stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 146/80 SEPTEMBER

A.H. VEEN

RECONCILING DATA FLOW MACHINES AND CONVENTIONAL LANGUAGES

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B99

ACM-Computing Reviews-category: 4.12, 4.20

RECONCILING DATA FLOW MACHINES AND CONVENTIONAL LANGUAGES*

by

Arthur H. Veen

ABSTRACT

This paper discusses the problems that arise when programs written in a conventional language are to be compiled into machine code for a data flow machine. It describes the current state of a project aimed at developing a compiler for an existing high level language for string processing. The compiler is intended to accept the full language without changing its semantics in any respect. The implementation of a sizeable subset of the language is described and solutions to the remaining problems are suggested. The results indicate that the gap between data flow machines and conventional languages is easier to bridge than previously assumed.

KEY WORDS & PHRASES: data flow computers, data flow languages, data flow analysis, code generation, parallel computing

---

*) This report is not for review; it is submitted for publication in the proceedings of CONPAR 81, Conference on Analysing Problem-Classes and Programming for Parallel Computing.

1. INTRODUCTION

Designs for data flow machines have been with us for half a decade. Some machines have been built while others are in various stages of development. A number of high level languages have been designed specifically for these machines. In this paper I will investigate whether such languages are indeed necessary, or whether conventional languages will do.

The original impetus for data flow research came from the need for a simpler and more reliable scheme to express concurrency in programs. The data flow schema's of Dennis [6] led to a design for a new type of parallel computer and since then the focus of the data flow field has been on both computer architecture and programming languages. Several higher level languages have been designed specifically for data flow machines. The argument given for this development is usually the radical difference in architecture between data flow machines and conventional machines.

There has been wide-spread pessimism in the data flow field concerning the suitability of existing conventional languages as source languages for data flow machines. The following quote from [15] illustrates this:

> "While it is possible that compile time analysis can be performed on sequential programs to produce an equivalent program of greater concurrency, this does not help programmers to express computations in a form which exhibits a high level of concurrency. Furthermore, no compile time analysis has been able to extract the inherent concurrency from a program containing unnecessary constraints which are the result of language features based on the assumption of sequential computer organization."

The first objection raised in this quote will be discussed in section 7. The second objection probably refers to static analyzers that attempt to break up a given program written in a language like Fortran into parts which can safely be executed concurrently. This task is complicated by features like goto statements, pointers, arrays, global variables, aliasing, and the like. In a data flow computer, however, part of determining which instructions can be executed in parallel is done by special hardware at run time. This simplifies the analysis that has to be performed at compile time and I will argue in this paper that it is quite feasible to use a conventional language as a source language for a data flow machine. The compiler that translates a conventional language into data flow machine code can be of the same order of complexity as a conventional compiler and can generate code that is as efficient as the code that can be generated from some of the special data flow languages that have been defined so far.

The advantages of using an existing language to program a data flow machine are quite obvious. It would greatly enhance the chances that these machines will be used as general purpose computers and that they will help to satisfy the ever growing need for computing power. For good reasons most people are very reluctant to purchase a machine on which none of their old software runs and which forces programmers to learn and use completely new languages. The number of general purpose programming languages which are used today is already staggering and new ones should not be added lightly.

In the next two sections I will give a short description of data flow machines and languages. Section 4 describes an (implemented) algorithm to translate a somewhat restricted conventional language to machine code for an existing data flow computer. Section 5 describes the steps that are necessary to extend the existing implementation to a compiler for a complete language. Section 6 summarizes the results obtained so far.

2. DATA FLOW MACHINES

A data flow computer is a type of parallel computer on which sequencing of computation steps is governed by the availability of data rather than by special synchronization or control flow instructions.

I will not describe the design or architecture of data flow computers or their advantages and disadvantages as compared to other parallel machines. Introductory material on these topics may be found in [10,12]. Instead I will concentrate on those characteristics that are interesting from a programmer's point of view.

- The computational steps that can be performed in parallel are small. These steps are the machine instructions which are typically of the same level of complexity as conventional machine instructions.

- A machine instruction is initiated when all its input data are available and then it runs to completion without interruption. The instructions are purely functional and free of side effects. This means that all the information used by an instruction is contained in its input data and that the only effect of its execution is the production of output data.

- An instruction contains pointers to all instructions which use its result. A data flow machine program is thus less compactly coded than the equivalent conventional program. This extra dependency information enables special hardware to sequence instruction execution very efficiently.

This last property of machine level programs implies that they can be represented as directed graphs. Because of this, the instructions and the data dependency pointers are often called nodes and data paths. The

values that travel from node to node over the data paths are called tokens. A node absorbs the tokens on its input data paths, performs a functional operation on it and produces the computed result as tokens on its output paths.

Data flow machines differ considerably in the way they handle reentrancy. Reentrant graphs are subgraphs corresponding to loops and (recursive) procedures, which are used more than once in a computation. Reentrant graphs can present problems because they contain cycles and therefore certain nodes will have an input point where tokens belonging to different activations may arrive simultaneously. There are three methods of avoiding confusion between these tokens. One is to surround the graph with guards which prevent new tokens from entering until all tokens of a previous activation have left the graph. This seriously limits the concurrency of the computation. The second method is to create a copy of a subgraph every time it is activated. The third method, which is equivalent to making a copy of the subgraph but is much cheaper in resources, is to color all tokens corresponding to the activations they belong to and to require that all nodes copy the color of their input tokens. A node is only activated when there are tokens of the same color on all its input paths. Special instructions are provided which manipulate the color of tokens.

The first data flow machines which were completed were the DDM1 [5] and the LAU machine [11]. The data flow group at MIT is in the process of building its first prototype [7]. The first machine using token coloring, is currently near completion at Manchester University [14].

Machine level programs in the form of instructions with pointers are hard to read. Therefore they are usually presented graphically. In this form, which will also be used throughout this paper, they are slightly more readable than conventional assembly language programs. There is however a clear need for a higher level language that can be translated into data flow machine code and such languages are the subject of the remainder of this paper.

3. DATA FLOW LANGUAGES

At least four high level data flow languages have been defined in conjunction with the design of a data flow machine. One of the oldest is LAU, from Toulouse, France [11]. The language LAPSE has been designed for the data flow machine at Manchester [8]. The data flow group at MIT has developed a language called VAL [1]. The group at Irvine has produced the language ID [3]. A comparison of some of these languages can be found in [2] and [13]. The differences between data flow languages on the one hand and conventional languages on the other reflect the differences in the underlying machines:

- In a data flow language statement delimiters and goto statements cannot be used to affect the sequencing of statement execution. The order of the statements in the program is generally irrelevant. This reflects the fact that, in a data flow machine, the execution sequence is governed by the availability of data and is in no way influenced by the order in which instructions are stored in the program memory.

- In a conventional language variables correspond to memory locations in the machine which can be accessed an arbitrary number of times. In a data flow language variables correspond to data paths in the machine code. As a consequence, a variable gets a value once, namely when a token appears on the corresponding data path. This is reflected in the Single Assignment Rule, which states that every variable appears exactly once at the left hand side of an assignment statement.

- In a data flow language procedures are purely functional. They have no intermediate memory through which information can be transferred from one procedure activation to another. There are no global variables. The single assignment rule implies that parameter passing is by value only. Recursive calling of procedures presents no problem.

- Arrays in a conventional language most closely resemble a dynamically addressable memory but they play a markedly different role in data flow languages. The single assignment rule forbids the assignment to arbitrary elements of an array. Data flow languages provide constructs like a forall statement which allow the programmer to specify the calculation of all elements of an array in one statement. The effect of changing a single element in an array can be achieved by creating a new array that is a copy of the old array with one element replaced by a new value.

4. USING A CONVENTIONAL LANGUAGE

In this section I will present an algorithm that translates a subset of a conventional language into data flow machine code. Algorithms to translate data flow programs into data flow machine code have been described in [4] and [8]. Whitelock was the first to implement a conventional-to-data flow compiler [16] and the development of the algorithm described in this section greatly benefited from his work.

The language accepted by the algorithm contains the following features: different data types (integer, real, string), sequencing by means of conventional statement delimiters, multiple assignment, loops, conditionals, case-statements, procedures, recursion and global variables. The language is in fact a subset of SUMMER [9], a locally designed language which is tailored towards string processing applications (such as compilers). The reasons for choosing SUMMER were

purely pragmatic. The features of SUMMER that were included in the subset have exactly the same semantics as in the original language and can be mixed and nested just as freely. The language is expression oriented in the sense that every construct delivers a value and can be used as part of other expressions. This stands in contrast to the language that Whitelock's compiler accepts, which includes, roughly spoken, the same features but in a much more restricted manner. The original SUMMER compiler, which is written in SUMMER itself, consists of a parser and a code generator. The data flow compiler described here, was obtained by using the almost unmodified parser and by writing a new code generator. Almost all of the algorithm described here concerns this new code generator.

The machine code I have chosen to generate is that of the Manchester machine, with a few additions which were needed to support string processing. This machine was chosen because its instruction set is clearly defined [8,16], and because it supports efficient interpretation schemes due to its color mechanism described in section 2. (The Manchester group uses the term tag rather than color) In the sequel the reader is assumed to be familiar with the general principles of a data flow computer but not with the details of this particular machine.

The code generator processes the parse tree representation of a program and produces the nodes and data paths of the equivalent data flow graph. In contrast to data flow languages the order of statements in a conventional language is important and the code generator makes explicit use of this order. However certain permutations of this order do not change the meaning of a program and these permutations will have no effect on the generated graph. This means that an important source of the overspecification of execution sequencing in conventional languages is removed by the translation process.

The code generator makes extensive use of objects called cocoons. The code for each expression is generated in the context of one or more nested cocoons, the innermost of which is called the current cocoon. The cocoons serve a dual purpose. First, they associate variables with nodes in the graph. Just like in a data flow language, there is a correspondence between variables and data paths in the generated graph. At any particular point during code generation a variable is associated with the node in the graph that is to produce the value the variable refers to. Because there is no single assignment rule these associations might change and the same variable can correspond to data paths in different parts of the graph. The second purpose of a cocoon is the generation of nodes in the graph that are needed for certain statements as an interface to the rest of the graph. Different statement types therefore lead to the creation of different varieties of cocoons, as I will now describe in more detail.

ASSIGNMENT

For the statement

a := b + c

only one new node with instruction code ADD is created. Data paths that are to carry the values "b" and "c" are constructed terminating at the new node. The origins of these data paths are obtained from the current cocoon. The assignment itself is not visible in the generated graph, but has the effect that in the current cocoon "a" is associated with the ADD node. This is illustrated in figure 1a. The nodes and data paths of the generated code are drawn in solid lines. The two unlabeled nodes are assumed to be part of the graph generated by previous statements. The new statement only adds one node and two data paths. The broken lines indicate compiler constructs like cocoons and their associations.

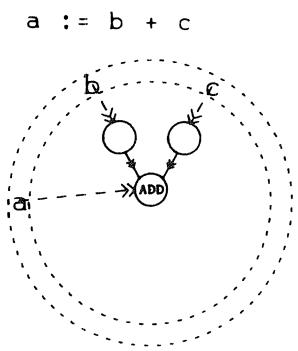

FIGURE 1a

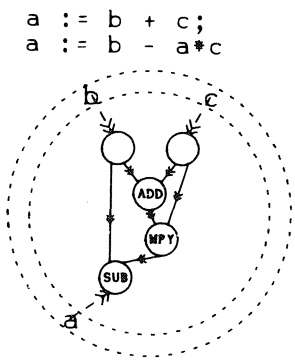

FIGURE 1b

In general, for a statement of the form

<identifier> := <expression>

a data flow tree is generated which closely resembles the parse tree of <expression> with the identifiers removed. Each logical or arithmetical operation of the source program is mapped onto one node or onto a small subgraph. Datapaths are then created which terminate at the new subtree and which originate at the nodes which are associated in the current cocoon with the identifiers in <expression>. The output node of the new subgraph is then associated with <identifier>. The resulting graph after processing the additional statement

a := b - a*c

can be seen in figure 1b, which also shows the effect of multiple assignment. The variable "a" is now associated with the SUBtract node,

while the ADD node is inaccessible to later statements: no more data paths originating at the ADD node can be created. In conventional terms one would say that the old value of "a" has been overwritten.

CONDITIONAL STATEMENTS

In a data flow machine the outcome of the evaluation of a condition does not affect the control flow but influences the flow of tokens along data paths. Each data flow machine provides for this purpose instructions which can switch the flow of tokens in different directions. The instruction set of the Manchester machine includes a BRAnch instruction which copies an incoming data token onto one of two outgoing data paths depending on the value of a boolean control input token. The Pass-If-True (PIT) and Pass-If-False (PIF) instructions are merely variations on the BRAnch instructions in which the data token is either copied or absorbed. These three instructions are called switches. For a statement of the form

if A then B else C fi

the graph corresponding to A is generated and linked to the appropriate nodes in the existing graph. The output node of this graph will produce a boolean value and is called the condition node. Then a new cocoon is created which is to generate all the switches that distribute input data to the two branches of the conditional. All switches have their control inputs connected to the condition node. The new cocoon will also collect those tokens that are produced in the two branches and are used later. Within this cocoon two additional cocoons are created in which the graphs corresponding to B and C are generated. The cocoons act as interfaces to the rest of the graph. The example in figure 2 illustrates some of the possibilities. The DUPlicate instructions simply copy the incoming data items to one or more output paths and are generally used in the interfaces. The purpose of the DUPlicate node labeled "a" in the outer cocoon, is to merge the data paths of the values "a" in the two branches. After the code generation for this statement the variable "a" is associated with this DUPlicate node. Through this mechanism a variable is, at any point during code generation, always associated with one node in the graph even if the data flow for that variable cannot statically be decided. Here we see an example of the division of labor between the compiler and the data flow machine: the compiler performs the data flow analysis as far as statically feasible and it generates the code that enables the machine to complete this analysis dynamically.
A conditional expression is translated in much the same way except that each of the graphs corresponding to the two branches contains a node which produces the value of that branch. These two nodes are linked to a DUPlicate node which represents the value of the whole expression. A conditional as the target of an assignment as in

if id<0 then neg else pos fi := id

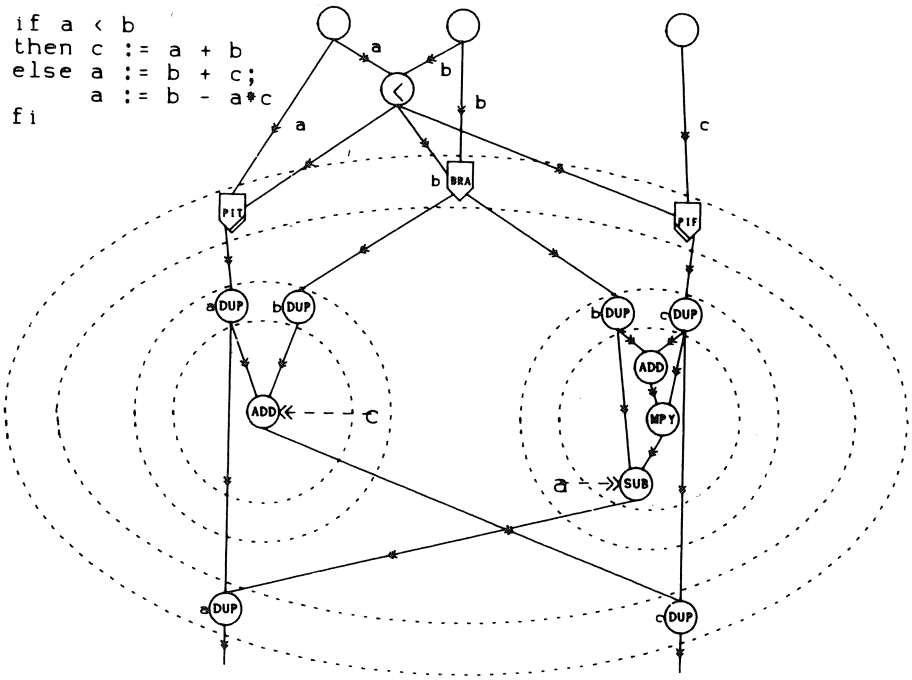

FIGURE 2

is also allowed. In this case a graph of switches is generated which distributes the value of "id" and the old values of "neg" and "pos" appropriately.
A case statement is converted into a series of tests, which can be evaluated in parallel, and a series of subgraphs corresponding to the branches. Each subgraph is surrounded by Pass-If-True switches controlled by the corresponding test. The parallel evaluation of the tests is possible because in SUMMER, of each case statement exactly one test evaluates to TRUE.

LOOP

The generated code for the iterative loop includes instructions to manipulate the color of tokens. Part of the color called the Iteration Level (IL) is used to separate the tokens of different iterations. Again nested cocoons are created which handle different parts of the interface process. In figure 3 we see the outer cocoon contains the switching nodes that distribute the tokens to the next iteration or to the continuation of the program. The other nodes in this cocoon are necessary for the correct manipulation of the color. This mechanism will not be described here. The right inner cocoon contains the test part and the left one the body of the loop. The latter creates the IIL nodes which Increment the Iteration Level between iterations. We see the values of "n" and "top" circulate around the loop. The tokens carrying

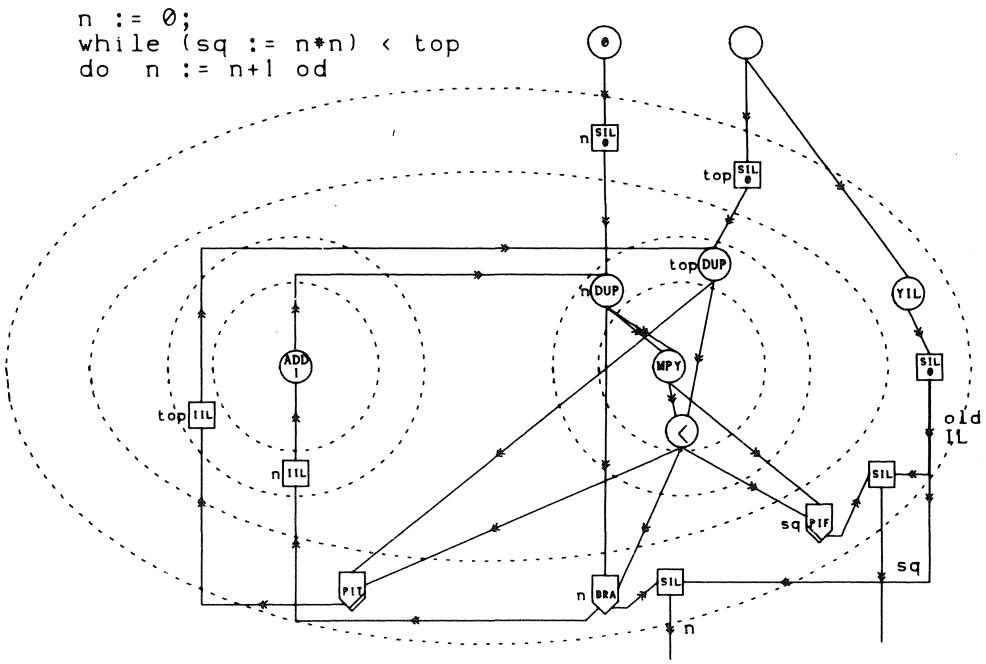

FIGURE 3

the successive values for "sq" are absorbed in the PIF node until the test evaluates to FALSE at which time the last token is ejected from the loop together with the last value for "n".

PROCEDURES

For a procedure call, nodes have to be generated that manipulate a part of the color called the Procedure Number. A special instruction causes the machine to generate a unique color (a non-functional operator!), which is distributed to all tokens that enter the procedure. These tokens are sent to the appropriate input points of the procedure. To the output points of the procedure tokens are sent which describe the nodes which are to receive the results computed by the procedure. In this way data paths are created dynamically and recursion is implemented without problems. Results produced by the procedure are given the old color of the invoking procedure. Global variables that are used in the procedure are treated as if they were additional input parameters. Global variables that are targets of assignments within the procedure are treated as if they were additional results. For this purpose the first pass of the compiler is slightly modified in such a way that the code generator knows of each procedure which of the global variables are needed as extra input and which ones are needed as extra output.

INPUT-OUTPUT

Communication with the outside world has intrinsic side-effects, which means that in general it does matter in what order input and output statements are executed. Therefore the basic I/O-routines need special consideration (in VAL [1] for instance, they are completely absent). In the current implementation the basic input and output routines increment a common global variable called an access token. This has the effect that all invocations of these routines are sequentialized because each invocation has to pass the access token to the next one. The possession of this token grants access to a non-functional module of the system. It is interesting to note that no additional measures have to be taken to provide sequential input-output; this can be solved completely within the framework of procedures with global side effects. A generalization of this concept is described in the next section.

OPTIMIZATION AND VERIFICATION

The representation of a program in the form of a data flow graph facilitates some useful optimization and program consistency checks. The code generator implemented so far performs checks on uninitialized or unused variables and superfluous assignments. It also carries out constant folding. If both inputs to an instruction are constants then the execution of the instruction is simulated and the instruction is replaced by the resulting constant. A warning is given when the controlling condition of a loop or conditional statement is folded into a constant. Carrying this constant folding to the extreme, would result in folding every program that does not contain an input statement, into one constant: the execution of the whole program would be simulated at compile time. In the current implementation, however, all instructions that manipulate colors are exempted from the constant folding, which has the effect that the compile time simulation stops at the boundaries of loops and procedure activations.
In SUMMER values rather than variables have types. This means that variables potentially refer to values of different types, which necessitates dynamic type checking. Most variables, however, always refer to values of the same type, and this type can usually quite easily be deducted from the data flow graph. This has the advantage that most of the type checking can be done at compile time.

5. EXTENDING THE SUBSET TO A FULL LANGUAGE

In this section I will describe how the subset described in the previous section can be extended to the complete language. Ultimately the compiler will have to be able to accept itself as source program. There are a number of features of SUMMER which have not been included in the subset. The implementation of some of them is quite trivial and they will be left undiscussed. The implementation of other features, however, touches upon some basic problems with data flow languages.

JUMPS

Almost all conventional languages provide facilities to specify sudden breaks in the control flow. The classical example is of course the goto statement. A more restrained form is the generation of an exception signal which transfers control to a specified point. Most languages include a return statement which has the effect that the execution of the current procedure is aborted and control is returned to the calling procedure. SUMMER has no goto statement but provides the other, more restrained forms of jumps.
The effect of a return statement can be simulated as follows: Each (compound) statement which contains a return statement delivers an additional boolean signal indicating whether the return has been activated or not. The subsequent parts of a statement sequence are surrounded by a new cocoon which controls the flow of incoming and outgoing tokens through a mechanism similar to that used in figure 2. The control input of the switches is in this case the return signal. This mechanism effectively postpones the execution of the remaining statements until the condition that controls the return statement has been evaluated.
The interruption of an expression evaluation due to the generation of an exception can be simulated by a more involved variation of the same mechanism.

ARRAYS

Arrays differ from simple data types in that one element can be changed while the rest remains the same. An attractive way of looking at such an operation, is to say that effectively a new array is created which differs from the old one at only one point. This is generally the way an assignment to an array element is treated in data flow languages. The programmer is forced to invent a new name every time she updates one element of an array. An equivalent and cheaper way to implement an assignment to an array element makes use of the access token mechanism described in the previous section. With each array an access token is associated, which is incremented by each access to that array. In this way all retrieves from and updates to a particular array are fully sequentialized. This can be optimized by letting retrieves from an array only wait for the previous update and letting each update wait for all retrieves since the previous update. With this optimization the same concurrency is obtained as when the algorithm was written in a data flow language.

OTHER DATA STRUCTURES

A central feature in SUMMER is the class mechanism which allows the programmer to specify abstract data types of arbitrary complexity. Access to class objects can also be controlled by access tokens. The same optimization as in the case with arrays is possible, if a distinction is made between retrieve and update accesses. All accesses

which change the internal state of the object in any way are update accesses and are synchronized accordingly. An access to an object can also make use of or change a global variable. These are treated as extra input and output in the same way as in the case of a normal procedure call.

6. CONCLUSIONS

The translation of a program written in a conventional language into code for a data flow machine is not as difficult a task as often assumed. An important reason for this may be that ambiguities in the data flow can be left unresolved at compile time since the target data flow machine is equipped with special hardware to complete this analysis at run time. A compiler for a quite extensive subset of a conventional language has been implemented. Implementation of the hereto excluded features seems feasible.

The algorithm described in section 4, to implement the subset is based on Whitelock's method, which in turn is based on methods generally used in data flow analysis. The chief characteristic of my implementation is the pervasive use of active data structures (classes). Each cocoon, constant or node in the graph is an active object with its own autonomous administration. The data flow graph is generated in response to the exchange of requests between these objects. These objects, of course, carry out essentially the same data flow analysis as Whitelock's compiler does, and in fact many conventional optimizers do. The implementation method chosen, however, seems to be far more flexible and amenable to generalizations and extensions than Whitelock's method. These properties have contributed greatly to the ease of implementation of the subset. This subset is expression oriented, which has as a consequence that side effects may arise in any part of an expression. This would greatly complicate a more conventional implementation scheme.

It is hard to give a quantitative measure for the complexity of a program, but it is indicative that, in my compiler, the code generator, where all the data flow analysis is concentrated, is smaller in size than the conventional parser. It is also hard to judge the quality of the produced code since there are almost no alternative implementations. Whitelock [16] compared his compiler with the one for LAPSE and reached the tentative conclusion that the code generated from his language was as good as or better than the code generated by the LAPSE compiler. The compiler described in this paper produces better code than Whitelock's because more attention is paid to optimization.

The literature gives three reasons for the development of special data flow languages:

- Data flow languages are safer in the sense that they make it easier to write correct programs since they exclude certain sources of common errors. This is an important reason for defining a new language, but much work in this area has already been done. It might be a more fruitful approach to start from a conventional language which has proven to be both safe and powerful and concentrate on adding facilities which would aid in program verification.

- Data flow languages coax a programmer into expressing his algorithm in a form which exhibits a high level of concurrency. Most features in the data flow languages developed so far have direct equivalents in most conventional languages. The only exceptions that might be of consequence are the forall construct and its derivatives. These constructs are tailored to efficient execution on a data flow machine and it remains to be seen whether a compiler will be able to easily recognize the equivalent statements in a conventional language. Similarly, certain language properties, like the fact that an array update can only be accomplished by specifying (and naming) a new array, might serve to discourage a programming style which would diminish the concurrency. Whether these features will indeed help to express algorithms more concurrently is still an open question which can only be answered when far more experience has been gained in the use of these languages by a diverse group of programmers. Only at that point could the concurrency contained in the resulting programs be contrasted with that of more conventional approaches.

- Without a data flow language it is difficult to construct a compiler that generates efficient code for a data flow machine. The work reported in this paper indicates that this is not a valid argument. When an appropriate source language is chosen and a sufficiently powerful implementation technique is used, then the complexity of such a compiler is quite manageable. Which properties a language should have in order to be suitable as a source language is still not completely clear. The absence of an unrestrained goto statement, which has as a consequence that a program can be transformed into a reducible flow graph, seems to be an important criterion.

REFERENCES

[1] Ackerman, W. and Jack B. Dennis, "VAL - A Value-Oriented Algorithmic Language Preliminary Reference Manual," Technical Report 218, MIT/Laboratory for Computer Science (June 1979).

[2] Ackerman, William B., "Data Flow Languages," Proceedings National Computing Conference Vol. 48, pp.1087-1095, AFIPS (Jun 1979).

[3] Arvind, Kim P. Gostelow, and Wil Plouffe , "An Asynchronous Programming Language and Computing Machine," Technical Report #114a, University of California, Irvine, Information and Computer Science Dept (Dec 1978).

[4] Brock, J. D. and Lynn B. Montz, "Translation and Optimization of Data Flow Programs," CSG Memo 181, MIT/Laboratory for Computer Science (July 1979).

[5] Davis, A. L., "A Data Flow Evaluation Sytem Based on the Concept of Recursive Locality," Proceedings National Computing Conference, pp.1079-1086, AFIP (Jun 1979).

[6] Dennis, J. B., J. B. Fosseen, and J. P. Linderman, "Data Flow Schemas," International Symposium on Theoretical Programming Vol. 5, pp.187-216 (1972).

[7] Dennis, Jack B., G. Andrew Boughton, and Clement K. C. Leung, "Building blocks for Data Flow Prototypes," Seventh Annual Symposium on Computer Architecture, pp.1-8 (May 1980).

[8] Glauert, J. R. W., "A Single Assignment Language for Data Flow Computing," Dissertation, Dept. of Computer Science - Victoria University of Manchester (Jan 1978).

[9] Klint, Paul, "An Overview of the SUMMER Programming Language," Conference Record of the Seventh Annual ACM Symposium of Programming Languages, pp.47-55, ACM (Jan 1980).

[10] Organick, E. I., "New Directions in Computer Systems Architecture," Euromicro Journal Vol. 5, pp.190-202 (1979).

[11] Plas, A. and others, "LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment," Proceedings of the 1976 International Conference on Parallel Processing, pp.293-302, IEEE (Aug 1976).

[12] Treleaven, P. C., "Principal Components of a Data Flow Computer," Large Scale Integration: Technology, Applications and Impacts, Fourth Euromicro Symposium on Microprocessing and Microprogramming, pp.366-374 (1978).

[13] Veen, Arthur, "Data Flow Computers," in Colloquium Hogere Programmeertalen en Computerarchitectuur, ed. P. Klint (1980). (in dutch)

[14] Watson, Ian and John Gurd, "A Prototype Data Flow Computer with Token Labelling," Proceedings National Computing Conference Vol. 48, pp.623-628, AFIPS (Jun 1979).

[15] Weng, Kung-Son, "An Abstract Implementation for a Generalized Data Flow Language," Technical Report 228, MIT/Laboratory for Computer Science (May 1979).

[16] Whitelock, P. J., "A Conventional Language for Data Flow Computing," dissertation, Dept. of Computer Science - Victoria University of Manchester (Oct 1978).