on a sequential machine.

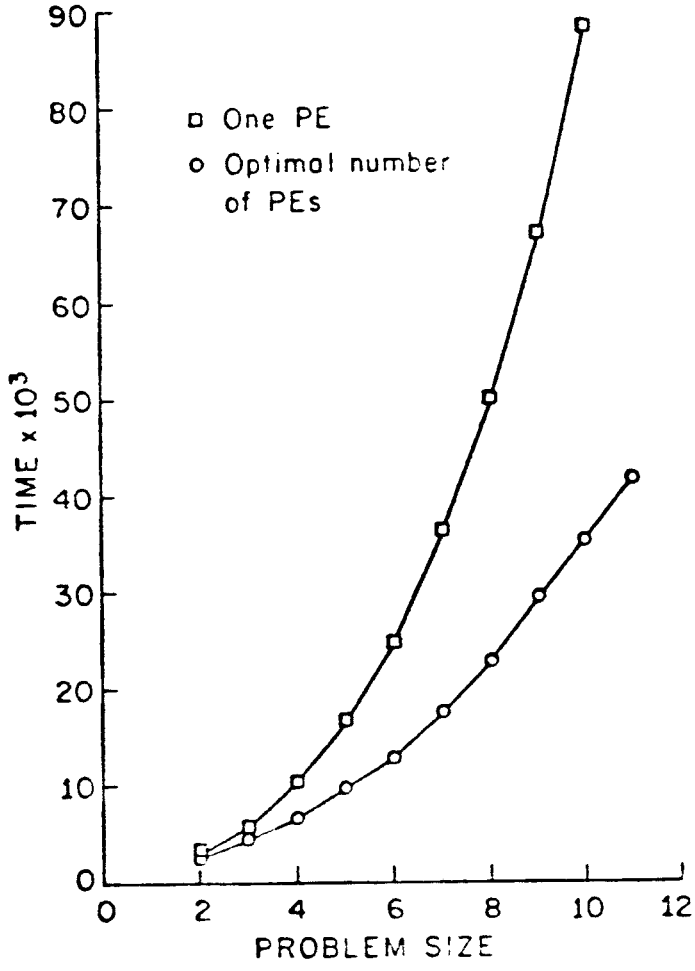


Figure 4.3b

Execution time complexity curves for Gaussian elimination: the curve for one processor is $O(n^3)$ and is similar to what would be achieved on a sequential machine, while the dataflow machine gives $O(n^2)$.

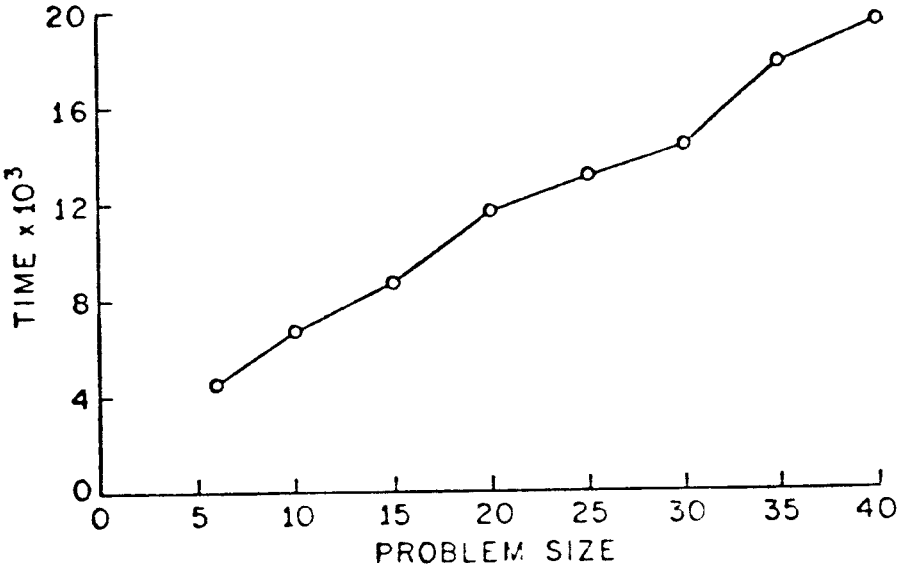


Figure 4.3c

Execution time complexity curve for recursive quicksort: the average behavior is O(n) on the dataflow machine with optimal number of PEs and O(n log n) on a sequential machine.

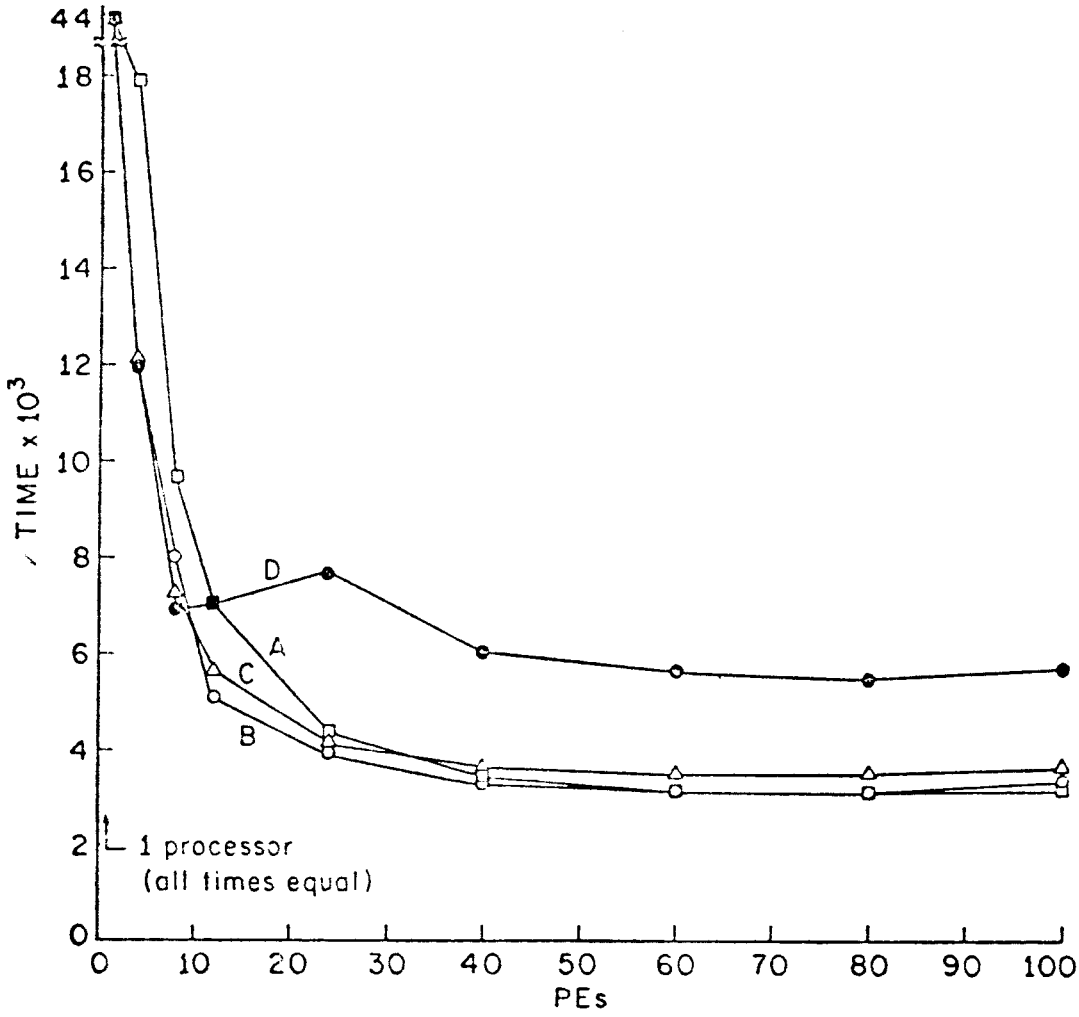


Figure 4.4

Speedup curves for 7×7 matrix multiply under four different assignment functions. Curves A-C encourage locality, while D does not.

Representation	Time			Space
	select	append	copy	(words)
vector	1	1 or n	n	n
2-3 tree	log n	2 log n	2n (leaves only)	8n

TABLE I

Structure representations
and their assigned costs

station	(a) standard configuration	(b) memory buses @.27 speed of standard configuration
sorter queue	.81	.53
sorter	4.00	4.00
ALU queue	3.34	2.20
ALU	8.44	8.44
code fetch*	3.15	16.67
data fetch*	3.40	25.72
output*	3.80	3.35
token bus	5.58	5.56
mean activity cycle time	32.51	66.47

*Includes box and associated queues

TABLE II

Activity flow analyses for
7×7 matrix multiply using 60 PEs

station	(a) standard configuration	(b) memory buses @0.27 speed
local bus queue*	.58	55.77
local bus*	3.20	12.00
local request processor queue	4.88	2.62
local request processor	6.83	6.61
global time**	2.14	8.00
memory response queue (in PE)	1.82	1.98
mean memory request cycle time	19.49	86.97

*Sum of request (from PE) and response (from memory controller) time

**Mean cycle time of all inter-memory controller messages proportioned by fraction of such messages out of all memory messages

TABLE III

Memory request flow analyses
for 7×7 matrix multiply using 60 PEs

	standard configuration			
	A	B	C	D
sorter queue	.90	.81	.68	.22
sorter	4.00	4.00	4.00	4.00
ALU queue	5.75	3.34	3.53	.88
ALU	8.44	8.44	8.44	8.44
code fetch*	1.23	3.15	5.36	2.54
data fetch*	2.86	3.40	3.51	2.45
output*	3.85	3.80	5.86	23.94
token bus	5.16	5.58	6.54	19.11
mean activity cycle time	32.20	32.51	37.93	61.57

*Includes box and associated queues

TABLE IV

Activity flow analyses for run using 60 PEs from each of curves A, B, C, D (Figure 4.4)

[1] Ackerman, W.B. and J.B. Dennis, "VAL--a value-oriented algorithmic language," Preliminary Reference Manual, Laboratory for Computer Science, MIT, Cambridge, MA, September 1978.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[3] Arvind, and K.P. Gostelow, "Some relationships between asynchronous interpreters of a dataflow language," Formal Description of Programming Languages, E.J. Neuhold, Ed., North-Holland, NY, 1977, pp. 849-853.

[4] Arvind, and K.P. Gostelow, "A computer capable of exchanging processing elements for time," Information Processing 77, B. Gilchrist, Ed., North-Holland, NY, 1977.

[5] Arvind, K.P. Gostelow, and W.E. Plouffe, "Indeterminacy, monitors, and dataflow," Proc. Sixth ACM Symp. on Operating Systems Principles, Nov. 1977, pp. 159-169.

[6] Arvind, K.P. Gostelow, and W.E. Plouffe, "An asynchronous programming language and computing machine," TR. 114A, Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, Sept. 1978.

[7] E.A. Ashcroft, and W.W. Wadge, "LUCID - a formal system for writing and proving programs," SIAM J. Comp., 5, 3 (Sept. 1976), pp. 336-354.

[8] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," CACM 21, 8 (Aug. 1978), pp. 613-641.

[9] A. Bährs, "Operation patterns: an extensible model of an extensible language," Proc. International Symposium on Theoretical Programming, Lecture Notes in Computer Science 5, Springer-Verlag, NY, 1974, pp. 217-246.

[10] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes, "The ILLIAC IV computer," IEEE Transactions on Computers, C-17, 8 (Aug. 1968), pp. 746-757.

[11] J.L. Bentley, "An introduction to algorithm design," Computer, February 1979, pp. 66-78.

[12] L. Bic, "Protection and security in a dataflow system," Tech. Report 126, (Ph.D. dissertation) Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, Oct. 1978.

[13] O. Cert, "Parallelism, control and synchronization expression in a single assignment language," SIGPLAN Notices, 13,1 (January 1978), pp. 25-33.

[14] A.L. Davis, "The architecture and system methodology of DDM1: a recursively structured data driven machine," Proc. Fifth Symposium on Computer Architecture, April 1978, pp. 210-215.

[15] J.B. Dennis, "First version of a data flow procedure language," Lecture Notes in Computer Science, 19, Springer-Verlag, NY, 1974, pp. 362-376.

[16] J.B. Dennis, and D. Misunas, "A preliminary architecture for a basic data flow processor," Proc. Third Symposium on Computer Architecture, Dec. 1974, pp. 126-132.

[17] S.H. Fuller, D.P. Siewiorek, and R.J. Swan, "Computer Modules: an architecture for large digital modules," AFIPS Conf. Proc., vol. 46 (1977), pp. 637-643.

[18] V.M. Glushokov, M.B. Ignatyev, V.A. Myasnikov, and V.A. Torgashev, "Recursive machines and computing technology," Information Processing 74, vol. 1, J.L. Rosenfeld, Ed., North-Holland, NY, 1974, pp. 65-70.

[19] K.P. Gostelow, and R.E. Thomas, "A view of dataflow," AFIPS Conf. Proc., vol. 48 (June 1979), pp. ?.

[20] R.M. Karp, and R.E. Miller, "Properties of a model for parallel computations: determinacy, termination, queuing, SIAM J. Appl. Math., 14, 6 (November 1966), pp. 1390-1411.

[21] R.M. Keller, "Parallel program schemata and maximal parallelism II: construction of closures," JACM 20, 4 (Oct. 1973), pp. 696-710.

[22] R.M. Keller, G. Lindstrom, and S. Patil, "An architecture for a loosely-coupled parallel processor", UUCS-78-105, Dept. of Computer Science, Univ. of Utah, Salt Lake City, Oct. 1978.

[23] P.R. Kosinski, "A data flow language for operating systems programming," ACM SIGPLAN Notices 8, 9 (Sept. 1973), pp. 89-94.

[24] P.J. Landin, "The next 700 programming languages," CACM, 9, 3 (March 1966), pp. 157-166.

[25] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I", CACM 3, 4 (Apr. 1960), pp. 184-195.

[26] S.S. Patil, "Closure properties of interconnections of determinate systems," Record of the Project MAC Conf. on Concurrent Systems and Parallel Computations, June 1970, pp. 107-116.

[27] J.E. Rodriguez, "A graph model for parallel computations," TR-64, Dept. of EE, Project MAC, MIT, Sept. 1969.

[28] J.E. Rumbaugh, "A dataflow multiprocessor," IEEE Transactions on Computers, C-26, 2 (Feb. 1977), pp. 138-146.

[29] I.E. Sutherland, and C.A. Mead, "Micro-electronics and computer science," Scientific American, 237, 3 (Sept. 1977), pp. 210-228.

[30] R.E. Thomas, "A comparison of methods for implementing dataflow structures," Dataflow Note 35, Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, May 1978.

[31] R.E. Thomas, "Performance analysis of two classes of dataflow computing systems," TR. 120 (M.S. Thesis), Dept. of Inf. and Comp. Science, Univ. of Ca., Irvine, CA, 1978.

[32] P.C. Treleaven, "Exploitation of parallelism in computer systems", Ph.D. dissertation, Dept. of Comp. Science, Univ. of Manchester, Manchester, England, Feb. 1977

[33] W.A. Wulf, and S.P. Harbison, "Reflections in a pool of processors - an experience report on C.mmp/Hydra," AFIPS Conf. Proc., vol. 47 (1978), pp. 939-951.