Dataflow Computing Models, Languages, and Machines for Intelligence Computations

JAYANTHA HERATH, MEMBER, IEEE, YOSHINORI YAMAGUCHI, NOBUO SAITO, MEMBER, IEEE, AND TOSHITSUGU YUBA

Abstract—**Dataflow computing, a radical departure from von Neumann computing, supports multiprocessing on a massive scale and plays a major role in permitting intelligence computing machines to achieve ultrahigh speeds. Intelligence computations consist of large complex numerical and nonnumerical computations. Efficient computing models are necessary to represent intelligence computations. An abstract computing model, a base language specification for the abstract model, high-level and low-level language design to map parallel algorithms to abstract computing model, parallel architecture design to support computing model and design of support software to map computing model to architecture are steps in constructing computing systems. This paper concentrates on dataflow computing for intelligence computations and presents a comparison of dataflow computing models, languages and dataflow computing machines for numerical and nonnumerical computations. The high level language—graph transformation that must be performed to achieve high performance for numerical and nonnumerical programs when executed in a dataflow computing environment are described using the DCBL transformations and applied to the Lisp language. Some general problems in dataflow computing machines are discussed. Performance evaluation measurements obtained by executing benchmark programs in the ETL's nonnumerical dataflow computing environment, the EM-3, are presented.**

Index Terms—**Architecture, dataflow, functional and logic programming, parallel computation, performance analysis.**

I. Introduction

INTELLIGENCE computations compute characteristics associated with human intelligence. They consist of large numerical and nonnumerical computations, including understanding, learning, reasoning, and problem solving. An ultrahigh speed computing system is necessary to compute such complex computations. The demand for ultrahigh speed computing machines for analyzing physical processes, solving scientific problems, and intelligence computations is increasing every day. The major difficulty in satisfying this demand in uniprocessing is the physical constraints of hardware and the sequential and centralized control in the von Neumann abstract computing model.

Sequential and deterministic von Neumann machines are not oriented to intelligence computations involving parallel and nondeterministic computations. The alternative to sequential processing is parallel processing with high density devices. To solve nondeterministic problems, it is necessary to research efficient computing models and more efficient heuristics. The machines processing intelligence computations must be dynamic. Efficient control mechanisms for load balancing of resources, communicating networks, garbage collectors, and schedulers are important in such machines. Computer hardware improved from vacuum tubes to VLSI but there has been no significant change in the sequential abstract computing model, sequential algorithms, languages and architecture. Circuit improvements that neglect the parallelism of problems do not lead to achieve higher computing speeds. Higher speeds in uniprocessor systems are achieved by using parallel control mechanisms such as interleaved memory, instruction fetch and execution overlap, extended instruction set, I/O processors, smaller and faster local storages and multiple execution units.

Parallelism in problems can be detected by users and compilers. New computing models are necessary to exploit the parallelism expressed by different algorithms which give different parallelism for the problem. New languages map the algorithms to computing models by expressing all the possible parallelism of an algorithm and defining the parallel tasks. New machines exploit instruction level parallelism. However, if there is no parallelism, no speedup can be expected. The dataflow graph partitioning for the vector machines is horizontal to generate the vectors and for the multiprocessors is vertical to generate one or more tasks, the basic unit for scheduling. Synchronization of control and data flow during execution assure the execution order.

Dataflow computing [1] provides multidimensional multiple pipelining instruction parallelism and hardware parallelism. Scheduling is based on availability of data. Processes are instruction size. The problems in multiprocessing due to the physical structure and operation are eliminated by parallelism and simple dataflow principle. The dataflow approach has the potential to exploit large scale concurrency efficiently, for maximum utilization of VLSI in computer design, compatibility with distributed networks, and compatibility with functional high-level programming. In dataflow, an instruction is enabled immediately after the arrival of required operands, and partial results of the execution are passed directly as data tokens. The computations are free of side effects, and in-

Manuscript received April 6, 1987; revised November 21, 1987. This work was supported by the Ministry of Education and the Ministry of Internal Trade and Industry, Japan, and the Ministry of Higher Education, Sri Lanka.

J. Herath is with the Department of Computer Science, George Mason University, Fairfax, VA 22030.

Y. Yamaguchi and T. Yuba are with the Electrotechnical Laboratory, Niiharigun, Ibaraki 305, Japan.

N. Saito is with the Department of Mathematics, Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223, Japan.

IEEE Log Number 8824635.

0098-5589/88/1200-1805$01.00 © 1988 IEEE

dependent computations proceed in parallel. In dataflow computing, there is no concept of shared data storage. Every instruction is allocated by the computing element.

The dataflow concept can be easily implemented in both major computing application areas; numerical and nonnumerical computations. Basic elements of dataflow computing are operators, arcs and tokens. An operation is enabled as soon as its operands are available. The output token value is determined by the operation and input token values. All dataflow graphs shown in this paper are drawn according to the following convention. Boxes represent oprations; arrows represent arcs; arrows with black heads represent the path for data tokens; arrows with white heads represent the path for control tokens; black dots represent the data tokens; and the white dots represent the control tokens. Common evaluation techniques of these models include strict and nonstrict evaluations. In strict dataflow computing, all the operands of an operation or arguments of a function must be presented to enable the execution. Figs. 1(a) and 1(b) show the firing sequence of the strict computation, multiply. In nonstrict dataflow computing, a selected number of operands of an operation or arguments of a function is sufficient to enable the execution. This avoids unnecessary computations, eliminates nonterminating computations and optimizes the computations. Figs. 1(c) and 1(d) show the firing sequence of the nonstrict computation, HCONS. The HCONS operator has two arguments. The result of HCONS, the first argument, is generated immediately after the arrival of first argument. Figs. 1(e) and 1(f) show the dataflow computing for the numerical computation $(A * B) - (B/C) + (C/D)$ and the nonnumerical computation CONS(CAR($x1$, $x2$, $\cdots$), CDR($x1$, $x2$, $\cdots$)).

In dataflow computing, data structures in the storage are represented by a pointer token. This reduces the parallelism of the computation, but provides safe execution. In static dataflow, arrays are treated either as a set of scalars which allow the elements of the array to be handled simultaneously by independent dataflow instructions or as a sequence of values which spreads the array out in time for pipeline execution. Heaps are functional directed acyclic graphs. They must be completely produced before consumption. The append, select, create and delete actors are used to access these structures. The I-structures allow a selection of elements before complete production of the structure. The position of an element is defined by a tagged token. The presence, absence and wait bits indicate the state of the element. Read of an unwritten storage cell is deferred by the controller until a write arrives. Pipelining between consumers and producers gives better performance. Streams are sequentially allocated arrays.

In the basic dataflow computing model, the number of tokens per arc is restricted to one during the entire computation which results in huge acyclic graphs. This makes the computation strictly iterative. This problem is solved by many other advanced computing models. These models support the building of highly parallel and asynchronous

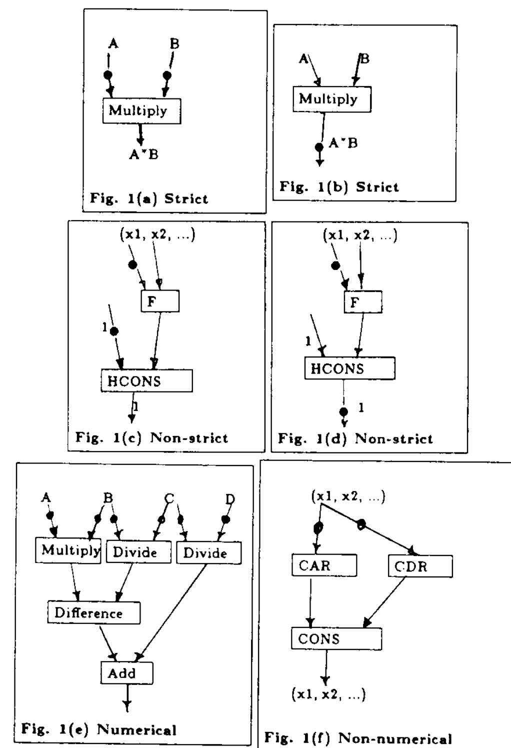


Fig. 1. Dataflow computing.

computing machines, but differ in their approach as to how the computing should proceed. Section II of this paper discusses the acknowledgment static, strictly static, recursive dynamic, tagged token dynamic, eduction, lazy-eager, pseudo-result, and Not(operation) dataflow computing models. The discussion is based on the representation of conditional computations, the root of iterative and recursive computations. Section III overviews the logic programming and functional languages used to represent dataflow computations and the process of high-level languages to graph transformation using DCBL transformation. This transformation is applied to Lisp to obtain dataflow graphs. Section IV discusses the characteristics of representative dataflow computing machines for numerical and nonnumerical computations and some general problems. Section V gives the performance evaluation measurements made using EM-3.

II. Dataflow Computing Models

A. Static Computing

Static dataflow computing [1] to [5] was proposed by Dennis for ultrahigh speed computing machines. The VIM, Texas DDP, LAU, Hughes, and NEDIPS systems are based on the static computing model. This computing model consists of operators, data and control arcs, and data and control tokens. In static computing, concurrent reentrance is inhibited. Several tokens per link are allowed but there is a restriction of one token per time. An

actor fires when there are no tokens on any of the actor's output arcs.

Two models are used to represent static dataflow computing. The strictly static model used in Texas DDP [6], [7] prohibits initiation of a new iteration before the previous one is concluded. The branch node does not provide new tokens until the previous iteration is completed. This model provides safe execution of the computation but limits the parallelism. In the acknowledgment static model [3] consumers send acknowledgment signals to producers, indicating the possibility of accepting a new set of tokens. This enables pipeline production of tokens and exploits the parallelism by allowing initiation of new iterations before the previous one has been concluded. Safe execution of reentrant graphs is provided with added complexity.

A token consists of a value and one component tag representing the target actor identity. No code copying or recursion is allowed. Only iteration is supported. The switch-t, switch-f, and merge operations are introduced to support conditional computations in both models. A true token at the input of switch-t copies the other token in the input to the output. A false token at the input of switch-t does not dispatch the other input token to the output. Similarly, the switch-f operator dispatches the input token to the output if and only if the boolean input token is false. Three input merge operators are executed when the boolean input and appropriate data token is available. Fig. 2 shows the implementation of the conditional computation, IF C(x) THEN A1(x) ELSE B1(x), and three instances of the firing sequence.

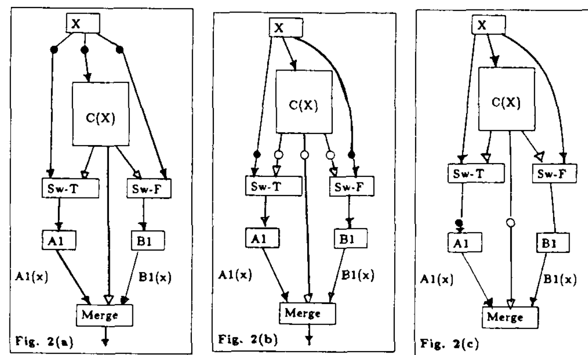


Fig. 2. Static computing.

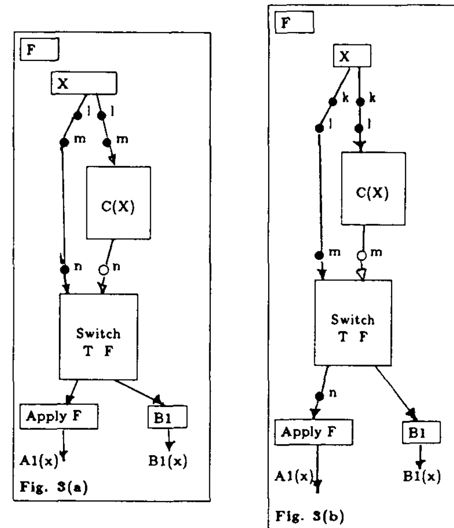


Fig. 3. Recursive dynamic computing.

B. Recursive Dynamic Computing

In dynamic dataflow computing, several instances of a node can be fired at a time and these nodes can be created at run time. Concurrent reentrance is permitted using code copying. In code copying, a new instance of a subgraph is created. The tokens must be directed to the corresponding instance. This enables recursive computations in the dataflow computing environment. Davis [16], [17] and Dennis *et al.* [4] proposed the recursive dynamic computing model based on FIFO queues and recursive computations resulting in acyclic directed graphs. In each invocation, a maximum of one token is placed on a link. An apply actor causes a new copy of the program graph. There are no merge actors because in any instance of the graph only one of the data inputs of the merge is used. A token consists of a value and a two component tag, one representing the graph instance, the other representing the actor within the graph. A token is represented by, $< v, < u, s > d >$, a data value v, an activation instance u, an actor within the function s, and the operand of the target actor d. The operand of the target actor is not necessary for single operand operations. The DDM 1 machine is based on this recursive dynamic computing. Fig. 3 shows the firing sequence for recursive dynamic dataflow computing.

C. Tagged Token Dynamic Computing

The tagged token dynamic computing model, proposed separately by Arvind [8] to [11] and Gurd-Watson [12]–[15], is more efficient in exploiting the parallelism to a large degree. A tag assigned to each token distinguishes its identity. Identically tagged tokens enable the execution of an operation. Tagging allows many data values per link at one time. Several instances of a node are fired at one time. Each node can be created at run time. Recursion and iteration are represented directly. Successive cycles of an iteration are allowed to overlap by unfolding loops.

A token consists of a value and tag representing the target actor identity. A token is represented by, $< v, < u, c, s, i > d >$, a data value v, an activation instance u, a code block (loop body) c, an actor within the function s, an index representing the cycle of an iteration (data structures) i, and function activation d. No merge actor is used. Identity actors, such as D-operator for loop entry which establishes a new context for iteration and sets the index of result tokens to one greater than the index of the input token and D-reset operator for loop exit which restores the tag of result token to that of the context surrounding

the tag, are used. Special mechanisms, such as loop throttling are used to limit the parallelism exploited by tagged tokens.

The MIT TTDA, Manchester Dataflow machine and all tagged token dynamic dataflow machines are based on this model. Id, LAPSE, MAD, SASL, SISAL, and many other dataflow languages support these machines. The switch operation used to implement conditional computations has two input arcs, one for boolean tokens and the other for data tokens to be switched. This operator also has two output arcs. The incoming boolean token determines the output arc along which the incoming data token is sent. A true token copies the other token in the input to the *T* output while a false token copies to the *F* output. No merge operators are used. The BRR operation used by Gurd's group is similar to this switch operation. Fig. 4 shows the firing sequence for tagged token dynamic dataflow computing.

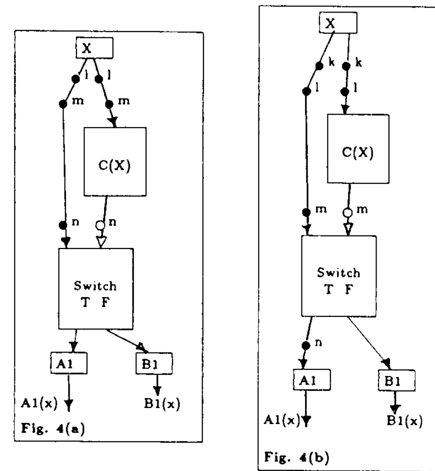


Fig. 4. Dynamic computing.

D. Eduction Computing

The Eduction model, proposed by Ashcroft and Jagannathan [18], is a hybrid computing model of dataflow and demand flow computing. Operator nets represent the eduction computations graphically. The demand for a result triggers its computation which in turn triggers evaluation of its arguments. The demand propagation continues until constants are encountered, then a value is returned to the demanding node and execution proceeds in the opposite direction. This minimizes the computation to compute only necessary computations for a particular problem. The arguments for branches of conditional computations are not evaluated in parallel. Only necessary arguments are evaluated. The modal operators, where, first, next, followed by, as soon as, merge, whenever, upon and is current, are used to express recursion and iteration in a purely functional way. The Eazy flow engine is proposed to execute operator nets described in LUCID language. Fig. 5 shows the firing sequence of eduction computing. The wvr node is similar to the switch-t operation. The switch operation is occasionally used. Operators such as wvr and merge need extra memory to remember the last token arrived in dynamic dataflow computing.

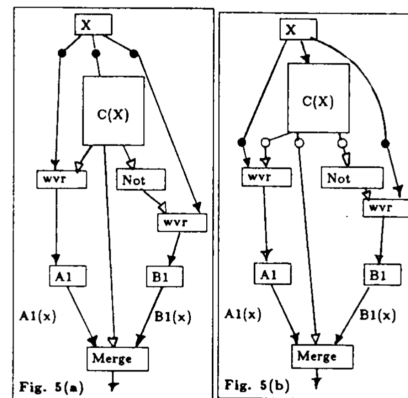


Fig. 5. Eduction computing.

E. Dataflow—Control flow Computing

The Dataflow-Control flow computing model proposed by Treleavan *et al.* [20] use two basic mechanisms. One instruction causes the execution of others using the control mechanism. Instructions receive and dispatch data using the data mechanism. Instruction execution is caused by the availability of specific set of data and control tokens. Data tokens carry partial result values while control tokens carry null values. Instructions are activated by the set of control tokens. Conditional computations are supported by the many input two output switch operation.

F. Eager-Lazy Computing

The eager-lazy dataflow computing model was proposed by Amamiya *et al.* [21] to [26] for artificial intelligence applications. In eager evaluation, all possible computations are executed in parallel without optimizing. Conditional computations are executed parallel to the branches. CAR and CDR parts are evaluated in parallel to the CONS. CONS(x, y) is implemented using the getcell, writecar and writecdr operations shown in Fig. 6. This is the lenient cons mechanism. In lazy evaluation, selected computations are executed to optimize the computation. The selected branch is executed after the execution of the conditional computation. In the lazy cons mechanism, the car or cdr part is evaluated only when its value is demanded. In this model, eager, lazy, nonstrict, and demand driven computing mechanisms are selectively and efficiently implemented to obtain the maximum efficiency. The model is implemented in DFM using VALID language.

G. Pseudoresult Computing

Yamaguchi *et al.* [28]–[30] proposed the pseudoresult dataflow computing model shown in Fig. 7, particularly for AI applications. In Fig. 7 black boxes represent pseu-

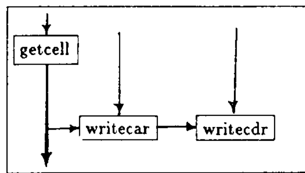
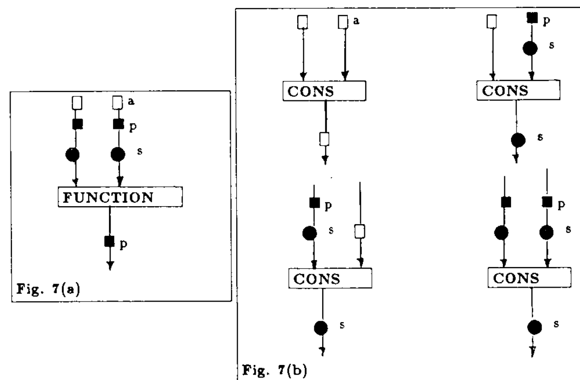


Fig. 6. Eager-lazy computing.

doresults, black dots represent semiresults and white boxes represent actual results. Pseudoresults are generated immediately after the arrival of all arguments, as a result of function execution. See Fig. 7(a). This pseudoresult enables the successive computations relaxing the firing conditions. The operations in the function are executed concurrently with the evaluation of its successor. The identifiers of pseudoresults are realized by addresses in a result store and are eventually filled by actual results. The semiresult is the pseudoresult used in nonnumerical computations, and the partial-result is the pseudoresult used in numerical computations. When the input to an operation or function is actual, semi or pseudo, the output is an actual or semiresult. The execution of an arithmetic operation is deferred until the inputs become actual. Fig. 7(b) shows four different instances of CONS execution and Fig. 7(c) shows an application example of pseudoresults. This model is implemented in the EM-3 using the languages EMLISP and EMIL.



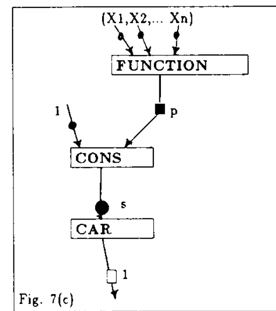


Fig. 7. Pseudoresult based computing.

H. Not(operation) Computing

In the Not(operation) computing model [31]–[33], parallel computations are represented by sequential, parallel and decision making computation segments. Ordered sequential computation segments ensure the logical correctness of the computation. Parallel computation segments composed of independent computations. The conditional computation is represented using two parallel complementary computations. The transformation of a traditional conditional computation to a Not(operation) based computation is performed in two steps. First, the traditional conditional computation is disintegrated into two complementary basic operations which must be executed for deadlock-free computation. The positive state is denoted by Operation and the negative state is denoted by Not(operation). The negative state represents many other positive and negative states. n sequential conditional computations are represented by n different independent parallel operations. Then the semantics of the execution are defined. One of the operations executed will give an output value if the operation is satisfied. Figs. 8(a) and 8(b) show the firing sequence of the conditional computation. IF (Operation) THEN S1 OR IF (Not(operation)) THEN S2. The Operation and Not(operation) receive a copy of the input token. When the Operation satisfies the input data token, the data token is given as the result of execution, and the output of the Not(operation) is frozen. Otherwise, the Not(operation) gives the data value output while the Operation output is frozen. Figs. 8(c) and 8(d) show two instances of firing sequence of n parallel conditional computations.

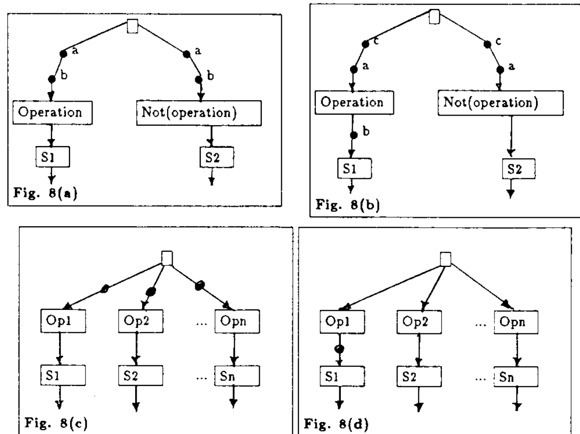


Fig. 8. Not(operation) computing.

III. Dataflow Computing Languages

The language is very important in representing parallel algorithms for intelligence computations and mapping them efficiently onto the computing environment. Functional and logic programming languages are two major declarative language paradigms to enhance intelligence computing productivity. In dataflow computing it is possible to use an existing sequential language, functional language, parallel logic programming or any other high-level language. The use of existing languages allows ex-

isting software to run on the new machine and give a programmer high degree of control over the run time behavior. Conventional programs consisting of sequences of statements and control statements alter the data stored in memory one at a time. Variables are used to represent storage cells and a statement is necessary to alter data for each variable.

The use of sequential languages to represent parallel algorithms and map for a parallel execution environment complicate the execution process. Algorithms that form dataflow graphs from conventional languages are complex. The concurrency detected by a compiler is also limited. The use of a language which reflects the parallel machine features exploits the machine parallelism but increases the programming complexity. Dataflow programming requires no knowledge of machine structure and there is no need of explicit expression of parallelism. The compiler detects the parallelism. Users should not consider the explicit control of memory allocations in using machines, but should deal only with data values. Low-level languages for dataflow computing machines should describe dataflow computing efficiently.

A. Logic Programming

Robinson's resolution principle [34] applies only one powerful rule of inference to mechanical theorem proving. This enables computer making deductions from set of logical formulas. Drawing inferences at a very high speed is the future objective of expert systems. Logic programming, based on symbolic logic, is suitable for knowledge processing systems dealing with large databases [35]. Implicit search strategy and parallelism support symbolic processing. Logic programming describes the facts and their relationships in a problem and controls the execution nondeterministically. Questioning gives the answer using declared facts and defined rules. A question is answerable if it is the head of any other clause and each of its goals is true. When answering a question, logic programming looks for matching facts in the database. Two facts match if their predicates and corresponding arguments are the same. The process of matching, unification, is the execution mode. Clauses in logic programming are transformed into dataflow graphs.

1) Prolog: Colmeraur's Prolog design [36] based on language theory and mathematical logic with practical constraints. Prolog, a sequential logic programming language, draws inferences efficiently. Relationships are represented as predicates, and objects are represented as arguments. Facts declare the relationships between objects. Assertion, a fact, has no body. Conditional assertion, a rule, has a head and body. Rules are used to describe or define the relationships. The execution mode is unification with backtracking. Prolog languages start the execution of a goal only after the completion of the previous goal.

2) Relational Language: Clark's Relational language [38] is focus on parallel execution of logic programs. Relational language features include AND-parallel execution of conjunctive goals, process communications by shared variables and OR-parallel reduction. The commit operator is introduced to separate the guard and body. AND-parallel processes are synchronized by defining the instances of the variables as producers or consumers.

3) PARLOG: Clark's PARLOG [39] augments the expressive power of Relational language. In PARLOG, the resolution tree has one chain at AND levels, and OR levels are partially or fully generated. Communicating processes combine the partial solutions. Restriction of the access mode is specified by mode declaration. The modes of predicate variables are predefined as input or output.

4) Concurrent Prolog: Many features of Relational Language are implemented in Shapiro's Concurrent Prolog [40]. In Concurrent Prolog the search strategy is multiple, depth first. The resolution tree consists of one chain from top to bottom. Guards can bind variables. Read only variables in a process are introduced to support process synchronization. The clause activation is suspended until the variable is assigned a value.

B. Functional Programming Languages

In functional programming languages, programs are mathematical functions based on functional algebra. Function application is the major operation. The object is mapped onto another object. There is no concept of storage, assignment, goto, or side effects. Programs are free building blocks for larger programs. Functional languages do not reflect von Neumann properties or the machine structure and are zero or single assignment languages. They provide specially controlled reassignment constructs for loop. Functional languages such as Pure Lisp and FP [41], [42] can be used effectively to execute computations in dataflow computing machines. In FP, programs are used to construct new programs using program forming operations. It increases the expressiveness of algorithms, exploits the massive parallelism in scientific computations, permits abstract data structures, streams, and irregular data structures, and allows powerful programming constructs. However, some problems, including storage control, need efficient solution.

1) VAL: VAL, the high level language designed by Dennis's group [1]–[5], is value oriented, as opposed to traditional variable orientation. In a value oriented system, new values are defined and used but no values can ever be modified. Values may be bound to identifiers but identifiers are prevented from being used as variables. The design principles of VAL provide implicit concurrency and synchronization by using completely functional language features. Expression based features inhibit all forms of side effects. Once the values of all inputs are known, execution cannot influence the results of any other operation ready to be executed. Automatic detection of parallel computations by compilers, vectorization, has been used to exploit concurrency. Side effect features, memory update and aliasing are banned. VAL helps simplify critical programming chores such as error handling, debug-

ging, and speed analysis. VIMVAL, an extension of VAL, treat functions as first class objects. They are passed as arguments and returned as results of functions. Stream types, free variables, recursion, and mutual recursion features are added.

2) Id: Id [8]–[11] was proposed by Arvind and Gostelow. Id, or Irvine Dataflow, is a block structured, expression oriented, side effect free, single assignment language. A program in Id is a list of expressions. The four basic expressions are blocks, conditionals, loops and procedure applications. Id variables are not typed. SELECT and APPEND create new and logically distinct structures. Executions are dynamic compared to the static nature of Dennis's model. Id supports streams, non-deterministic programming and higher order functions.

3) LUCID: LUCID [19], proposed by Ashcroft and Wadge, is the programming language of operator nets. The transformation from one to another is simple since the programs represent mathematical semantics of operator nets. In LUCID, programming proofs are carried out and incorporates iterations by regarding all values as histories [17]. Everything, including constants, is an infinite history. Assignment statements are equations between histories. A program in LUCID is an unordered set of equations. Conventional LUCID is implemented employing demand driven computing for infinite objects.

4) Manchester Languages: Languages used in the Manchester machine [12]–[15] are conventional languages, LAPSE, MAD, SASL, and SISAL. SASL, based on LUCID, treats functions as first class objects. In particular, the function takes one argument, and currying is used to obtain the effect of multiple argument functions. LAPSE, a single assignment language, has Pascal-like syntax. LAPSE stores arrays during iteration or for-all loops which have used them. MAD, based on Id, is typed, using streams and has operators such as list processing operations. MAD stores arrays longer, and garbage collection is performed using reference counts. SPNLN and TASS are low-level dataflow graph languages used for these languages.

SISAL [43], [44], stream and iteration in a single assignment language, is a cooperative research by the Colarado State University, DEC, Lawrence Livermore Laboratory, and Manchester University. This language is a value oriented functional language for sequential, vector, multiprocessor, and dataflow computing machines. SISAL is implemented on the VAX, CRAY, HEP, and Manchester dataflow machines. SISAL is strongly typed. Recursion has been added. Error values are simplified. Some Id (/MAD) features are added. Tokens are labeled to allow multiple use of arcs. Labels are used for data structures, loops, and functions. IF1, the intermediate language for SISAL [36], performs machine independent optimizations and machine dependent analysis.

5) Valid: Valid, value identification language [23] designed by Amamiya *et al.*, is a functional language with implicit and explicit parallel constructs. Lenient-cons computing is applied in function evaluation to achieve parallelism. List computations and higher order functions are written using an Algol and Lisp-like syntax.

6) EMLISP: EMLISP, is a single assignment language [27]–[30]. To obtain side effect free, pure functional list processing, the features added to conventional Lisp to increase efficiency in von Neumann computing, such as relatives of PROG, flow controlling operations, list modifiers, relatives of array, and side effect operations such as RPLACA and RPLACD, are removed. The global and free variables and loops are inhibited. Special features such as parallel COND, parallel OR, parallel AND, and BLOCK are added. EMIL is the low-level language used to represent dataflow computing graphs.

C. DCBL Transformations for Dataflow Computing Languages

The objectives of DCBL (pronounced decibel) [32] design is to define operational semantics for dataflow computing languages, development of high level languages based on abstraction mechanisms that frees the user from consideration of machine characteristics and are comfortable for users to express many forms of concurrency, to facilitate the natural expression of parallelism of the problems for processing on dataflow machines, and to enable a compiler to generate optimized code that exploits the inherent dataflow parallelism without the application of sophisticated analysis techniques.

1) Specification of DCBL: DCBL allows parallel algorithms to be expressed as a collection of expressions. The execution of a DCBL program consists of a sequence of parallel executions of expressions. An expression execution may generate zero, one or more than two values. Tuple expressions, multivalue function expressions, conditional expressions, and parallel expressions generate more than two values. The DCBL syntax specification for iterative computations is given below.

```
exp ::= function(exp)
    | exp, exp, · · · exp
    | IF exp THEN exp, IFNOT exp THEN exp
    | identifiers
    | constants
    | LET idlist = exp IN exp
    | IF exp THEN exp
    | FOR idlist = exp DO iteration
iteration ::= ITER exp NOTITER exp
    | LET idlist = exp IN iteration
    | IF exp THEN iteration
    | IF exp THEN iteration, IFNOT exp THEN iteration
idlist ::= id
    | idlist id
```

The application of a function to an expression, **funct(exp)**, is used to represent sequential computations. The elementary functions are operators. The operations performed on expressions can be characterized by mathematical functions. The application of function F to the

imports, x, y, and z, produces export $F(x,y,z)$. The expression **exp, exp,** $\cdots$ **exp** is used to represent parallel computations. Identifiers and constants are the most elementary expressions. Values can be bound to identifiers, which can be bound to simple types (integer and real), structured types and function calls. The **LET IN** expression provides local binding to extend the execution environment.

Decision making computations, conditional expressions and the **FOR DO** expressions, sequence the parallel computation to ensure logical correctness and avoid initiating computations whose results can never be used. Conditional expression represents operational semantics for conditional, iterative and recursive computations. The general IF THEN ELSE expressions and case expressions are represented by the **IF exp THEN exp**. All predicates in **IF exp THEN** are supported by expressions. **IF exp THEN exp** provides the single branch conditional expression. **IF exp1 THEN exp2 IFNOT exp1 THEN exp3** is a two branch conditional expression with two parallel complementary conditional computations. Combining n **IF exp THEN exp** expressions, gives n parallel computations with a live branch and $n - 1$ dead branches.

The **FOR idlist = exp DO** iteration expression implements iterative computations that depend on the previous iterative computation result. The **FOR** expression has loop initiation and a loop body. Loop initiation is performed by the **FOR idlist = exp** part and the loop body appears in **DO iteration**. The iterative expression is evaluated by binding the iterative identifiers, the elements of idlist, to the values of **exp**. The evaluation of the iteration body results in a **NOTITER** expression and an **ITER** expression. Both these expressions are evaluated concurrently in each iteration exploiting the hidden parallelism of the iteration expression. If the **NOTITER** expression satisfies the condition, this terminates the iteration and gives the computation result. Otherwise, the output is given by **ITER** expression. Here, the **ITER** expression is satisfied and continues iteration. The iteration is terminated when the evaluation of the **ITER** body results in an ordinary **NOTITER** expression. The value of this expression is the value of the **ITER** expression. Parallel expressions for the computations of the type **For i := 1 to n do C[i] := A[i] * B[i]**, represents iterations that do not depend on the previous computation result.

2) Dataflow Graph Specification Language: A dataflow graph representing dataflow computations can be defined by $N = [T, O, L]$ where T, O, and L represent the set of tokens, the set of operations and the set of links. For an element Oi in O, the set $\mathrm{Im}(Oi)$ represents the import ports of Oi and $\mathrm{Ex}(Oi)$ represents the export ports of Oi. Firing of an operation maps imports to exports. The semantics of firing define the minimum set of import ports, varying from one to the total number of imports, that must receive imports to enable an operation/function. The output semantics may vary depending on the execution of an operation. Firing an operator dispatches exports to zero or more export links. Expressions and the compiler help identify concurrency in algorithms and their program and map that concurrency onto graphs. The graph, which connects subgraphs composed of operators, is an explicit representation of the concurrency available in evaluating expressions.

An element of a dataflow computation consists of import ports, imports, export ports, exports, import links, export links, and operators. Specifications of a dataflow graph include imports, exports, data links, and operators. Operators are defined recursively using local imports and exports. Imports to the operator embark at import ports. Exports of the operators disembark at export ports. The number of imports or exports in a link is unlimited. This gives the dynamic computing features. The restriction of values to one gives the static computing feature. The operators communicate values through their import and export ports. The graph has an import port for each free variable of the expression and an export port for each value returned by the expression.

The exports produced are exported via export ports to defined destinations to enable successive computations. The destination of an export value is specified by the import port number of the destination operator. The export port of an operator is connected by a link to the import port of another operator. The export value of one operator is the import value to another operator.

The following notations are used to specify the dataflow graphs. **T(exp)** is used to represent the operators of the translated expression exp. **IM.T(exp)** represent the set of imports to T(exp). **EX.T(exp)** represents the exports at the export ports of the T(exp). The imports and exports have defined import ports and export ports. Links are represented by **EX.T(exp1)** $\rightarrow$ **IM.T(exp2)** which means that the exports of T(exp1) are linked as the imports to the defined import ports of the T(exp2). The import ports of all parallel subgraphs are assigned the set of import values. The graph export ports are formed by concatenating the export ports of the component subgraphs. This graph language provides facilities to design demand driven computing languages and to specify parallel and distributed systems.

The complexity of a dataflow graph increases with the number of operators and arcs. This increases execution and communication time and creates many problems when executing in a limited resource. The fundamental principle of managing the complexity is to reduce the size of the graph while preserving the original properties of the graph. In graph reduction, the number of operators, arcs and tokens generated are reduced without changing the final result of the computation.

3) DCBL Transformation: The transfer function T maps expressions to dataflow graphs and the functionality of the operator F maps imports onto exports. The operational semantics are defined and derived by the applica-

tion of F(T(exp)). The expression transformation to graphs gives the informal operational semantics of the dataflow graphs. The transformation of funct(exp), T(funct(exp)), is sequentially connected dataflow subgraphs. The transformation is made by connecting the export ports of T(exp) to the import ports of T(funct). The transformation of [T(exp1, exp2, · · · expn)] consists of n subgraphs, [T(exp1)], [T(exp2)], · · · and [T(expn)], that can be executed in parallel. Two subgraphs are connected sequentially in the implementation of the simplest conditional expression, **IF exp1 THEN exp2**. Predicate exp1 controls the evaluation of exp2. The import data value of T(exp1) is the export of T(exp1) if this data satisfies the condition expressed by **exp1**; if not, the data value is simply absorbed. This expression provides the facility to evaluate n parallel conditional expressions. The transformation of parallel complementary conditional expressions, **IF exp1 THEN exp2 IFNOT exp1 THEN exp3** is illustrated in Fig. 9. The transformation of the identifier, [T(id)], gives a graph with no operators. The transformation of a constant expression gives the const operator with import export links. A trigger-value import produces the value, const, as the export.

DCBL binds identifiers locally. In evaluating **FOR idlist = exp DO iteration**, the elements of idlist are bound to the values of exp, and iteration is terminated when the iteration results in an ordinary expression. **ITER(exp)** supports iteration if the imports satisfy the expression exp. **NOTITERexp** gives the result of the computation. The iteration body, **LET idlist = exp IN iteration**, is implemented in the same way as the expression **LET idlist = exp1 IN exp2** is implemented. The dataflow graph implementation of the conditional iteration body, **IF exp THEN** iteration, is similar to that of the conditional expression. Both subgraphs, IF exp and IFNOTexp, provide a complete set of exports. [T(exp)] and [T(notexp)] are placed on the import paths of the iteration body subgraphs, [T1(iteration1)] and [T1(iteration2)]. Exports of [T(exp)] or [T(notexp)] enable the evaluation of a selected iteration body.

4) Functionality: The functionality of dataflow graph represents the operational semantics of expressions and the formal simulation of dataflow graph execution. The graph is mapped onto its semantic characteristics using the functionality of its operators. The operational semantics of a dataflow operator are given by its functionality which maps its imports onto exports. The functionality of an operator is the usual arithmetic or boolean function associated with it. For example, $F+(x, y) = x + y$ and $F\text{constant}(x) = \text{constant}$. Arrival of x token triggers the constant operator to give the defined export. The functionality can be extended for an ordered set of dataflow imports. The operator, plus, can be applied to the ordered sets $x.X$ and $y.Y$ where x represents the first value of one ordered set, X represents the rest of that ordered set, y represents the first value of the other ordered set, and Y

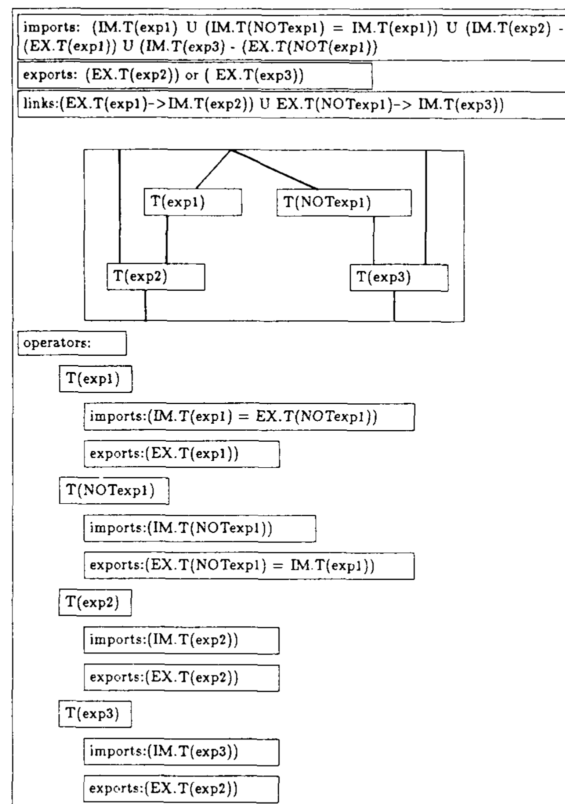


Fig. 9. DCBL transformation—T(IF exp1 THEN exp2 IFNOT exp1 THEN exp3).

represents the rest of that ordered set. Hence, $F\text{plus}(x.X, y.Y) = F\text{plus}(x, y)$. $F\text{plus}(X, Y) = (x + y)$. $F\text{plus}(X, Y)$.

The characteristics of operators or functions can be distinguished by either imports or exports. According to imports there are two types, strict and nonstrict, of operators or functions. In strict operator or function the availability of all imports enables the execution. For strict operators, the operator will not execute without its complete set of imports. **F(X, Y, · · ·) = e if X or Y · · · = e = empty** Nonstrict operators or functions need the availability of specified operands or arguments to enable the execution. According to exports there are two types of operators; one produces exports in the execution with the arrival of imports, and the other, used in the implementation of conditional, iterative, and recursive computations, produces and seizes or freezes the exports, depending on the arrival of imports. In executing conditional operators, the data value imported is exported if it satisfies the conditional operator; if it does not satisfy the conditional operator, no export values are produced. The complementary set of conditional operators, $F\text{cond}(x)$ and $F\text{notcond}(x)$, can be executed concurrently.

5) DCBL Transformations in Lisp: The DCBL transformation process can be used in any language to repre-

sent dataflow computations. Flow graph languages can be defined, using imports or exports. Import based languages can be used to represent demand driven computations. Imports and exports show the relationship between dataflow and demand flow computations. IL is the intermediate form for transformed Lisp programs. IL-1 represents the dataflow computations based on exports from an operator. The format of the codes is

(OPCODE CONSTANT DEST-LIST)
(CALL FUNCTION-NAME NO-OF-ARG NO-OF-RET DEST-LIST)
(PROC FUNCTION-NAME NO-OF-ARG DEST-LIST).

The opcode represents an operation or a function name. Constant type operands of an operation are placed in the constant datum field of the operation. Dest-list, corresponding to the number of output arcs of a node, represents the destinations of the result of an operation. A destination field consists of a label field and an attribute field. The label field represents the destination node and the attribute field represents the node attribute. Proc specifies the function name, total number of arguments, and destination list of each argument. Fig. 10(a) shows the Lisp program and Fig. 10(b) shows the IL-1 code for Fibonacci computation,

$$F(1) = 1;$$
$$F(2) = 1;$$
$$F(n) = F(n-1) + F(n-2).$$

Fig. 10(c) shows the IL code based on imports to an operator. Instead of destinations of exports being defined, the origins of imports are defined. DEST-LIST in IL code is replaced by ORG-LIST, which represents the origins of the imports, to get the IL format for import based computations.

The nonnumerical and numerical operations in IL are car, cdr, cons, add, multiply, subtract and divide. The dataflow computing support codes are distribution, procedure, call, return and constant. The first column of Table I gives the definitions of these operators. The second column of Table I gives the definitions of nonstrict operators used in IL. The third column of Table I gives the definitions of the conditional operations. CONSTANT is used to obtain constants, TRUE, FALSE, or any other required value. IL codes give the Fig. 10(d) dataflow graph of the Fibonacci computation. Parallel EQUAL and NOTEQUAL satisfy the conditional computation requirement. The functionality of IL operators for nonnumerical, numerical, and conditional operations is shown below. Here, N and E imply frozen export and error value export.

- *List operations:*
  Fhead((x1 x2 · · ·)) = x1   Ffhead(( )) = E
  Ftail((x1 x2 · · ·)) = (x2 x3 · · ·) Ftail(()) = E
  Fcons(x1 (x2 · · ·)) = (x1 x2 · · ·)

```
Fib n = 1; if n=1 or 2
= Fib n-1 + Fib n-2

(defun fibonacci (n)
(cond (( eq n 1) 1)
(( eq n 2) 1)
(t (plus (fibonacci(difference n 1))
(fibonacci(difference n 2))))))
```
Fig. 10(a)

```
G0001 (PROCEDURE FIB 1. (G0002 MONO-0) )
G0002 (*DISTRIBUTE (G0001 MONO-0) (G0004 MONO-0) )
G0003 (*EQ (C-1 1.) (G0012 (RETURN 1.)) )
G0004 (*EQ (C-1 2.) (G0006 MONO-0)) )
G0005 (*GT (C-1 2.) (G0007 MONO-0) (G0008 MONO-0) )
G0006 (*CONSTANT (C-1 1.) (G0012(RETURN 1.)) )
G0007 (*DIFFERENCE (C-1 1.) (G0009 (ARG 1. 1.)) )
G0008 (*DIFFERENCE (C-1 2.) (G0010 (ARG 1. 1.)) )
G0009 (*CALL FIB 1.1. (G0011 0.) )
G0010 (*CALL FIB 1.1. (G0011 1.) )
G0011 (*PLUS (G0014 (RETURN 1.)) )
G0012 (*RETURN 1.)
END
```
Fig. 10(b)

```
G0001 (PROCEDURE FIB 1. )
G0002 (*DISTRIBUTE (G0001 MONO-0) )
G0003 (*EQ (C-1 1.) (G0002 2.) )
G0004 (*EQ (C-1 2.) (G0002 1.) )
G0005 (*GT (C-1 2.) (G0002 1.) )
G0006 (*CONSTANT (C-1 1.) (G0004 1.) )
G0007 (*DIFFERENCE (C-1 1.) (G0005 1.) )
G0008 (*DIFFERENCE (C-1 2.) (G0005 1.) )
G0009 (*CALL FIB 1.1. (G0007 0.) )
G0010 (*CALL FIB 1.1. (G0008 1.) )
G0011 (*PLUS (G0009 1.) (G0010 2.) )
G0012 (*RETURN (G0003 1.) ( G0004 2.) G0011 3.)
END
```
Fig. 10(c)

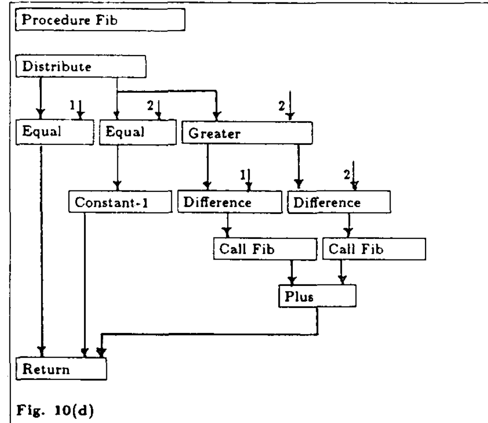


Fig. 10(d)

Fig. 10. Fibonacci—dataflow computing.

- *Numerical operations:*
  Fplus(x y) = x+y Fdifference(x y) = x−y
  Fquotient(x y) = x/y Fremainder(x y) = rem x/y
  Ftimes(x y) = x * y

- *Conditional operations:*
  Fnull(()) = () Fnotnull(x1 · · ·) = (x1 · · ·)
  Fnull(x1 · · ·) = N Fnotnull(()) = N

  Fatom((x)) = (x) Fatom((x1 · · ·) = (x1 · · ·)
  Fatom((x1 · · ·)) = N Fnotatom(x) = N

  Fnumberp(1) = 1 Fnotnumberp(1 · · ·) = 1 · · ·
  Fnumberp(1 · · ·) = N Fnotnumberp(1) = N

  Fequal(x x) = x Fnotequal(x x) = N
  Fequal(x y) = N Fnotequal(x y) = x

TABLE I
BASIC DEFINITIONS

Common	*non-strict*	conditional-IL	conditional-EMIL
1.CAR: First element of an input list	1.HCONS: Availability of first argument gives that value as the output	1.ATOM: Atom for atom input, freeze otherwise	1.ATOM: True for atom input, false otherwise
2.CDR: List of input list other than first element	2.TCONS: Availability of second argument gives that value as the output	2.NOTATOM: Freeze for atom input, Input otherwise	2.NUMBERP: True for integer input, false otherwise
3.CONS: Combined *list of two input lists*	3.AND: Availability of any argument with the value FALSE gives that value as the output. Otherwise *TRUE is the output.*	3.NUMBERP: Integer for integer input, freeze otherwise	3.EQUAL: True for equal inputs, false otherwise
4.PLUS: Addition of two inputs	4.OR: Availability of any argument with the value TRUE gives that value as the output. Otherwise FALSE is the output.	4.NOTNUMBERP: Freeze for integer input, input otherwise	4.NULL: True for null input, false otherwise
5.DIFFERENCE: Difference of two inputs		5.EQUAL: Right input if equal, freeze otherwise	5.GREATERTHAN: True if right input is greater than left, false otherwise
6.TIMES: Multiplication of two inputs		6.NOTEQUAL: Right input if not equal, freeze otherwise	6.LESSTHAN: True if right input is less than left, false otherwise
7.QUOTIENT: Division of one input by other		7.NULL: Null for null input, freeze otherwise	7.SWITCH-T: T: Freeze if not true, switch input otherwise
8.REMAINDER: Remainder of division of two inputs		8.NOTNULL: Input for not null input, freeze otherwise	8.SWITCH-F: F: Freeze if not false, switch input otherwise
9.DISTRIBUTE: Distributes input		9.GREATER: *Right* input if greater than left, freeze otherwise	
10.CONSTANT: Constant data when input is received		10.NOTGREATER: Right input if not greater than left, freeze otherwise	
11.PROCEDURE: Defines procedure			
12.CALL: Calls procedure			
13.RETURN: Returns value of procedure			

for integers x greaterthan y
Fgreater(x y) = y Fnotgreater(x y) = x
Fgreater(y x) = N Fnotgreater(x y) = N

- *nonstrict operators:*
  Fhcons($x1$ ($x2$ $\cdots$)) = $x1$
  Ftcons(($x1$, $x2$ $\cdots$) xn) = xn
  Fand($\cdot\cdot F \cdot\cdot$) = F Fand(T,T,T $\cdots$ T) =T
  For($\cdot\cdot$ T $\cdot\cdot$) = T Fand(F,F,F $\cdots$ F) =F

IV. Dataflow Computing Machines

Considerable progress has been made in building dataflow machines during last few years to support intelligence computations [52]–[56]. Dataflow machines contributed to advance in building parallel systems. Recursive computations are implemented using tags or code copying. Software simulation is the most economic way to verify the effectiveness of the dataflow computing concept and to identify and solve some problems. Real hardware prototypes help to identify and solve hardware problems. Larger programs can be executed at a higher speed in large scale prototypes with sufficient resources.

A. Static Machines

1) VIM: The Dennis group at MIT introduced the dataflow computing concept and laid the foundation for most other dataflow projects [1]–[5]. Research and development projects on dataflow computing started in 1968, and a 1 GFLOP VAL interpretive machine, VIM, is being developed. The group's contributions include basic and advanced dataflow computing models, design of dataflow graphs, dataflow computing languages, and computer architecture. Their main objective is to prove the feasibility of the practical application of static dataflow computing with acknowledgment signals for large scale numerical computations. Acknowledgment signals provide safe execution of the computation. In static computing, data tokens are stored in an instruction or a copy of the instruction. Instruction has an operation code to holding operand values and destination fields. The nodes of a program are loaded to memory before the computation begins and, at most, one instance of a node is enabled for firing at a time. To activate an instruction operand fields must be filled and acknowledge signals must arrive. Enabled nodes are detected by associating a counter with each node. Resource allocation decisions are made by the programmer or compiler. Computations that do not contribute to the final result are avoided by demand driven processing. The system has been implemented as an interpreter on a Lisp machine and eight PE multiprocessor prototype. Benchmark programs such as the weather model, Navier-stoks problem, and plasma simulation were executed. A larger prototype with 1024 cell blocks, 1024 functional units, and 32 array modules has been proposed.

1) Global Configuration: The VIM machine consists of a routing network, cell blocks, functional units and array memories, shown in Fig. 11. Interconnection network tolerate the latency. Cell blocks store program graphs, operations, operands, destination addresses of nodes, and recognize the instructions ready for execution. The functional unit performs operations on data values. Array memories store array structures.

2) Processing Elements: Functions of PE's are performed by cell blocks and functional units. Simple instructions such as duplicating values and performing tests are executed within the cell block. A PE, with instruction enabling and execution mechanisms, consists of an update unit, an operation unit, a queue, a fetch unit, and an activity store. The activity store holds dataflow instructions. The fetch unit picks addresses of an enabled instruction from the queue, fetches that instruction with its operands from the activity store and delivers it to the operation unit. Instruction execution gives result packets which are sent on to the update unit. The update unit enters the address of the enabled instruction in the FIFO queue. If the target instructions of a result packet reside in some other PE, the packet is sent off through the network. Program graph execution terminates when none of the nodes is enabled. Streams are handled by pipelining. There is no scheduler to assign nodes to a processor. Faults in the machine require restarting the computation from the beginning.

3) Packet Formats and Instructions: Result packets consist of a result value and a reference. Control packets contain boolean values and control values. Data packets contain integer or complex values. Floating point, fixed point, logical packet communication, and shift instructions are used in the processor instruction set.

2) Texas Distributed Data Processor: The DDP was designed by Texas Instruments [6], [7]. The project started in 1976 and the DDP has been in operation since 1978. The main objective is to investigate the feasibility of static dataflow computing without acknowledgment signals for high speed computing systems. The DDP uses

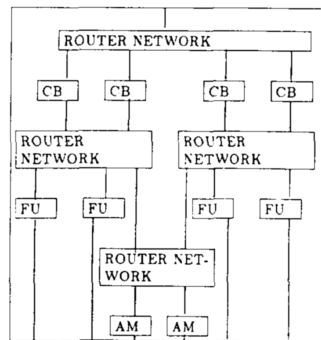


Fig. 11. VIM architecture.

a strict compound branch node to prevent the initiation of a new iteration before the completion of the previous iteration. The DDP is implemented in TTL. Each dataflow computer contains 32K words of MOS memory. ADA is used in the four-processor DDP at the computer science department of the University of Southwestern Louisiana.

a) Global Configuration: The DDP shown in Fig. 12, consists of four identical dataflow computers to execute programs and a TI 990/10 minicomputer acting as a front-end processor. These computing elements are connected by a DCLN ring.

b) Processing Elements: Each dataflow computer consists of an arithmetic unit which processes executable instructions, a program memory holding dataflow instructions, an update controller which updates instructions with tokens and a pending executable instruction queue. Each node is associated with a counter. When an instruction completes execution, a series of token packets is released to the update controller which stores the token operand in the instruction and decrements the count by one. If this count is zero, the instruction is executable and is placed in the instruction queue. Two communication paths are used: one for transmitting instruction packets and result packets, and the other for maintenance and diagnostic purposes. A maintenance controller detects faulty processors. Computations can restart at the preceding checkpoint. A maintenance bus provides communication facilities to monitor the performance of each processor, to load and dump contents of the memory, and to diagnose the faults. The local memory of the processor has an instruction memory and a data memory. Result packets are stored in data memory. Recursive computations are not supported.

c) Packet Formats and Instructions: Instruction packets use up to fifteen 35-bit words. An instruction can have up to 13 input and 13 output arcs with a total of 14 input and output arcs. Result packets are two words long and contain routing information and data. Monitor call, semaphore instructions and pipelining are used in implementing streams. Floating point instructions, fixed point instructions, logical and shift instructions, loop control, memory fetch and communication with front end processor oriented instructions support Fortran IV programs.

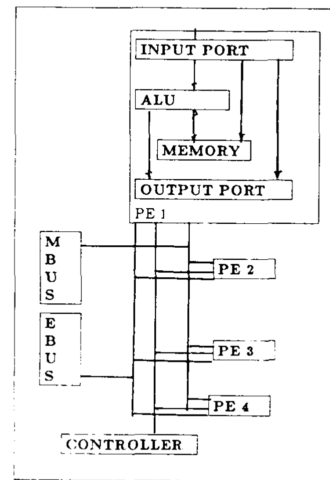


Fig. 12. Texas distributed processor.

FETCH, STORE, and MC instructions handle instructions and data.

3) LAU System: The LAU project started in 1976 at the CERT Laboratory, Toulouse, France [45], [46]. The LAU machine has been in operation since 1979. The group designed the LAU high-level single assignment language, programmed a large number of problems, and implemented a compiler and detailed simulator.

a) Global Configuration: The machine consists of a memory unit, control unit and 32 processing units, shown in Fig. 13. The memory unit stores instructions and data. The control unit maintains the control memory. Six unidirectional buses are used for communication.

b) Processing Elements: Each processing element is built in 16-bit microprogrammed processes using AMD 2900 bit slice microprocessors. Execution units read data from central memory. Enabled instructions are kept in a ready instruction queue until results come out of the processor. This helps to reassign instructions to a healthy processor. Enabled nodes are detected by associating a counter with each node. The memory unit stores instructions and the data control unit maintains control memory. The von Neumann program counter is replaced by an instruction control memory which handles instructions and a data control memory which handles data.

c) Packet Formats and Instructions: Each node can have a maximum of two input arcs and several output arcs. The length of instruction and data packets are 64-bits. The LAU system does not handle stream data structures. The instruction set includes fixed point, logical, shift, control instructions such as CASE, LOOP, CALL, RETURN, and EXPAND.

4) NEDIPS: NEDIPS and IPP were the first commercially available dataflow processors [64]. They are special purpose dataflow processors with static architecture, well tuned to image processing applications developed by the Nippon Electric Co. NEDIPS is a 32-bit machine for scientific computation and uses high speed logic. The Image

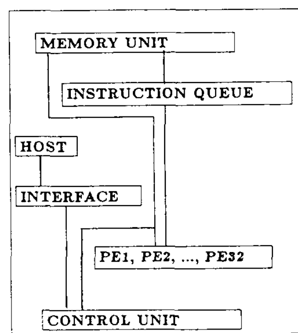


Fig. 13. LAU system.

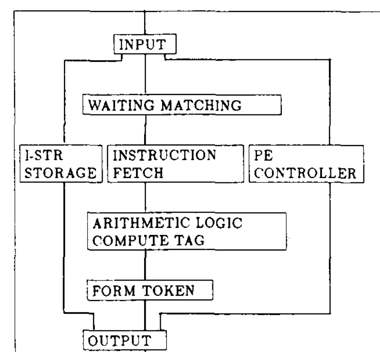


Fig. 14. MIT tagged token dataflow PE.

Pipelined Processor (IPP) is a single chip processor of similar architecture. This processor is a building block for highly parallel image processing systems. Special mechanisms are used to implement multiple tokens per arc. Special hardware operations are provided for generating, splitting and merging streams of tokens.

B. Dynamic Machines

1) MIT Tagged-Token Dataflow Machine: The Irvine dataflow project started in 1975 at the University of California at Irvine and is being continued at MIT by Arvind's group [8]–[11]. The major contributions include the tagged token dynamic computing model, I-structures, Id language and computer architecture. The main objective is to exploit VLSI and provide highly concurrent program organization. A 32-PE machine using Symbolic Lisp machines is being constructed. A 256 board 1 BIP machine is under construction.

a) Global Configuration: This asynchronous machine has 64 processing elements connected via an n-cube communication network. The organization minimizes communication overhead by matching at the processing element holding the storage instruction and bypassing the network to the processor itself.

b) Processing Elements: A PE consists of the input, waiting matching, instruction fetch, service, and output sections, shown in Fig. 14. The input section accepts inputs from other processing elements, the waiting matching section forms data tokens into sets for one instruction, the instruction fetch section fetches executable instructions from local program memory, and the output section routes data tokens containing results to the destination processing element. Enabled nodes are detected using tags carrying the information of the node. Tagged stream elements are processed in parallel using multiple instances, one for each element. Program memory stores instruction codes. The data memory, an I-structure memory, stores arrays. Waiting matching storage, an associative memory, matches or stores the incoming tokens. The allocation of memory and tags is controlled by a manager. Recursive computations are supported by tagged tokens. Processing units asynchronously evaluate the executable instruction packets. Faults require computations to be restored at the previous checkpoint.

c) Packet Formats and Instructions: There is a maximum of two input tokens and several output tokens per node. Thirty-two enabled nodes can wait for the ALU. The instruction set includes floating point, fixed point, and logical instructions. The instruction and data packet lengths are 33 and 71 bits.

2) Manchester Dataflow Computer: The Manchester project started in 1975 by the Gurd–Watson group [12]–[15] at Manchester University. The group's main objective is to investigate the use of tagged token dataflow computing concept for very high speed dynamic computing systems. They completed the construction of 20 processor, strongly typed, tagged dataflow machine in 1980 using Schottky bit slice microprocessors. Their contributions include tagged token dynamic computing model, several high level dataflow computing languages and the dataflow machine. The reported performance of the machine is approximately 1.6 MIPS.

a) Global Configuration: The machine consists of a switch, token queue, matching unit, instruction unit and a processing unit, Fig. 15. A switch provides input and output for the system. The token queue is the FIFO buffer providing temporary storage for tokens. The matching unit matches pairs of tokens, employing hardware hashing. The instruction store holds dataflow programs and PE's execute instructions.

b) Processing Elements: There are fifteen functional units in the processor. One enabled node is assigned to each functional unit. The node store supplies enabled nodes to the processing unit. Enabled nodes are assigned to the functional units using any hardware distributor. Therefore, there are no multiple assignments to a functional unit. The matching unit can hold 16K units and employ dynamic hashing. The PU consists of distribution and arbitration systems and a group of microprogrammed microprocessors. Streams are processed in parallel using multiple instances, one for each element. Recursion computations are supported using tags.

c) Packet Formats and Instructions: The instruction set supports floating point, fixed point, data branch, token label, flow control, and token relabeling instructions. A maximum of two input arcs and two output arcs is allocated to an operation. The lengths of instruction and data packets are 167 and 96 bits.

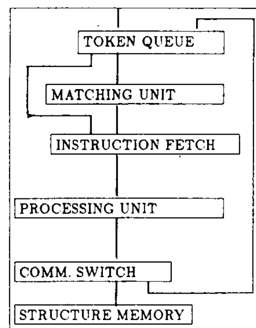


Fig. 15. Manchester machine.

3) DDM1: The Data Driven Machine project [16], [17] started in 1975 by the Davis group at Burroughs Interactive Research Center. Construction was completed in 1976, and the machine is now at the Utah University. This dynamic machine employs FIFO queues instead of tagged tokens to distinguish computations. Program execution and machine organization are based on recursion. The DDM1 is in operation and has been used to study basic issues in dataflow. The DEC20/40 is used for software support. The graphs are generated from a high-level functional language GPL. Parenthesized strings in dataflow programs provide localized dynamic computing.

a) Global Configuration: The machine is composed of an octary tree hierarchy of computing elements. This hierarchy exploits VLSI, utilizing the locality of reference to reduce the communication and control problems.

b) Processing Elements: A PE consists of an atomic storage unit, an atomic processor, an agenda queue, an input queue, an output queue, and a switch, Fig. 16. The atomic storage unit, a 4K–4bit character store, is the program memory. The atomic processor is the execution unit. The agenda queue is the message store for the local atomic storage unit. The input queue is the buffer to the messages from the superior element. The output queue is the buffer to the messages to the superior element. The switch connects to eight computer elements. The tree structure inhibits immediate rerouting of the results before the fault.

c) Packet Formats and Instructions: Data tokens provide all communications. Each instruction is represented as a variable length instruction packet. Each instruction has an enabling counter for input arcs. An instruction can have any number of input and output arcs. Streams are handled by pipelining the tokens. The atomic processor processes integer-oriented, logical, indexed read and write, and relational operator-oriented instructions.

4) SIGMA-1: The SIGMA-1 project [48]–[51] was started in 1982 by Yuba's group at the Electrotechnical Laboratory (ETL). Their main objective is to develop a large scale tagged token dynamic dataflow machine with 100 MFLOPS performance for scientific and technological computations. SIGMA-1 uses a C-like high-level dataflow computing language, DFC (Dataflow C), and SAS intermediate language to describe the dataflow graphs. A preliminary version of the PE and SE using advanced Schottky TTL logic and MOS memories has been in operation, with 1.3 MIPS, since November 1984. The final version of a single group uses gate-array LSI chips. The full hardware configuration with the total predicted performance of 100 MFLOPS is in operation now.

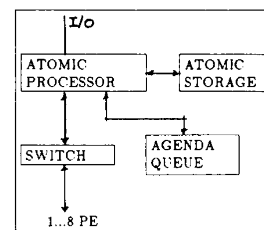


Fig. 16. DDM1.

a) Global Configuration: The SIGMA-1, shown in Fig. 17(a), has 128 PE's and 128 structure elements (SE's) which are divided into 32 groups connected by a two level hierarchical network. This hierarchy corresponds to parallel execution of iterations and procedure calls which appear frequently in numerical computations. A single group consists of four PE's and four SE's connected by a 10 × 10 crossbar switch. The remaining two ports of the switch are used for the interfaces to the global network and the maintenance architecture. The global network is a two stage omega network.

b) Processing Elements: A PE, shown in Fig. 17(b), consists of several functional units, each of which works synchronously and constitutes a two stage pipeline. A chained hashing hardware with 64K cells is used as the matching memory unit. Each PE consists of about 81K logic gates, using nine types of 28 gate-array LSI's. An SE controls array structures allowing single write and multiple read operations. It is implemented by memory of 256K cells where each cell is attached with a waiting queue for asynchronous access control.

c) Packet Formats and Instructions: The data transfer between PE's and SE's is in fixed length packet form. A packet consists of the PE or SE number (8 bits), the cancel bit, the destination identifier (28 bits), the tagged data (40 bits), and miscellaneous information (12 bits). The length of the instruction is 40 bits in a primitive format. The first 20 bits indicate the operation to be performed, and the next 20 bits indicate the destination address of the result data. It is possible to allocate a maximum of three different destinations to an instruction.

5) EM-3: The EM-3 project was started by Yuba's group at ETL [27]–[33] in 1982. The objectives of the project include evaluating the effectiveness of pseudoresult dataflow computing for symbolic manipulations, implementing new parallel architectures, and evaluating the performance of a hardware simulator by executing application programs. The eight PE prototype started operation in 1984, and the 16 PE organization has been in operation since 1985. It is used to implement new parallel control mechanisms. The maximum performance of the hardware

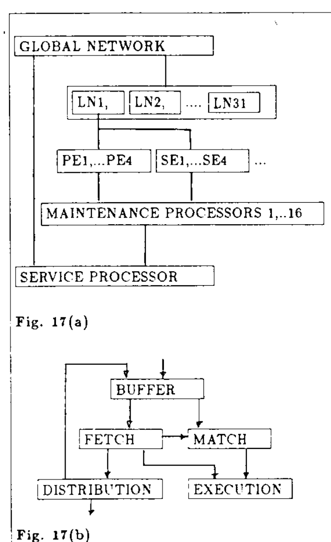


Fig. 17. SIGMA-1—PE.

is about 10 MIPS. An advanced version of the EM-3 which will be a more practical dataflow computer prototype is being developed.

a) Global Configuration: Sixteen identical PE's are connected via a packet communication network. There is no locality in the network. The router network is adopted for communication and a special gate-array LSI chip has been developed for this purpose. The LSI chip is a 4-bit slice 4×4 router and the transfer rate of a packet through the network is 150 nanoseconds.

b) Processing Elements: A PE is constructed using MC68000 microprocessor with special hardware, shown in Fig. 18. Almost all the functions, including the function evaluation mechanism, are performed sequentially within the PE. The MC68000, the packet memory control unit used as the network interface, and the I/O interface to the host computer are connected by a common bus. Each PE comprises three boards excluding the interface to the host computer PDP-11/44 and the network boards. Packet memory is accessed from the microprocessor, and each packet is represented as a pointer to packet memory. Hence, there is no overhead in moving packets in a PE.

c) Packet Formats and Instructions: The 96-bit result packet carries output data of an operation. A result packet consists of the PE number (8 bits), the type-of-packet field (4 bits), the packet length (4 bits), the destination-identifier (48 bits) and tagged data (32 bits). The packet is divided into six 16-bit segments in the network. The length of an instruction is 48 bits, comprising the 32-bit destination field and the 16-bit operation field. The immediate data (32 bits) can be contained, and the number of destination fields within an instruction is not fixed.

6) EDDY: Amamiya *et al.* [22] at Nippon Telegraph and Telephone Corporation (NTT) started research and development on dataflow machines in 1980. A dataflow processor array system for scientific and technological computations, EDDY, was set up as a prototype in 1983. High speed was achieved by adapting the operational characteristics of scientific and technological computations to the machine architecture at a hardware level. Application programs were written in VALID. The machine exploits parallelism inherent in the application programs, and its performance was not sensitive to inter-PE communication delay or to load imbalance.

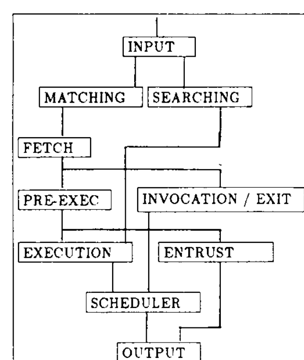


Fig. 18. EM-3 PE.

a) Global Configuration: Sixteen PE's are connected in a 4×4 cellular array structure. Each PE connects directly to eight neighbouring PE's. There are two broadcast control units for loading programs and data to each PE, which are located at the interfaces between the host computer, PDP-11/60, and a set of PE's.

b) Processing Elements: Each PE is constructed using two Z8000 microprocessors. One controls the communication and the other controls the dataflow and execution of instructions. The tagged token concept is applied for function invocation and iteration handling. Each array element in a program is given a unique identifier, and all elements are processed in parallel. Each PE works logically as a circular pipeline, but practically, each functional unit within a PE operates sequentially.

c) Packet Formats and Instructions: A data packet consists of the identifier (color), the destination field and the value field. An identifier comprises the array element name, the instantiation name and the loop count. An instantiation name corresponds to a procedure instance name and is statically determined at compiling time by caller–callee analysis. The array elements are also statically allocated to the PE's according to a specific mapping strategy. The instruction contains almost the same information as the data packet except for an operation code.

7) DFM: Amamiya *et al.* [21]–[26] at NTT started the DFM project in 1982. Their main objective is to develop a dataflow machine for symbolic manipulations [19] to [25] using lenient and lazy cons mechanisms. In 1985, the construction of the DFM-II was started using CMOS gate-array technology. Parallel processing of the DFM is realized by parallel evaluation of function arguments, partial execution of a function body and pipeline processing of a delayed evaluation scheme. The two PE version of the DFM has been in operation since the beginning of

1986. The basic cycle of a PE is 180 nanoseconds and the maximum speed is about 1.4 MIPS per PE.

a) Global Configuration: Several clusters were connected via a network, shown in Fig. 19. A cluster consists of eight PE's and eight structure memories connected by multiple buses. Structure memories are separate from PE's for efficient list processing. Each cluster is supervised by a cluster control unit. The cluster control unit controls the load balancing among PE's within the cluster and communicates with other clusters via the network or the host computer. The two level network is based on clustering. The load distribution is within a cluster, and function distribution is among clusters. A blocked content addressable memory scheme is applied to reduce the amount of hardware.

b) Processing Elements: Each PE is composed of an instruction memory, an operand memory and an execution unit. The matching unit contains content addressable memory for each function activation. These units work as a circular pipeline. A hardware queue is placed at the entrance of the instruction memory to ease packet traffic in the circular pipeline. Each structure memory is constructed by multiple memory banks equipped with the list operation unit. Each cell of the structure memory is composed of the cell type field (one bit), the reference count field (9 bits), the CAR field, and the CDR field (23 bits each).

c) Packet Formats and Instructions: The size of a result packet is 56 bits and its contents are the destination-identifier with the function name (24 bits), and data (32 bits). There are instructions to the cluster control unit and to the structure memory as well as to the execution unit. An instruction to the execution unit consists of the operation field (8 bits), two operand-fields (32 bits each), and the destination-identifier. An instruction to the structure memory is 90 bits and is associated with the 3-bit PE number.

8) PIM-D: Itoh *et al.* [58] at ICOT began the research and development of a dataflow PROLOG machine in the middle of 1982. The objective of ICOT is to develop all computer related technology from the viewpoint of predicate logic. Dataflow architecture, logic programming and natural language understanding are three research directions identified for this paradigm. Three different types of architecture, the PIM-D, a parallel reduction machine and a parallel inference machine with an efficient task distribution mechanism, were studied to overcome the highly parallel processing problems in logic programs. The PIM-D employs the breadth-first search. To avoid the deadlock problem caused by the number of processes, each process is associated with execution priority. The eight PE PIM-D is in operation now. LSI implementation is being developed.

a) Global Configuration: The machine consists of 16 PE's, 15 structure memories (SM's) and a three level hierarchical network, implemented by a 113-bit bus, shown in Fig. 20. The PE's and SM's are divided into four clusters, each of which consists of four PE's and four SM's, except for one cluster. Each bus is connected by the network node with a 128 packet buffer. The minimum transmission time is 450 nanoseconds per packet.

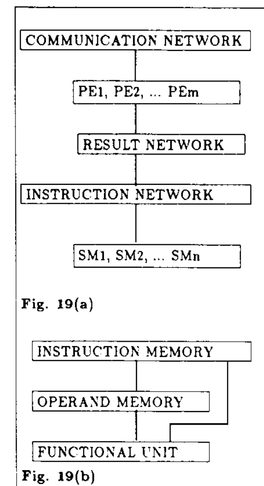


Fig. 19. DFM—PE.

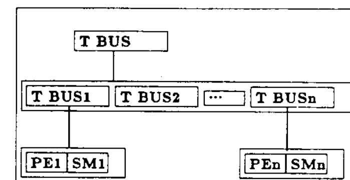


Fig. 20. PIM-D architecture.

b) Processing Elements: A PE is composed of a packet queue, an instruction control unit, and two atomic processing units for execution which are also connected via a bus. The instruction control unit serves as the matching function of dataflow control. Each hardware unit is constructed using bit-sliced microprogrammable processors and TTL IC's.

c) Data Formats: A packet transferred between PE's via a bus consists of the PE/SM number (5 bits), the packet type (9 bits), the packet color (16 bits), the destination identifier (24 bits) and the operand data (32 bits). The length of the instruction is 59 bits. A cell of a SM is composed of data (32×2 bits), the type flag (2×2 bits), and the reference count area (10 bits).

9) TOPSTAR: The TOPSTAR [59] is a macro dataflow machine, developed from 1978 to 1982 by Suzuki *et al.* at the University of Tokyo [53] to support the recognition of printed Chinese character patterns. The TOPSTAR-I, composed of three PM's and two CM's, is the prototype of the more advanced TOPSTAR-II. The TOPSTAR-II was easily expandable by plugging in additional modules. Both machines were in operation with the system software. Data buffers causes the pipeline effect. Using TOPSTAR-II as a testbed, some experimental studies such as the dataflow Lisp compiler, logic simulation and parallel PROLOG implementation, were carried out.

This led to the development of the parallel PROLOG machine called the PIE.

a) Global Configuration: Sixteen processing modules (PM) and eight communication and control modules (CM) are organized in a bipartite graph. Each PM is connected to a maximum of four CM's, while each CM is connected to a maximum of eight PM's. Each PM or CM is separated from each other but has indirect paths through CM's or PM's. A procedure level dataflow graph is dynamically mapped into the PM-CM connection network. Each PM interrupts one of the connected CM's and requests to allocate a task. Each CM contains allocated procedures of an execution program, and if executable tasks exist, their instances as well as their argument data are sent to the requested PM.

b) Processing Units: Each PM or CM is constructed using a Z-80 microprocessor and a direct memory access (DMA) controller. The communication between PM and CM is through the DMA system at high speed because the data block is transferred when a new instance of a procedure is needed at an allocated PM. The CM's communication memory is shared with each PM and contains execution programs and their argument data.

c) Data Formats: The data packet has a variable length and consists of the serial number, the field indicating the stack depth, the destination addresses and the procedure instance. The serial number corresponds to a color, and the stack depth is used for recording the history of the data passed. The data format supports the implementation of the control mechanism of iteration and recursion.

C. Other Projects

In addition to the projects mentioned above, there are many other dataflow research projects. These include projects at the University of Southern California [47], Hughes Aircraft Company, University of Adelaide [65], University of New South Wales, Keio University, Japan [31], [57], Osaka University [62], Gunma University [57], Tokyo University [61], and Indian Institute of Technology.

D. Problems in Dataflow Computing Machines

1) Matching Bottleneck: The dataflow processing element must consist of mechanisms to recognize the data tokens to an operation, perform the execution of operation and dispatch unit to distribute result data tokens. Hence, a dataflow processing element basically consists of a matching unit, instruction fetch unit, execution unit and distribution unit, shown in Fig. 21. The execution unit performs the execution and structure handling. Matching performs the synchronization of multiple segments of execution. The matching unit sequentially matches and synchronizes the operands of double-operand operations for execution. No matching is necessary for single-operand operations. The larger the number of double-operand operations, the more matching to be performed in the matching unit. This narrows the pipeline between the matching unit and execution unit and reduces the computing speed. This is the matching bottleneck.

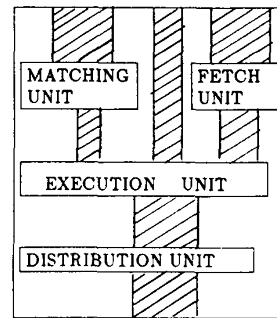


Fig. 21. Dataflow PE.

2) Remaining Packet Garbage: The large number of unexecuted packets waiting in the matching unit after completing the execution of computation is remaining packet garbage, RPG. RPG is generated due to vertical branches created by conditional computations and multiargument functions. A conditional computation divides the dataflow computing into two vertical branches. Execution of the conditional computation makes one branch LIVE and the other DEAD. Data flow to the operations ignores the liveness of the branch. Loading only one operand of a double operand operation in the DEAD branch results in RPG. The multiple-arguments in a function divide the dataflow computation into live vertical branches. RPG is created when a DEAD branch of a conditional computation in one vertical branch receives the data values from the same vertical branch and/or from some other vertical branch.

3) Control of Parallelism: Dataflow computations have huge parallelism, many times larger than the parallelism available in the hardware. Such computations tend to use excessive amounts of storage since many partial results are created long before resources are available to process them. Therefore, it is necessary to restrict excess program parallelism to approximately match machine parallelism.

The tokens generated must be dispatched to the corresponding nodes. The more tokens generated the greater the communication delay. The parallelism can be used efficiently to hide latency. Balanced load distribution among PE's increase the performance of the system and utilization of the resources. Unbalanced load distribution and heavy load degrade the performance of a dataflow system. Therefore, efficient techniques to reduce unnecessary token generation and efficient communication ways are necessary.

4) Sequential Computing Segments: A sequential computing segment is a program segment in which the maximum parallelism is less than the number of processors and/or pipeline stages. Such computations are involved in conditional computations, recursive procedure calls and iterative computations which use previous computing results to continue computation. The performance

of the sequential computing segments in a program is important in parallel computing.

5) Parallel Execution of Conditional Computations: Communication control systems and production systems consist of a large number of parallel conditional computations. Nondeterminate computations such as guarded expressions can be implemented in parallel with the use of the simple switch-t operation. The guarded condition evaluated as true enables successive computations to return the value of the guarded expression evaluation. The switch operation with two outputs used in the conditional computing implementation cannot be used in the parallel implementation of conditional computations without additional complex control mechanisms.

6) Optimization of Dataflow Computations: In functional languages, different occurrences of an expression always yield the same value. Loop invariant expressions produce the same value on each pass of the loop. The value produced by such common computation, CC's, is evaluated once and can be used at all occurrences of the expression. Reusing CC's within conditional expressions saves computational effort. Detecting and arranging CC's in conditional expressions to optimize program execution in dataflow machines is a complicated process. Identifying nonstrict operators or functions and not evaluating the arguments that will not contribute to the final result of the computation optimizes dataflow computations. Such nonstrict operators are identified and added to enhance the efficiency of dataflow computing systems.

V. Performance Evaluation Using the EM-3

A. EM-3 Operational Model

The functional configuration of the dataflow computing element is shown in Fig. 18. The result packets received at the input section are checked and the packets corresponding to single-operand operations are sent to the operation fetch section. The result packets corresponding to double-operand operators or to a function which has a plural number of arguments are sent to the operand matching section. This section matches and synchronizes the packets. The unmatched packets awaiting their partners are stored in the matching store and searched for when necessary. If the arrived result packet finds its partner in the matching store, both are removed from the store and sent to the instruction fetch section.

The program store is attached to the instruction fetch section and stores the program to be executed. The operation fetch section fetches operations from the program store according to the operation addresses and combines them with their operands to generate internal execution packets.

The invocation section is activated when a call operation is fetched at the operation fetch section. A call operation invokes a defined function. The result store, simulating the pseudoresult control mechanism, is handled by the search, invocation, exit, and execution sections. This store consists of a result table, a deferred buffer, and a storage for list cells. The result table manages a set of pseudoresults. Each entry consists of tags and a result value. The deferred buffer is storage for entrust packets until the pseudoresults become actual. The list cell storage is for list cells created by the cons operation. The storage management of the result table and the deferred buffer is carried out by the reference count garbage collection scheme. A pseudoresult identifier is created for the newly invoked function and an invocation packet is generated. This packet is sent to the PE scheduler section.

The initiation section accepts invoke packets and extracts a function name. Its arguments are placed in the packets and generate result packets corresponding to each argument. The body of the function is activated by these result packets.

The search section is activated by an entrust packet which is associated with a pseudoresult identifier. The pseudoresult table is searched using the pseudoresult identifier for the actual result. If found, the entrust packet is sent to the execution section. If not found, it is stored at the deferred buffer and waits for the completion of the predecessor operation assigned by the pseudoresult identifier.

The exit section stores the values of actual results or pointers to semiresults which correspond to each pseudoresult of the function and are stored in the pseudoresult table at the exit of each section. If the activated entrust packets corresponding to the exit operation and waiting for completion of the function are executable, they are sent to the execution section, otherwise they are sent to the entrust section.

The entrust section generates entrust packets and defers the execution of the operation when input packets include a pseudoresult. The generated entrust packets are sent to the PE assigned by the pseudoresult identifier. The scheduler section decides the destination PE by using a hashing function. The preexecution section examines the operands of an execute packet. If there is a pseudoresult in an operand, the packet is sent to the entrust section, otherwise it is sent to the execution section. The output section sends external packets through the communication network to the corresponding destination PE's.

B. EMIL

EMLISP is the high-level language and EMIL is the low level language used in the EM-3. Column 4 of Table I shows the basic definitions of EMIL codes. NULL, NUMBERP, ATOM, LESSTHAN, EQUAL, and GREATERTHAN are conditional operations. SWITCH-T and SWITCH-F operations are executed with all the conditional operations. Distribute, procedure, constant, call and return are dataflow computing support codes. The functionality of EMIL operations is summarized below. Here, E, T and F imply error, true and false, and integer x is greater than integer y.

- *List operations:*
  Fcar(($x1$ $x2$ $\cdots$)) = $x1$ Fcar(()) = E
  Fcdr(($x1$ $x2$ $\cdots$)) = ($x2$ $x3$ $\cdots$) Fcdr(()) = E
  Fcons($x1$ ($x2$ $\cdots$)) = ($x1$ $x2$ $\cdots$)

- *Attribute checking:*
  Fatom(a) = T Fatom($x1\ x2 \cdots$) = F
  Fnumberp(1) = T Fnumberp($1 \cdots$) = F
  Fnull(()) = T Fnull($x1 \cdots$) = F
  Fequal($x\ x$) = T Fequal($x\ y$) = F
  Fgreaterthan($x\ y$) = T Fgreaterthan($y\ x$) = F
  Flessthan($y\ x$) = T Flessthan($x\ y$) = F
- *Numerical operations:*
  Fplus ($x\ y$) = $x+y$ Fdifference($x\ y$) = $x-y$
  Fquotient($x\ y$) = x/y Fremainder($x\ y$) = rem x/y
  Ftimes($x\ y$) = xy

C. Performance Evaluation Measurements

This section discusses the experimental results obtained by executing Benchmark programs in the EM-3 [28]–[34]. The software simulator, written in SIMULA, describes the EM-3 dataflow computing environment, which interprets and executes the EMIL code that describes the dataflow computation. The Fibonacci function (F(13)), the Ackermann function (AK(2 9)), sequential and parallel versions of the n queen problem (4QS and 4QP), quicksort algorithm with maximum parallel data (QUI), Fibonacci function F(13), same-fringe (SF) and copy (CP), are some of the nonnumerical and numerical computations performed in the EM-3.

1) Effectiveness of Pseudoresult Model: Fig. 22 shows the performance measurements of SF with and without pseudoresults. When not using pseudoresults, the execution of CONS is deferred until the operands become actual data values. In this case, pseudoresults are generated but never used in any instruction. The execution time difference in a single PE configuration is due to the entrust packet overhead. The number of entrust packets used without pseudoresults is five times greater than with pseudoresults. Dataflow parallelism in SF is very small without pseudoresults and the performance is not improved with the increase of PE's. The pseudoresult dataflow computing model revealed the hidden parallelism in Lisp languages and accelerated the program execution in parallel computing environment.

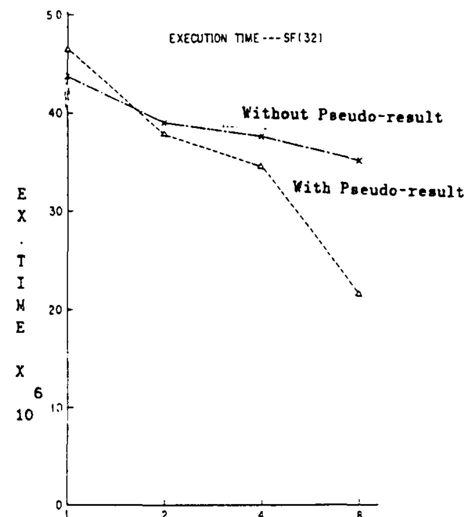


Fig. 22. Execution time variation—pseudoresult.

TABLE II
TIMING PARAMETERS

Operation	Traditional	Not(op)
CAR	2	2
CDR	2	2
CONS	2	4
ATOM	2	4
NULL	2	4
EQUAL	3	5
PLUS	3	3
DIFFERENCE	3	3
TIMES	7	7
PRINT	2	2
GREATERP	3	-
LESSP	3	-
SWITCH-T	1	-
SWITCH-F	1	-
NOTATOM	-	4
NOTNULL	-	4
NOTEQUAL	-	5
GREATER	-	5
NOTGREATER	-	5
CONSTANT	1	-

2) Effectiveness of Not(operation) Model: The effectiveness of the Not(operation) model is observed by comparing the performance characteristics measurements. The timing parameters used for operation executions, shown in Table II, are larger in the Not(operation) model than the traditional model. Figs. 23(a) and 23(b) show EMIL code and corresponding dataflow computing of Fibonacci numbers. Fig. 10 shows Not(operation) based dataflow computing of Fibonacci numbers.

Ideal dataflow parallelism of an algorithm is the number of parallel operations that can be executed in one time step in an idealized machine. The idealized machine consists of unbounded processors and memories, where all operations have equal execution time and operators are executed as soon as operands are available. It is assumed that an unlimited number of concurrent operations can be executed in one time step. Fig. 24(a) shows the ideal dataflow parallelism in F(13) with the traditional computing model. Here, the maximum parallelism, 495 concurrent operations, is observed at the 49th of 85 steps. Fig. 24(b) shows the ideal dataflow parallelism in the Not(operation) model. Here, the maximum parallelism, 261 concurrent operations, is observed at the 34th of 60 steps. This demonstrates the increase in real dataflow parallelism and the removal of a large number of unnecessary computations with pseudo-parallelism, and hence the reduction in dataflow computing cost.

Table III shows the frequency of operations executed in each benchmark program. This illustrates that SWITCH operations account for a large percentage of all operations executed.

Fig. 25 shows the F(13) execution times in the traditional and Not(operation) models for varying the number of EM-3 processing elements. The shape of the graph does not change, but the computing speed approximately doubles in the Not(operation) model. The reasons for speed increase include reduction of double-operand operations, parallelization of sequential computing segments, balanced pipeline stages and reduction of token generation

```
(PROCEDURE FIB 1. (G0002 MONO-0) )
G0002 (*DISTRIBUTE (G0003 MONO-0) (G0006 DATA) (G0010
DATA)(G0014 DATA) )
G0003 (*EQ (C-1 1.) (G0004 MONO-CONTROL) (G0006 CONTROL)
(G0007 MONO-CONTROL) )
G0004 (*SWITCH-T (C-0 1.) (G0018 (RETURN 1.)) )
G0006 (*SWITCH-F (G0005 0.) )
G0007 (*SWITCH-F (C-0 2.) (G0005 0.) )
G0005 (*EQ (G0008 MONO-CONTROL) (G0010 CONTROL) (G0011
MONO-CONTROL)(G0014 CONTROL) (GOO15 MONO-CONTROL) )
G0008 (*SWITCH-T (C-0 1.) (G0018 (RETURN 1.)) )
G0010 (*SWITCH-F (G0009 0.) )
G0011 (*SWITCH-F (C-0 1.) (G0009 1.)) )
G0009 (*DIFFERENCE (G0012 (ARG 1. 1.)) )
G0012 (*CALL FIB 1.1. (G0017 0.) )
G0014 (*SWITCH-F (G0013 0.) )
G0015 (*SWITCH-F (C-0 2.) (G0013 1.) )
G0013 (*DIFFERENCE (G0016 (ARG 1. 1.)) )
G0016 (*CALL FIB 1.1. (G0017 1.) )
G0017 (*PLUS (G0018 (RETURN 1.)) )
G0018 (*RETURN 1.)
END
```

Fig. 23(a)

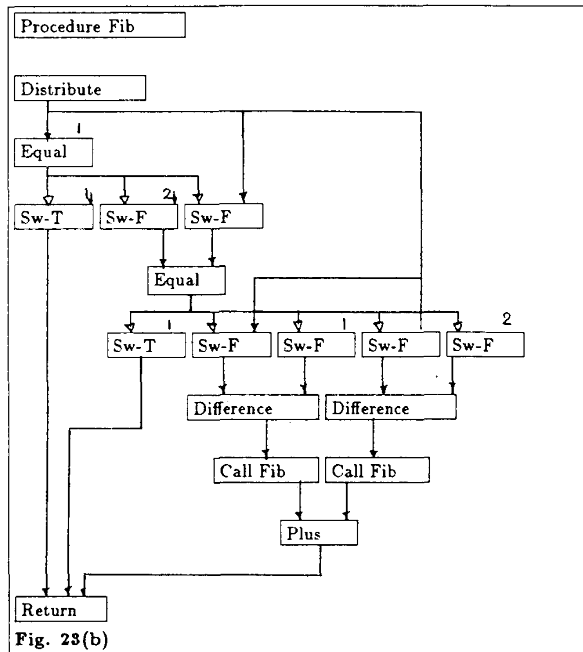


Fig. 23. Fibonacci—traditional.

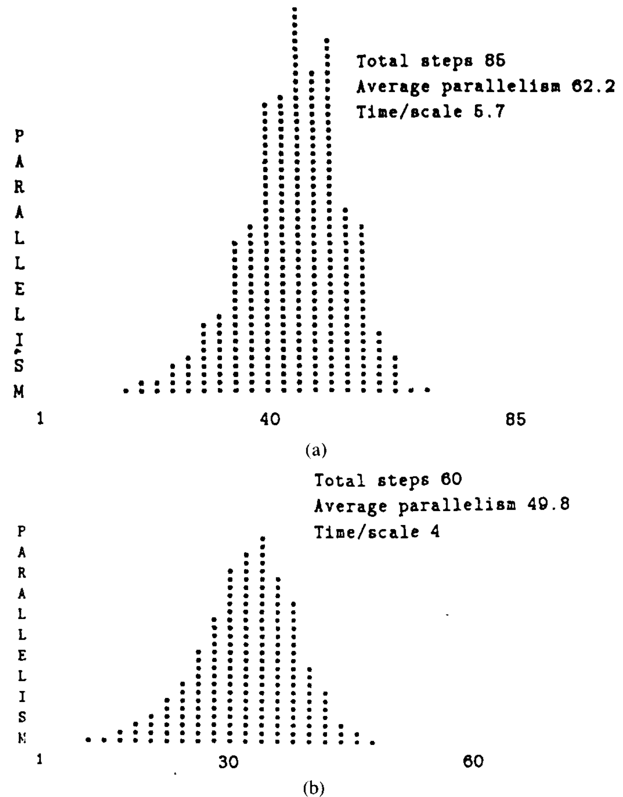


Fig. 24. Ideal dataflow parallelism.

TABLE III
EXECUTION FREQUENCY OF OPERATIONS

	BENCHMARK PROGRAM						
OPERATION	A(2 9)	4QS	4QP	F(13)	QUI	SAF	CP
CAR	0	124	296	0	671	702	189
CDR	0	44	104	0	482	702	189
CONS	0	24	48	0	482	640	189
PLUS	110	125	308	232	0	0	0
DIFFERENCE	229	44	100	464	0	0	0
TIMES	0	0	0	0	0	0	0
DIVIDE	0	0	0	0	0	0	0
SWITCH-F	1060	958	2488	2434	2246	2330	763
SWITCH-T	820	383	900	841	2115	1052	381
ATOM	0	0	0	0	0	126	381
NULL	0	39	90	0	483	0	0
EQL	350	162	390	841	0	736	0
GREATERP	0	0	0	0	0	0	0
LESSP	0	0	0	0	189	0	0
CALL	229	101	241	464	522	798	381
DISTRIBUTE	460	366	875	465	1383	1469	382
PRINT	0	1	2	0	0	0	0
CONSTANT	0	3	3	0	0	0	0

and traffic. In traditional dataflow computing, each single-operand conditional operation must execute two or more additional double-operand operators and must create many packets to support intermediate executions. These packets contribute much to the congestion. Not(operation) based single-operand conditional computation eliminates matching. Tokens are executed directly in the execution unit. In traditional dataflow computing, each double operand conditional operation must execute two or more additional double-operand operators and must create many packets to support intermediate executions. Not(operation) based double-operand conditional computation matches and executes directly to give the result immediately. Unnecessary packet creation is eliminated, thereby reducing congestion.

Fig. 26 shows the average waiting time variation in the matching section in each PE of the 32-PE EM-3. The Not(operation) model reduces the waiting time to one tenth that of the traditional model. This is due to the reduction of a large number of double-operand operations to be executed.

Table IV compares the single-operand packets and double-operand packets generated in the execution of F(13). The number of single-operand packets generated decreased from 8673 to 4148. The number of result packets entering the matching section, double-operand opera-

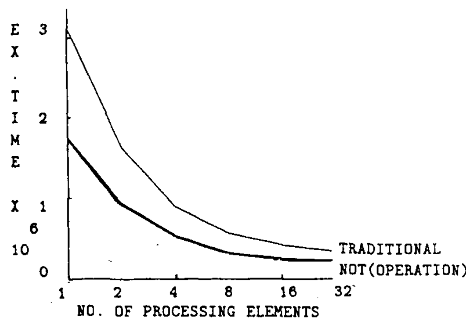


Fig. 25. Execution time variation.

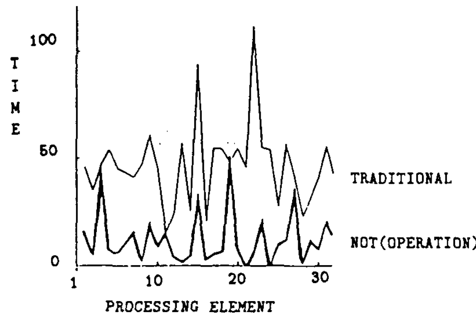


Fig. 26. Matching waiting time—average.

TABLE IV
PACKETS GENERATED

Packet type	Traditional	NOT(OP)
One operand	8673	4148
Two operand	4756	464

tions, decreased from 4756 to 464. The Not(operation) model significantly reduces the number of single-operand packets, double operand packets and operations executed. This results in a proportionate reduction of computing and communication costs and increase in computing speed.

Table V compares the number of operations executed showing the reduction in the number of operations executed. This reduction is due to the elimination of additional double-operand operations used in the execution of traditional conditional computations.

Table VI shows the maximum number of packets waiting in the queue, and the congestion of each functional unit of the EM-3 at the busiest instance.

Fig. 27 shows the maximum number of packets waiting in the queue to the matching section in each PE of the 32-PE EM-3. The maximum number of packets in a single-PE EM-3 with traditional computing is very high compared to the Not(operation). The more tokens that are generated, the greater the communication delay and cost and waiting time in the queues. The Not(operation) computing model provides an efficient way of reducing tokens generated and hence reduces the token traffic.

No RPG is collected in the matching store when executing F(13) with the Not(operation) model. The

TABLE V
OPERATIONS EXECUTED

Operation	Traditional	NOT(OP)
call/return	928	928
distribute	465	465
constant	-	144
switch-t	841	-
switch-f	2434	-
notequal	-	841
equal	841	841
plus	232	232
difference	464	464

TABLE VI
PACKET QUEUE

Section	Traditional	NOT(OP)
input	1	1
match	355	1
ifetch	4	3
preexec	2	2
exec	10	218
invo	1	1
exit	1	1
init	1	1
search	1	1

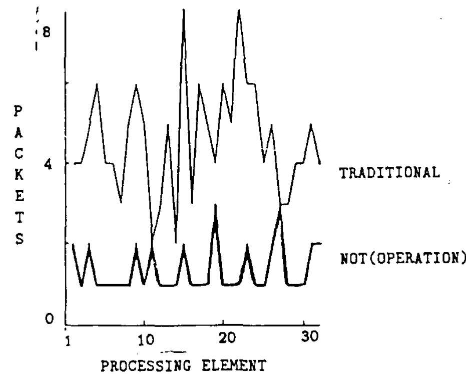


Fig. 27. Maximum packets—matching queue.

Not(operation) completely stops the flow of data from the predecessor of the conditional computation to the dead branch. Dispatching a special packet to execute all the operations in the dead branch, a control operation to stop the flow of data into the dead branch, setting the life of double operand packets or other efficient garbage collecting mechanism must be implemented to remove RPG completely.

The Not(operation) computing model provides an efficient way of implementing parallel conditional computations. The optimization will reduce the number of arithmetic and conditional operations, the size of the dataflow graph, execution time, parallelism, total number of tokens, and token traffic.

VI. CONCLUSIONS

Computing models, languages, and architecture have not changed very much over the last thirty years. Applications of new computing models, languages, and ma-

chines to numerical and nonnumerical computations show promise. The market is responding to the availability of such machines for intelligence computations. Intelligence computing research is highly dependent on high performance low cost parallel computing systems. Researchers are examining radically different approaches. The dataflow computing concept is the most effective, promising computing method to implement in machine architecture for high speed computing. This paper first analyzed the dataflow computing models such as static computing with and without acknowledgment signals, recursive dynamic, tagged token dynamic, eduction, dataflow-control flow, eager-lazy, pseudo-result, and Not(operation).

The research on functional and logic programming languages and their applications to intelligence computations will revolutionize the computer paradigm. Logic programming allows high level program specifications without explicit control directives. Functional languages provide the facility to programmer not to think in terms of storage. Functional programming transforms objects to other objects without naming. The functional languages designed to map algorithms into dataflow computing such as VAL, Id, LUCID, VALID, DFC, and EMLISP were discussed. The DCBL transformation for dataflow computing and its application to Lisp were discussed.

The major difficulty in realizing very high speed dataflow computing machines is the highly tuned, widely available and familiar von Neumann machines. The dataflow machines for numerical and nonnumerical computations such as VIM, DDP, LAU, NEDIPS, MIT TTDA, Manchester Dataflow Machine, DDM1, SIGMA-1, EM-3, EDDY, DFM, PIM-D, and TOPSTAR were discussed. The SIGMA-1, which predicts the highest computing speeds, is a milestone in computing machines. Some general problems in dataflow computing and performance evaluation measurements made in the EM-3 dataflow computing environment were presented.

A. Further Research

Considerable progress has been made in building dataflow machines during last few years. The construction of very high speed dataflow computing machines needs further research in the following areas.

1) Different dataflow computing models.

2) High-level languages suited to dataflow programming.

3) Low-level languages suited to dataflow architecture.

4) Algorithms for specific applications and systems.

5) Dataflow computing processors and system architectures.

6) Operating systems with efficient resource allocation schemes.

7) Optimum design of an instruction set processor.

8) High-speed operand matching mechanisms.

9) Efficient structure memory implementations.

10) Low cost high-speed communication networks to interconnect PE's.

11) Problems in dataflow computing.

12) Fault-tolerant computing for 1000 to 10 000 processor dataflow computing machines.

13) Impact of VLSI and device technology.

14) Applications of dataflow computing concept in other fields.

Acknowledgment

We wish to thank all the researchers in dataflow field for their efforts and contributions to realize dataflow machines. We would also like to thank Prof. M. Ishii and I. Shuichi of the University of Electro-Communications for their support in this research. We wish to acknowledge the invaluable support and discussions with Mr. Toda, Dr. Shimada, Dr. Hiraki, Dr. Furuya, Dr. Uchibori, and Mr. Nishida, the members of the computer architecture group. We would like to thank the members of the Saito lab at Keio University and appreciate the support extended by Dr. Rine, Dr. Wang, and colleagues at George Mason University. We thank the referees, Dr. B. Wah and Dr. C. V. Ramammoorthy for their helpful comments on an early version of this paper. Special thanks to S. Herath and R. Mattingley for helping in many ways with this work.

References

[1] J. E. Rodriguez, "A graph model for parallel computation," Tech. Rep. Lab. Comput. Sci., MIT, Tech. Rep. ESLR-398, MAC-TR-64, Sept. 1969.

[2] J. B. Dennis, "First version of a dataflow procedure language," in *Proc. Colloque sur la Programmation*, Vol. 19 (*Lecture Notes in Computer Science*). Berlin: Springer-Verlag, 1974, pp. 362-376.

[3] J. B. Dennis, G. R. Gao, and K. Todd, "Modeling the weather with a dataflow super computer," *IEEE Trans. Comput.*, vol. C-33, no. 7, pp. 592-603, July 1984.

[4] J. B. Dennis, "Data flow computation," in *NATO ASI Series, Vol. F14, Control Flow and Data Flow: Concepts of Distributed Programming*, M. Broy, Ed. Berlin: Springer-Verlag, 1985, pp. 346-397.

[5] J. D. Brock, "Operational Semantics of a data flow language," MIT, Tech. Rep. MIT/LCS/TM-120.

[6] M. Cornish, "The TI dataflow architectures: The power of concurrency for avionics," in *IEEE Proc. 3rd Conf. Digital Avionics Systems*, Fort Worth, TX, Nov. 1979, pp. 19-25.

[7] M. Cornish, D. W. Hogan, and J. C. Jensen, "The Texas Instruments distributed processor," in *Proc. Louisiana Computer Exposition*, Lafaytte, LA, Mar. 1979, pp. 189-193.

[8] Arvind and K. P. Gostelow, "Some relationships between asynchronous interpreters of a dataflow language," in *Proc. IFIP WG2.2 Conf. Formal Description of Programming Languages*, St. Andrews, Canada, 1977.

[9] —, "The U-interpreter," *Computer*, vol. 15, no. 2, pp. 42-49, Feb. 1982.

[10] P. Arvind, V. Kathail, and K. Pingaley, "A dataflow architecture with tagged tokens," Lab. Comput. Sci., MIT, Rep. TM-174, Sept. 1980.

[11] Arvind and D. E. Culler, "Dataflow architectures," Lab. Comput. Sci., MIT, Rep. TM-294, 1986.

[12] J. R. Gurd and I. Watson, "A multilayered dataflow computer architecture," in *Proc. 7th Int. Conf. Parallel Processing*, Aug. 1977.

[13] I. Watson and J. R. Gurd, "A prototype dataflow computer with token labelling," in *Proc. 1979 Nat. Computing Conf. AFIPS Proc.*, vol. 48, June 1979, pp. 623-628.

[14] J. R. Gurd, J. R. W. Glauert, and C. C. Kirkham, "Generation of dataflow graphical object code for the lapse programming language," in *Lecture Notes in Computer Science*, vol. 111. Berlin: Springer-Verlag, June 1981, pp. 155-168.

[15] J. Gurd and I. Watson, "Preliminary evaluation of a prototype dataflow computer," in *Proc. IFIP*, 1983, pp. 545-551.

[16] A. L. Davis, "The architecture and system method of DDM1: A re-

cursively structured data driven machine," in *Proc. 5th Annu. Symp. Computer Architecture*, Apr. 1978, pp. 210-215.

[17] A. L. Davis, "A dataflow evaluation system based on the concept of recursive locality," in *Proc. 1979 Nat. Computer Conf.*, vol. 48, AFIPS, 1979, pp. 1079-1086.

[18] E. A. Ashcroft and R. Jagannathan, "Operator nets," in *Proc. Fifth Gen. Comp. Arch.*, IFIP, 1984.

[19] E. A. Ashcroft and W. W. Wadge, "Lucid, a nonprocedural language with iteration," *Commun. ACM*, vol. 20, no. 7, pp. 519-526, July 1977.

[20] P. C. Treleaven, D. R. Brownbridge, R. P. Hopkins, and P. W. Rantenbach, "Combining dataflow and control flow computing," *Comput. J.*, vol. 25, no. 2, pp. 207-217, 1982.

[21] M. Amamiya, M. Takesue, R. Hasegawa, and M. Mikami, "Implementation and evaluation of a list processing-oriented dataflow machine," in *Proc. 13th Int. Symp. Comp. Arch.*, 1986, pp. 10-19.

[22] N. Takahashi and M. Amamiya, "A data flow processor array system—Design and analysis," in *Proc. 10th Annu. Int. Symp. Computer Architecture*, IEEE, 1983, pp. 243-250.

[23] M. Amamiya, R. Hasegawa, and S. Ono, "VALID: A high level functional language for dataflow machine," *Rev. ECL*, vol. 32, no. 5, pp. 793-802, 1984.

[24] M. Amamiya, R. Hasegawa, O. Nakamura, and H. Mikami, "A list processing oriented data flow machine architecture," in *Proc. Nat. Comput. Conf.*, AFIPS, 1982, pp. 143-151.

[25] M. Amamiya and R. Hasegawa, "Dataflow computing and eager and lazy evaluation," *J. New Gen. Comput.*, vol. 2, no. 8, pp. 105-129, 1984.

[26] S. Ono, N. Takahashi,and M. Amamiya, "Optimized demand-driven evaluation of functional programs on a dataflow machine," in *Proc. Int. Conf. Parallel Processing '86*, pp. 421-428.

[27] T. Yuba, Y. Yamaguchi, and T. Shimada, "A control mechanism of a Lisp based data-driven machine," *Inform. Processing Lett.*, vol. 16, pp. 139-143, 1983.

[28] Y. Yamaguchi, K. Toda, and T. Yuba, "A performance evaluation of a Lisp based data-driven machine (EM-3)," in *Proc. 10th Annu. Int. Symp. Comput. Arch.*, 1983, pp. 363-369.

[29] Y. Yamaguchi, K. Toda, J. Herath, and T. Yuba, "EM-3: A LISP-based data-driven machine," in *Proc. Int. Conf. Fifth Generation Comput Syst.*, ICOT, 1984, pp. 524-532.

[30] K. Toda, Y. Yamaguchi, Y. Uchibori, and T. Yuba, "Preliminary measurements of the ETL LISP-based data-driven machine," in *Proc. IFIP TC-10 Working Conf. Fifth Gen. Comput. Arch.*, July 1985.

[31] J. Herath, N. Saito, K. Toda, Y. Yamaguchi, and T. Yuba, "Not(operation) for high speed data-flow computing systems," in *Proc. Int. Conf. Super Computing Systems*, Dec. 1985, pp. 524-532.

[32] ——, "Data-flow computing base language with *n*-value logic," in *Proc. Fall Joint Comput. Conf.*, Nov. 1986.

[33] J. Herath, Y. Yamaguchi, T. Yuba, and N. Saito, "Extended not(operation) based dataflow computing for intelligent switching," in *Proc. Int. Conf. Supercomputing Systems*, May 1987.

[34] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *J. ACM*, Jan. 1965, pp. 23-41.

[35] R. A. Kowalski and M. H. van Emden, "The semantics of predicate logic as a programming language," *J. ACM*, Oct. 1976, pp. 733-742.

[36] A. Colmerauer, "Prolog and infinite trees," in *Logic Programming*. New York: Academic, 1982.

[37] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. New York: Springer-Verlag, 1981.

[38] K. L. Clark and S. Gregory, "A relational language for parallel programming," in *Proc. 1981 ACM Conf. Functional Programming and Computer Architecture*, Oct. 1981, pp. 171-178.

[39] ——, "PARLOG: Parallel programming in logic," *ACM Trans. Program. Lang. Syst.*, pp. 1-49, Jan. 1986.

[40] E. Y. Shapiro, "A Sub set of Concurrent Prolog and its interpreter," TR-003, ICOT, Tokyo, Tech. Rep. TR-003, Feb. 1983.

[41] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, pp. 613-641, Aug. 1978.

[42] ——, "Function level computing," *IEEE Spectrum*, pp. 22-27, Aug. 1982.

[43] J. McGraw, "SISAL: Streams and iteration in a single-assignment language reference manual," Univ. California, Lawrence Livermore Nat. Lab., Rep. M-146, Mar. 1985.

[44] S. Skedzielewski and J. Glauert, "IF1—An intermediate form for applicative languages," Univ. California, Lawrence Livermore Nat. Lab., Rep. M-170, July 1985.

[45] O. Gelly *et al.*, "LAU software system: A high level data driven language for parallel programming," in *Proc. 1976 Int. Conf. Parallel Processing*, Aug. 1976.

[46] D. Comte, N. Hifdi, J. Syre, "The data driven LAU multiprocessor system: Results and perspectives," in *Proc. IFIP Congress 80*, Tokyo, Oct. 1980, pp. 175-180.

[47] J. L. Gaudiot, M. Dubios, L. T. Lee, and N. Tohme, "The TX16: A highly programmable multimicroprocessor architecture," *IEEE Micro*, Oct. 1985.

[48] T. Shimada, K. Hiraki, and K. Nishida, "An architecture of a dataflow machine and its evaluation," in *Proc. COMPCON '84*, Spring 1984, pp. 486-490.

[49] K. Hiraki, K. Nishida, S. Sekiguchi, and T. Shimada, "Maintenance architecture and LSI implementation of a dataflow computer with a large number of processors," in *Proc. Int. Conf. Parallel Processing '86*.

[50] K. Hiraki, T. Shimada, and K. Nishida, "A hardware design of the SIGMA-1—A dataflow computer for scientific computations," in *Proc. 1984 Int. Conf. Parallel Processing*, IEEE, 1984, pp. 524-531.

[51] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi, "Evaluation of a prototype dataflow processor of the SIGMA-1 for scientific computations," in *Proc. 13th Annu. Int. Symp. Comput. Arch.*, 1986, pp. 226-234.

[52] T. Yuba, "Research and development efforts on data-flow computer architecture in Japan," *J. Inform. Processing*, vol. 9, no. 2, pp. 51-62, 1986.

[53] B. Wah and G.-J. Li, "Computers for artificial intelligence applications," *IEEE Tutorial*, 1986.

[54] V. P. Srini, "An architectural comparison of dataflow systems," *Computer*, pp. 68-88, Mar. 1986.

[55] P. C. Treleaven, D. Brownbridge, and R. P. Hopkins, "Data-driven and demand-driven computer architecture," *J. ACM Comput. Surveys*, vol. 14, no. 1, pp. 93-142, Mar. 1982.

[56] J. Herath, T. Yuba, and N. Saito, "Dataflow computing," in *Proc. Int. Workshop Parallel Algorithms and Architectures '87 (Lecture Notes in Computer Science)*. Berlin: Springer-Verlag, May 1987.

[57] M. Tokoro, J. R. Jagannathan, and H. Sunahara, "On the working set concept for dataflow machine," in *Proc. 10th Annu. Symp. Comput. Arch.*, June 1983, pp. 90-97.

[58] N. Ito and M. Sato, "The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D," in *Proc. 13th Int. Symp. Comput. Arch.*, 1986, pp. 149-156.

[59] T. Suzuki, K. Kurihara, H. Tanaka, and T. Moto-oka, "Procedure level dataflow processing on dynamic structure multimicroprocessors," *J. Inform. Processing*, vol. 5, no. 1, pp. 11-16, 1982.

[60] M. Kishi, H. Yasuhara, and Y. Kawamura, "DDDP: A distributed data-driven processor," in *Proc. 10th Annu. Int. Symp. Comput. Arch.*, IEEE, 1983, pp. 236-242.

[61] K. Oyama, N. Nguyen, V. P. Shrestha, T. Saito, and H. Inose, "System design of a distributed dataflow computer and its experimental evaluation," *Trans. Inform. Proc. Soc. Japan*, vol. 25, no. 1, pp. 101-108, 1984.

[62] H. Nishikawa, K. Asada, and H. Terada, "A decentralized controlled multiprocessor system based on the data-driven scheme," in *Proc. 3rd Int. Conf. Distributed Computing Systems*, 1982, pp. 639-644.

[63] M. Sowa and T. Murata, "A dataflow computer architecture with program and token memories," *IEEE Trans. Comput.*, vol. C-31, no. 9, pp. 820-824, 1982.

[64] T. Temma, S. Hasegawa, and S. Hanaki, "Dataflow processor for image processing," in *Proc. Mini and Microcomputers*, vol. 5, no. 3, pp. 52-56, 1980.

[65] A. L. Wendelborn, "A hybrid data and demand driven implementation of a lucid-like programming language," in *Proc. 9th Comput. Sci. Conf.*, Canberra, Jan. 1986.

Jayantha Herath (M'86) was born in Metigahatenna, Sri Lanka, on September 3, 1954. He received the B.Sc.(Hons) degree in electronics and telecommunication engineering from the University of Sri Lanka, Katubedda Campus, in 1978, the M.Eng. degree in electronics engineering from the University of Electrocommunications, Japan, in 1984, and the D.Eng. degree from Keio University, Japan, in 1987.

He worked as an electrical engineer in a steam power station at the Ceylon Electricity Board and

taught at the University of Sri Lanka, Katubedda Campus, from 1979 to 1981. In 1980, he was awarded a Japanese government scholarship and from 1981 to 1987 he worked as a Japanese government research student. From 1983 to 1984, he worked at ETL under the supervision of the EM-3 group. His research interests are high-speed parallel, distributed, and fault-tolerant computing systems, functional and logic progamming languages, and computer networks. Now he is working as an Assistant Professor with the Department of Computer Science at George Mason University, Fairfax, VA.

Yoshinori Yamaguchi was born in Oita, Japan, in 1949. He received the B.S. degree in electrical engineering from the University of Tokyo in 1972.

He is a Senior Researcher in the Computer Systems Division of the Electrotechnical Laboratory, MITI, Japan. He is the principal investigator of the EM-3 project. His current research interests include computer architecture, parallel computer systems and functional programming languages.

Mr. Yamaguchi is a member of the Information Processing Society of Japan and the Institute of Electronics and Communication Engineers.

Nobuo Saito (M'86) received the B.Eng., M.Eng., and D.Eng degrees from the Department of Mathematical Engineering and Instrumentation Physics at the University of Tokyo in 1964, 1966, and 1978, respectively.

He is a Professor of Mathematics at Keio University, Japan. During 1970-1971 he was a visiting scholar at the Department of Computer Science of Stanford University. He worked as a visiting computer scientist on the Gandalf project of the Department of Computer Science, Carnegie-Mellon University. His research interests include operating systems, parallel, and distributed processing systems, formal semantics of parallel programming, software engineering, software development environments, and application of knowledge engineering to software engineering.

Dr. Saito is a member of the ACM, the Japan Information Processing Society, the Institute of Electronics, Information and Communication Engineers of Japan, and the Japan Software Science Society.

Toshitsugu Yuba was born in Osaka, Japan, on September 22, 1941. He received the B.E. and M.E. degrees in electrical engineering from Kobe University in 1964 and 1966, respectively, and the Ph.D. degree in information science from the University of Tokyo in 1982.

He is currently the Chief of the Computer Architecture Section, Computer Systems Division, at the Electrotechnical Laboratory. He is the leader of the SIGMA-1 and EM-3 projects. In 1966 he joined the Nomura Research Institute. Since 1967, he has been with the Electrotechnical Laboratory of the Agency of Industrial Science and Technology of the Ministry of International Trade and Industry. His current research interests are computer architecture, parallel algorithms, and data structures and system software.

Dr. Yuba is a member of the Information Processing Society of Japan, the Institute of Electronics and Communication Engineers, and the Association for Computing Machinery.