LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Maximum Pipelining of Array Operations on Static Data Flow Machine

Computation Structures Group Memo 233
September 1984

Jack B. Dennis
Gao Guang Rong

This research was supported by the Department of Energy under contract number DE-AC-02-7910473 and the National Science Foundation under grant number MCS-7915255.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Maximum Pipelining of Array Operations on Static Data Flow Machine

JACK B. DENNIS
GAO GUANG RONG

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
Sept. 1984

Abstract

Data flow computers are a radical departure from conventional computer architectures, and new methodologies are required for generating efficient machine-level programs from high-level user programming languages. In this paper we show that, for certain programs in the Val language, it is possible to construct machine-level data flow programs that support fully pipelined computation. A Val program in the class considered consists of blocks of code each of which defines a new array value either by a **forall** expression in which each element may be computed independently, or by a **for-iter** expression that defines array elements by a first-order recurrence relation.

1. Introduction

Research on the structure and programming of highly parallel computer architectures based on data flow principles has shown data flow architecture to be a promising approach for future generations of high performance machines. Because data flow computers are a radical departure from conventional architectures, new concepts and methodologies are required for translating high-level programs into efficient machine level program structures.

A major part of large-scale scientific computation is the construction of arrays of numerical values from arrays of values computed by other portions of the algorithm. In this paper we concentrate on the principal program structures used to express these computations in the programming language Val, a user functional programming language designed for expressing computations to be executed by computers capable of highly

---

1. This research was supported by the Department of Energy under grant number DE-AC02-79ER10473 and the National Science Foundation under grant number MCS-7915255.

concurrent operation. The two program structures studied are the **forall** expression which expresses the independent computation of all elements of an array; and the **for-iter** expression, which specifies successive array elements by a recurrence relation.

We begin in Section 2 with a brief review of the architecture proposed for high performance data flow computers : the static data flow architectures. In section 3 we introduce the concept of pipelining as it is embodied in the operation of machine level data flow programs, and point out how the generation and consumption of array values lends itself naturally to pipelined implementation on a data flow processor.

In Section 4 we introduce the class of Val programs considered which consist of blocks of code each of which defines a new array value either by a **forall** expression or a **for-iter** expression. The program blocks form an acyclic directed graph by virtue of their roles as producers and consumers of array values. The main loops of several benchmark programs we have studied have this general form and we are hopeful that our methods can be extended to a significantly broader class of programs.

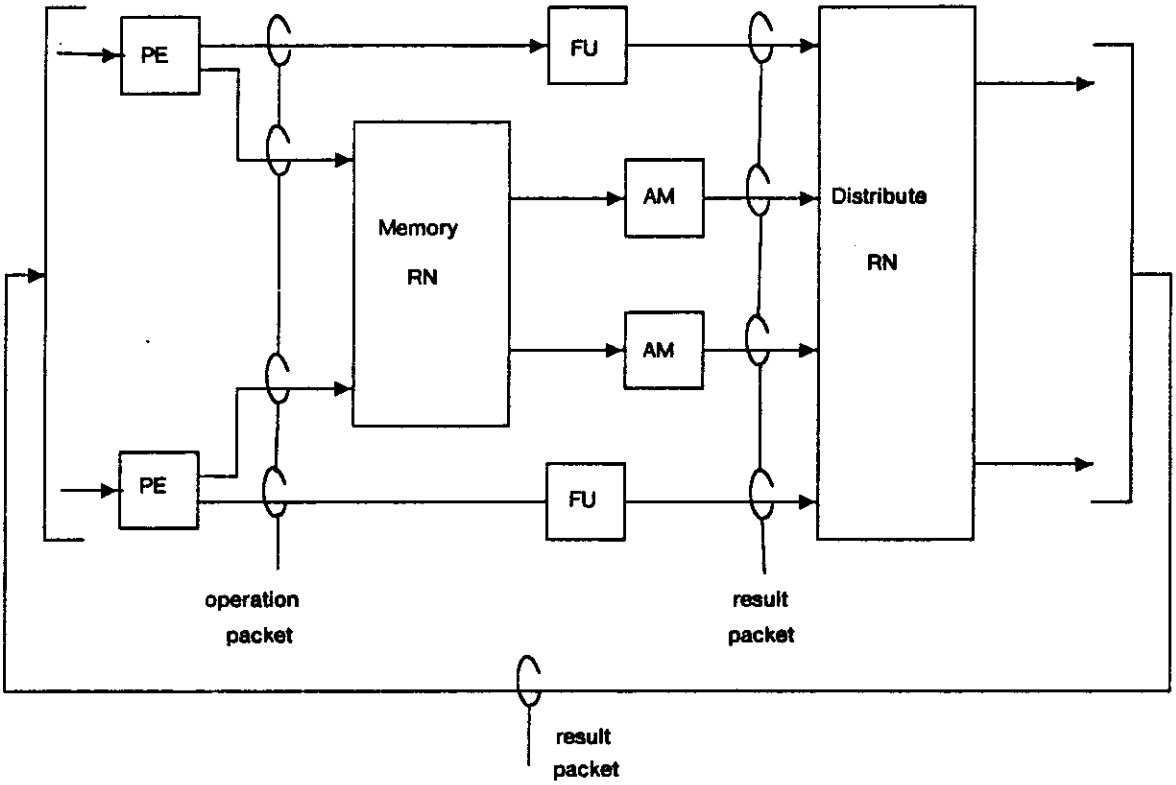
In Section 5 we show how primitive expressions—the expressions allowed as components of **forall** and **for-iter** program blocks—may be implemented as fully pipelined data flow machine code. Then in Sections 6 and 7 we show how this construction may be used to create fully pipelined code structures for certain **forall** blocks and **for-iter** blocks. In Section 8 we point out that these results imply that a useful class of *pipe-structured* Val programs may be automatically translated into data flow machine code capable of fully pipelined operation.

Section 9 concludes with a summary and suggestions for further research.

2. An Overview of The Static Data Flow Supercomputer

A data flow computer is a form of stored program computer in which instructions are activated by the data on which they operate. The organization of data flow computer that appears most attractive to us for high performance computation is the static data flow supercomputer described by Dennis [3] which has evolved from the architecture originally presented by Dennis and Misunas [4]. This kind of data flow computer, as shown in Figure 1, is composed of four types of units: *processing elements* (PE), *function units* (FU), *array memories* (AM), and *routing networks* (RN). The whole system is a packet communication architecture [5] using two kinds of packets: *operation packets* that represent instructions ready for execution, and *result packets* that contain result values destined to become operand values of target instructions.

The memory in the processing elements is divided into *instruction cells* which hold the instructions of a data flow program. Each instruction cell has fields for the operation code, the operand values, and the destinations of the instruction it holds [16][6][7].



PE : processing element
AM : array memory
FU : functional unit
RN : routing network

Figure 1. A Static Data FLow Architecture

Once the operand values of an instruction are in place, (and certain additional conditions are satisfied), the instruction is *enabled.* An enabled instruction may be executed within its processing element, sent to a function units if, for example, a floating point arithmetic operation is called for, or to any of the array memory units if the instruction calls for building or accessing an array element.

Instruction execution in a functional unit or array memory unit yields result packets each of which consists of a data value and a destination field that specifies the target instruction for the result packet. It is suggested that the routing networks be built as packet switched networks so the necessary throughput capacity

may be obtained at low cost [2].

A machine level data flow program, regarded as a collection of instruction cells, is essentially a directed graph, with nodes corresponding to instructions and an arc for each instruction destination field. We will use such diagrams to present data flow machine code structures in the remainder of this paper.

We call this sort of computer a *static* data flow architecture because the instructions of machine level programs are loaded into specific memory locations in the machine before computation begins, and at most one instance of each instruction is active at any time.

The rest of the paper addresses the issue of how a certain class of Val programs composed of blocks that consume and produce array values can be translated into machine level programs for efficient operation on static data flow supercomputers. We will see that these array values are generally *not* stored in the array memory units. Rather, it is better to treat arrays as sequences of values that are transmitted in succession from one instruction to another. The array memories are used only for data that must be held for a long time interval before being consumed by further computational blocks, for example, the data produced by one time step of a physics simulation which will not be used until the computation for the next time step begins. In this way, the primary packet traffic in the data flow machine is the flow of result packets between processing elements through the distribution network. In the case of application codes we have analyzed, one eighth or less of the operation packets would be sent to the array memories.

3. Pipelined Execution of Data Flow Programs

One approach to achieving high levels of concurrent operation in a computer system is by parallelism – replicating a function many times. High performance is attained by arranging for all copies of the function to be executed independently by separate parts of the computer system. Another approach is *pipelining* – splitting the function to be performed into a cascade of simpler functions or *stages*. The system is arranged so each stage is executed independently. Each stage is kept busy processing units of data, one after another, as they flow through successive stages of the "pipeline." A nice feature is that the computation rate of a pipeline is not dependent on the number of stages, but is determined by the processing rate of the slowest stage.

Pipelined execution of computations is very natural on the static data flow computer. Each stage of a pipeline is a group of instruction cells that act on their operands concurrently. The connections between stages are the destination fields that direct result packets from one instruction to its successors. Figure 2 illustrates this for the following code fragment.

```
let y: real := a * b
in ( y + 2. )*( y - 3. )
```

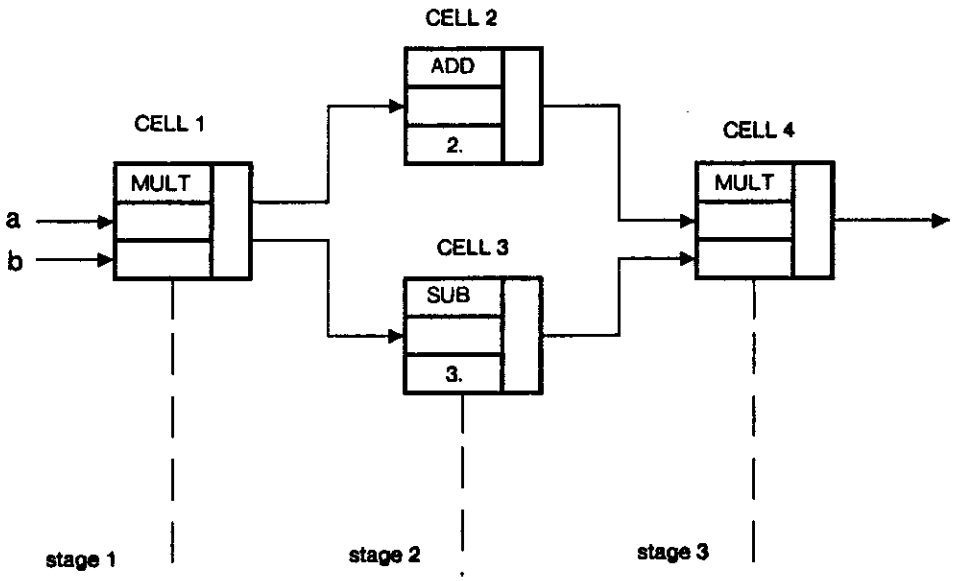


Figure 2. An Example of Pipelined Execution of Data FLow Program

endlet

When values of a and b arrive, cell 1 fires and sends results to cells 2 and 3. Once cells 2 and 3 fire and acknowledge receipt of their operands, cell 4 may fire, and cell 1 may fire again on new data. Thus data may flow continuously through the three-stage pipe made up of the four cells. To ensure proper coordination of data flow, it is necessary to arrange for each instruction to signal its predecessor indicating readiness to receive a new packet. This is done by providing in the instruction format for specifying destinations to which acknowledge packets must be sent. The mechanisms for doing this have been described [16][6][7].

For simplicity, in drawing data flow machine programs we will use a single arc to indicate both the forward path taken by a result value, and the reverse path taken by the acknowledge packet.

The computation rate of this pipeline is no greater than the rate of repeated execution of any instruction in the pipe. This rate is determined by the time from the moment of enabling to the time acknowledge packets are received from all successor instructions – about two instruction times for the data flow machine. If a data flow machine code structure is able to support pipelined computation at this rate, we say that the code structure is *fully pipelined* (or *maximally pipelined*). For an instruction graph to be fully pipelined, it is necessary that each path through the graph pass through exactly the same number of instruction cells. For programs generated from simple unconditional expressions, such as the above example, the pipeline may be

balanced by inserting identity operators [14]. Other program structures require more elaborate treatment and are discussed in later sections.

This form of pipelined computation is like the "systolic" computation of H. T. Kung [13] except that the set-up of the pipeline is represented here by the stored data flow program instead of by fixed hardware structures.

The power of pipelined computation in the data flow computer lies in the possibility of machine level programs that form one very large pipeline in which thousands of instructions in hundreds of stages are in concurrent execution.

To apply pipelined processing to computations involving array values requires a radical change of view regarding the role of arrays. For the purposes of the present study, for example, we regard an array as simply a sequence of values passed in succession from one block of code (*producer*) to another (the *consumer*).

4. Array Operation Constructs in Val Programs

Two constructs in the Val programming language are of major importance in expressing scientific computations.

A **forall** expression can be used to express the construction of an array where each element of the array is specified by the same computational rule, and all elements may be computed independently. The following is an example of a Val **forall** expression which defines an array C.

Example 1. A Val forall Construct

```
A : array[real] : =
  forall      i in [0, m+1]      % range specification
     P : real : =                % definition part
        if (i = 0)|(i = m+1) then C[i]
        else
           0.25 * (C[i-1] + 2.*C[i] + C[i+1])
        endif;
  construct
     B[i] * (P * P)              % accumulation
  endall
```

This **forall** expression takes as input two arrays B and C, and constructs a new array A. The index range of the constructed array is specified by the *range specification*, which gives the set of index values taken on by identifier i. The *definition part* gives values for several local variable names (only P in this example). The

definitions are executed once for each value of i. The *accumulation* part is also evaluated independently for each array index value; each evaluation yields the value of the corresponding element of the constructed array.

The **for-iter** expression in Val is the construct used to express iteration – the computation of sequences of values in which the value produced in one cycle depends on the value or partial results produced by the proceeding cycle.

The following is an example of a **for-iter** loop which constructs an array X.

Example 2. A Val for-iter Construct

```
X : array [real] : =
   for
      i : integer : = 1;                    % loop initialization
      T : array[real] : = [0: 0.]
   do
   let P : real : = A[i]*T[i-1]+B[i]        % definition part
   in
      if i ≤ m then                         % loop body
                  iter
                  T : = D[I : P]
                  i : = i+1
                  enditer
      else T
      endif
   endlet
   endfor
```

The loop *loop initialization* part binds initial values to each of the loop names (i and T in this case). In each cycle of the iteration, the *definitions part* are executed and then the conditional expression that makes up the *loop body* is evaluated. If the chosen arm of the conditional is an iter clause, the loop names are bound to new values and evaluation of the body is repeated. Otherwise the iteration terminates with the value of the conditional arm expression as its result. In this example, two input arrays A and B are consumed, and the array T accumulates the elements of the result array which is named X outside.

The Val programs of interest in this paper are those made up of program blocks, each of which is a **forall** or a **for-iter** block. Each block may be thought of as a 'producer' of an array value, and a 'consumer' of other array values produced by other blocks. This simple structure matches the main body of many practical programs of computational physics. If each such producer-consumer pair can handle array in a pipelined

fashion and appropriate buffers are included to implement proper interconnection between them, we may achieve high throughput for the whole computation on a data flow machine.

Definition A *pipe-structured* program is a Val program in which all array constructions are defined by non-nested blocks such that: (1) each block is either a **forall** block or a **for-iter** block, (2) the index ranges of the arrays generated by those blocks are fixed.

Due to the applicative nature of the Val language, the overall structure of a pipe-structured program can be described by an acyclic directed graph (called the *flow dependency graph* [8]) in which each node denotes a **forall** or **for-iter** block and each arc represents a link over which the elements of an array value flow from a producer block to a consumer block.

The combination of the two example program blocks forms a simple pipe-structured program, as shown in Figure 3. In a substantial application code the number of blocks in a pipe-structured program may amount to several hundred blocks

Pipe-structured programs are attractive candidates for implementation as fully pipelined machine code structures for data flow computers. However, obtaining fully pipelined machine code may not be possible for all programs in this class. Thus to obtain a class of programs for which fully pipelined implementations may be constructed, it is necessary to further restrict the structure of programs. This we will do in subsequent sections as we consider the **forall** and **for-iter** constructs and the simpler expressions of which they are built.

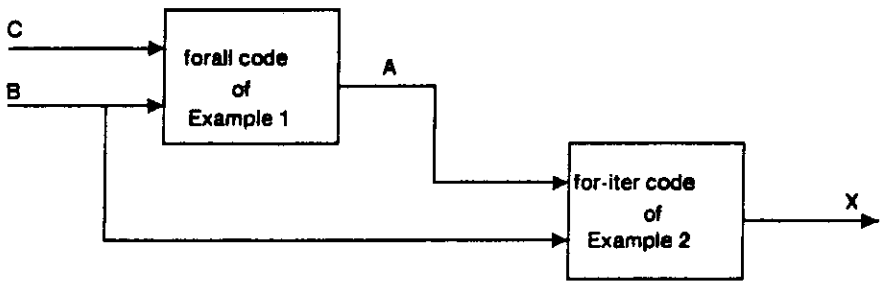


Figure 3. An Example of A Pipe-Structured Program

5. Pipelined Mapping of Primitive Expressions

We next introduce the pipelined implementation of a restricted class of Val expressions. These will be used later to define classes of **forall** and **for-iter** expressions that can be translated into fully pipelined machine code. The restricted class of expressions contains no nested **forall** or **for-iter** expressions, and no array constructor operations. We consider two kinds of primitive expressions—those that contain references to array elements selected by an index variable and those that do not.

Definition Let i be an identifier called an index variable. Then a *primitive expression* (PE) on i is any Val expression which may be constructed using only the following rules:

(1) A scalar literal constant is a PE.

(2) An identifier of a scalar value is a PE.

(3) If *E1* and *E2* are PEs, then (*E1* op *E2*) is a PE, where op is an arithmetic or relational operator.

(4) If A is an identifier that denotes an array, then A[i+m] is a PE, where m is an integer constant.

(5) Let *E* be a Val **let-in** construct expressed as Let ⟨*definition*⟩ **in** *E0* **endlet**. If ⟨*definition*⟩, the definition part, contains only PEs and *E0* is also a PE, then *E* is a PE.

(6) If *E1*, *E2*, *E3* are PEs, then **if** *E1* **then** *E2* **else** *E3* **endif** is a PE.

If rule (4) is not used in the construction, the resulting expression is a *scalar primitive expression.*

If a primitive expression is formed using only rules (1), (2), (3), and (5), its implementation as an acyclic data flow instruction graph is straightforward, and the methods developed by Montz[14] and by Gao[8] may be used to balance the instruction graph so it supports fully pipelined computation.

Next we show that PEs containing array access operations (rule (4)) may be fully pipelined. For this we assume that each of the arrays being accessed arrives as a sequence of result packets sent to a particular instruction cell; the first result value to arrive is for the least element of the array's index range. Two matters must be addressed to make pipelined operation work out correctly: (1) The elements of the incoming array not used in the computation must be discarded so they do not cause jams; and (2) Buffering must be inserted to introduce any skew needed to balance the pipeline.

As an example, consider the expression

$$0.25 * (C[i-1] + 2. * C[i] + C[i+1])$$

from the body of Example 1 of Section 4. The result required is the sequence of values of this expression for values of i in the set [1,..., m].

The corresponding fully pipelined instruction graph is shown in Figure 4. Here we suppose the array C is represented by m+2 result packets for the index set [0,..., m+1]. The boolean control sequences select just those array elements needed for the computation. The two FIFOs balance the pipeline by holding values of array elements between their arrival at the identity instructions and when they must enter the arithmetic pipeline. Other examples are given in [6][7]; we hope the generalization is clear.

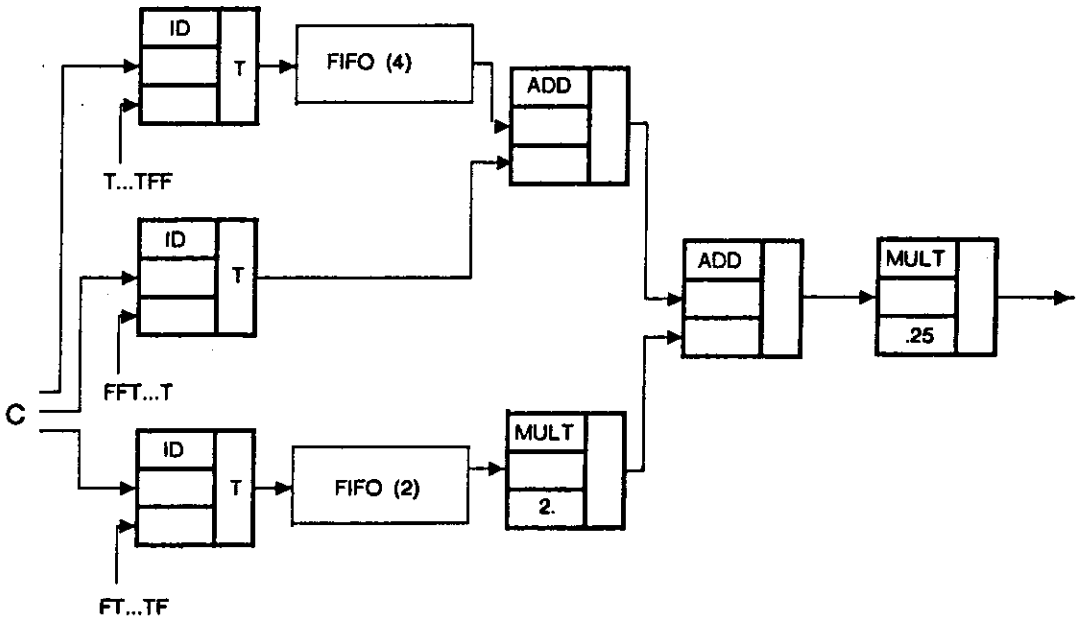


Figure 4. Pipelined Mapping for Array Selection Operations

The final case is that of conditional expressions. The general technique is illustrated in Fig. 6 for the following example:

```
if C[i]
then
      (A[i]+B[i])
else
      5.*(A[i]*B[i]+2.)
endif
```

This instruction graph makes use of instruction cells (identity operations in this case) in which a boolean operand directs a result packet to destinations according to a tag (T or F) on the destination arc. The merge instruction deserves further mention: it has two data inputs (I1 and I2) a merge control M. If a true value is present at M and a data value is present at I1, the instruction fires and the value on I1 is sent forward as the result packet, leaving the second operand on I2, if any, untouched. Conversely, if the merge control value M is false, the second operand value, if present, will be used and gated to the result arc. Thus the merge control operand directs the merge instruction to forward one or the other of its data operands.

Note that since data may be switched back and forth between the subgraphs corresponding to the two

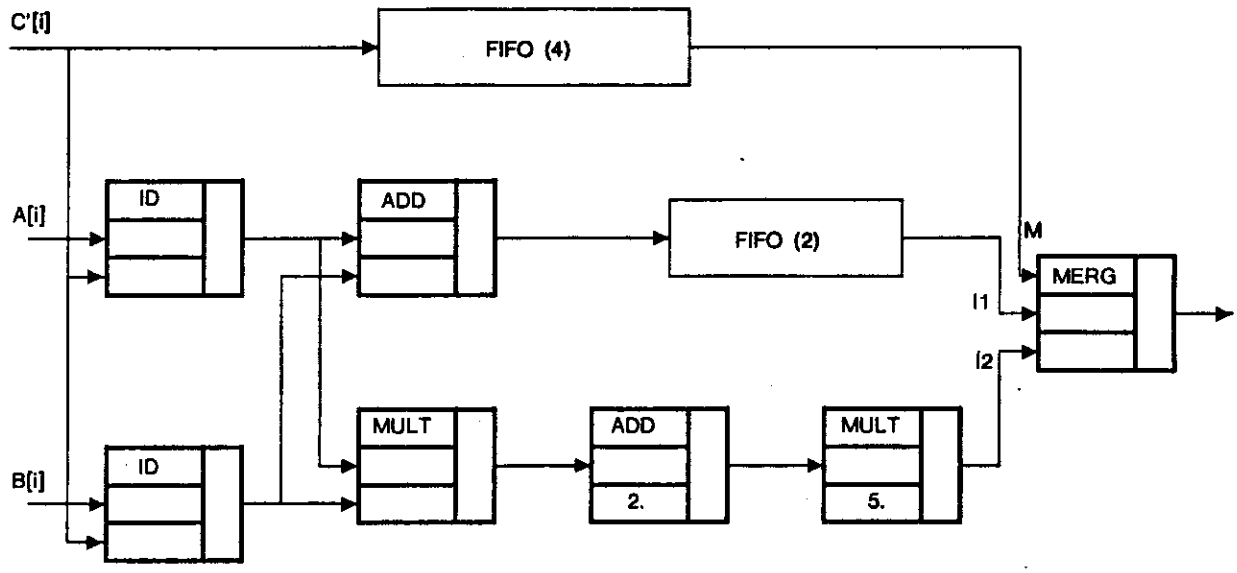


Figure 5. Fully Pipelined Instruction Graph for an if-then-else expression

arms of the conditional, fully pipelined operation is guaranteed only if all paths through the instruction graph are of equal length; it may be necessary to add FIFO buffering to one subgraph to match the length of the other. Furthermore, the path over which control values flow to the merge instruction cell must include a FIFO of correct length.

This discussion and examples lead us to the following theorem which provides a basis for the constructions presented in the next two sections.

Theorem 1 For any primitive expression, a fully pipelined data flow instruction graph can be constructed.

6. Pipelined Mapping of forall Constructs

Two general schemes for mapping primitive **forall** expressions into data flow instruction graphs that can exploit the potential concurrency of the static data flow architecture: the *parallel scheme* and the *pipeline scheme.* In the parallel scheme, a separate copy of the **forall** body expression is used for each element of the constructed array. In this scheme, all copies of the **forall** body may execute concurrently. This scheme is of limited interest in the present discussion, as it does not take advantage of our choice to implement array values as the sequence of result packets sent to an instruction cell.

In the *pipeline scheme* the array elements are generated in sequence by implementing the body of the **forall** construct as a pipelined instruction graph. A class of **forall** expressions for which we can construct fully pipelined instruction graphs is specified by the next definition:

Definition. A *primitive* **forall** expression is a **forall** expression in which: (1) The index range is specified as [p,q] where p and q are integer constants. (2) The right hand side of the definitions and the expression in the accumulation part are all primitive expressions in i, where i is the index variable of the **forall** expression.

The fully pipelined implementation of a primitive **forall** expression (Example 1 from Section 4) is shown in Figure 6. It is essentially the instruction graph obtained by cascading the instruction graphs for the definition expression and the accumulation expression. We suppose the input arrays B and C are fed to the instruction graph element by element for the index set [0,..., m+1]. The identity instructions select from the input arrays those elements needed for the computation, and the merge instruction combines results computed by different rules into the sequence of values that represent the constructed array. The sequences of boolean control values can be generated by straightforward arrangements of data flow instructions, as have been developed by Todd[15].

The example and the generalization of its construction show that all primitive **forall** expressions can be effectively pipelined.

Theorem 2 For any *primitive* **forall** expression, a corresponding fully pipelined data flow instruction graph can be constructed.

Further details of this implementation scheme can be found in [8].

7. Pipelined Mapping of for-iter Construct

In this section, we study the pipelined mapping of Val **for-iter** array operation construct.

To define a class of **for-iter** constructs which can be successfully mapped, we restrict our attention to those built on primitive expressions.

Definition A *primitive* **for-iter** construct is a **for-iter** expression with two loop variables – let them be i and X – such that: (1) Loop variable i takes on successive integer values p, p+1,..., q for successive

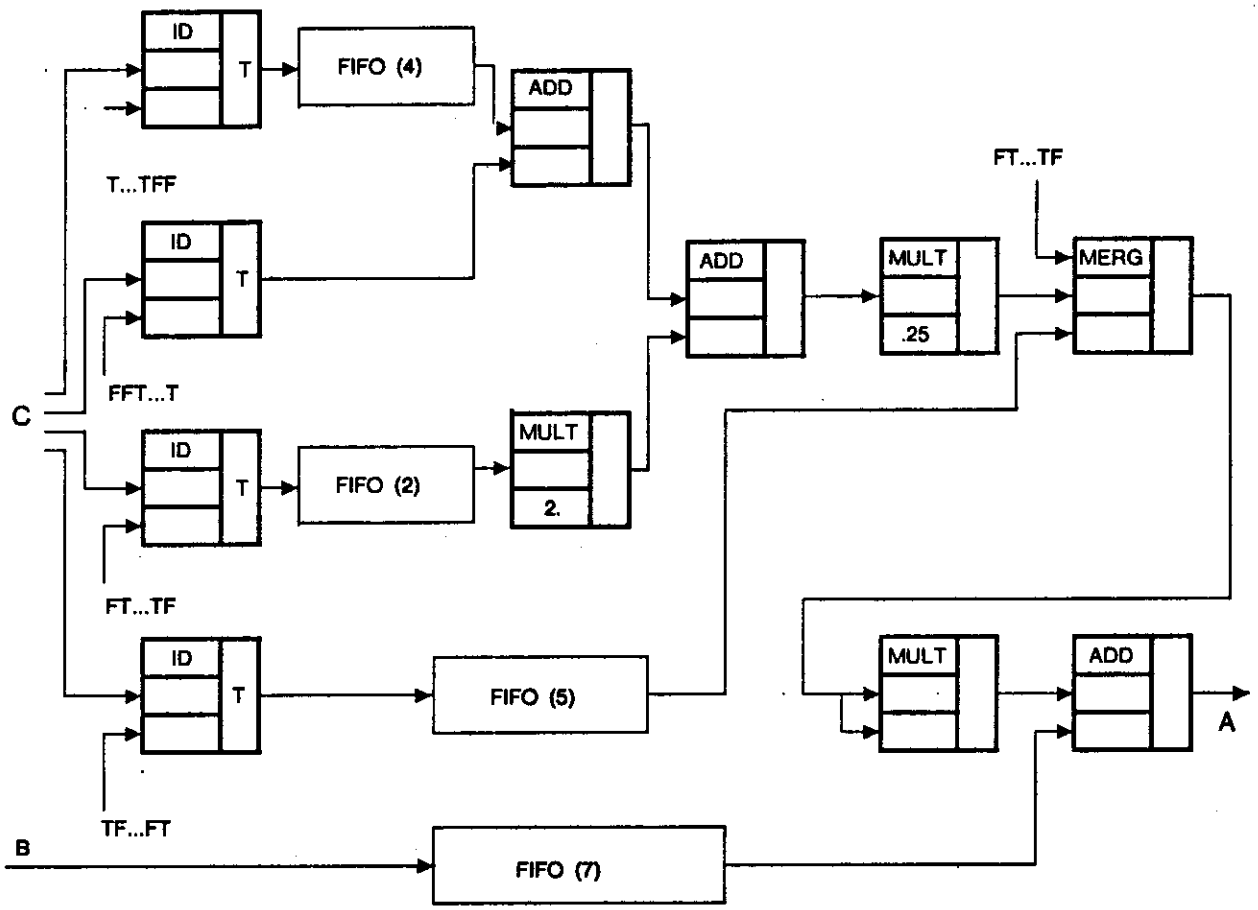


Figure 6. Pipelined Mapping of A Primitive forall Expression

evaluations of the **for-iter** body, and the loop terminates after i = q. (2) The loop variable X is initialized by X := [r: E] for some integer r and some primitive scalar expression E. Each iteration appends to the array by X := X [i: E] for some primitive expression on i. (3) The result expression on loop termination is X which will be the array constructed by the **for-iter** expression.

This definition restricts the structure of both the definition part and the conditional expression that makes up the body of the **for-iter** expression. Obviously, Example 2 given in section 4 is a primitive **for-iter** construct.

Several researchers have studied the translation of **for-iter** constructs into pipelined instruction graphs.

A scheme of mapping **for-iter** constructs has been developed by Todd [15]. Figure 7 shows the implementation of Example 2 using Todd's scheme. To understand how the scheme works we need explain the role of the merge instruction. The results of the merge are sent by the merge to two destinations. The first (the lower one in Figure 7) is forwarded as the output of the loop unconditionally. The second (the upper one in Figure 7) is fed back under the output switch control values, i.e. <T...TF>. It is obvious that the last element of array will be forwarded to the output arc without being fed back.

Due to the existence of cycles, the instruction graph produced by such scheme, in general, can not be fully pipelined. More specifically, the feedback link between the input of cell 1 and the output of cell 3 prevents the whole graph from being fully pipelined. This is because the value of X[i] depends on the value of X[i-1]. As a result, the calculation of X[i] can not start until the computation of X[i-1] is finished and the value of X[i-1] is available at the input of cell 1. Since there are 3 stages between the input and the output, the initialization rate of the pipeline can not be higher than 1/3. The difficulty we meet here is that the balancing algorithm developed in [14] does not apply to the case where the graph is not acyclic. To further classify the problems we restate them in a more formal context.

The most common problems involving **for-iter** array operations are recurrences, which define the sequence of elements of the output array X in term of a sequence of input values a_1, a_2...a_n as follow :

$$x_i = F(a_i, x_{i-1}) \qquad (1)$$

Mathematically, we have the following definition.

Definition : The function F in (1) is called a *first order recurrence function*, and a_i is called *parameter vector.*

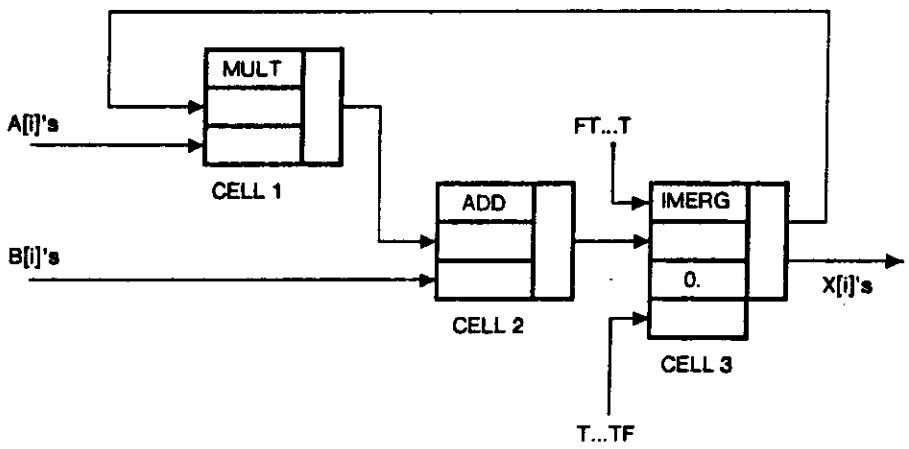


Figure 7. A Translation Scheme of for-iter Constructs

The recurrence function can be expressed using Val **for-iter** constructs. The **for-iter** code in Example 2 shows the general form of a first order recurrence function expressed in Val. In fact, it is exactly the Val code for the following mathematical notation.

$$\begin{aligned} x_i &= A_i x_{i-1} + B_i \\ &= \mathbf{a}_i(1)x_{i-1} + \mathbf{a}_i(2) \\ &= F(\mathbf{a}_i, x_{i-1}) \end{aligned} \qquad (2)$$

where the parameter vector $\mathbf{a}_i$ is the ordered pair (A_i, B_i), and the function F is an add and a multiply. The parameter vector of this example has length 2. Note we chose to use a boldface letter to denote the parameter vector a, and a(i) will denote its i^{th} element for it. Similar notation will be used through the rest of this section.

A pipeline to solve the recurrence problem would ideally accept each new input $\mathbf{a}_i$ and feed it into the pipeline as soon as it is available. Many authors have studied this problem, yet no general scheme is known that works for an arbitrary recurrence relation. In this section, we suggest a solution for a certain class of recurrence relations which has been first proposed [11][12] for conventional computers. We will apply this approach to the construction of fully pipelined data flow machine code.

The key observation is that, the recurrence function F has a *companion function* which makes an elegant solution possible. Let us first introduce the concept of companion function.

Definition If F is a first order recurrence function, and has the property that there exists some function G such that for all parameter vectors **a**, **b**, and all x of the proper domain, the following holds :

$$F(\mathbf{a},F(\mathbf{b},x)) = F(G(\mathbf{a},\mathbf{b}),x)$$

then the function G is termed a *companion function* for F.

Now let us consider the example (2) which is a first order recurrence relation, and show how to apply the above scheme to it. If we assume the function F has an execution delay of 2 ($p=2$), then it can be transformed as follows :

$$\begin{aligned} x_i &= A_i x_{i-1} + B_i \\ &= \mathbf{a}_i(1)x_{i-1} + \mathbf{a}_i(2) \\ &= \mathbf{a}_i(1)[\mathbf{a}_{i-1}(1)x_{i-2} + \mathbf{a}_{i-1}(2)] + \mathbf{a}_i(2) \\ &= \mathbf{a}_i(1)\mathbf{a}_{i-1}(1)x_{i-2} + (\mathbf{a}_i(1)\mathbf{a}_{i-1}(2) + \mathbf{a}_i(2)) \end{aligned}$$

We can see that the function F of (2) has companion function G, where

$$G(\mathbf{a}, \mathbf{b}) = (\mathbf{a}(1)\mathbf{b}(1), \mathbf{a}(1)\mathbf{b}(2) + \mathbf{a}(2))$$

As a result, F can be expressed as :

$$\begin{aligned} x_i &= F(\mathbf{a}_i, x_{i-1}) \\ &= F(\mathbf{a}_i, F(\mathbf{a}_{i-1}, x_{i-2})) \end{aligned}$$

$$= F(G(a_i, a_{i-1}), x_{i-2})$$
$$= F(c_i, x_{i-2})$$

where c_i is computed from the a's using only the G function.

The above transformation is interesting to us because x_i now depends on x_{i-2} instead x_{i-1}, and we know from Figure 7 that the function F has an execution delay of 3. Therefore we can compute F by adding an additional pipeline that computes c_i from a_i using companion function G. Note that an ID instruction is inserted as a buffer so the loop has an even number of stages, which is necessary for maximum pipelining [10]. Figure 8 shows this implementation scheme for Example 2, where

$$c_i(1) = A_i A_{i-1}$$
$$c_i(2) = A_i A_{i-1} + B_i$$

This added pipeline (see the dashed-line block in Figure 8) will be named the *companion pipeline* in the rest of this paper. By constructing the companion pipeline properly, it is possible to keep the whole pipeline running at maximum throughput.

We note that if the function G is associative, we may have a tree-like arrangement. That is, if the number of stages in F is p, we can construct a companion pipeline consisting of $\log_2 p$ levels of G. Fortunately, it can be proved that the companion function G is indeed associative. Furthermore, if a recurrence function F has a companion function G then any x_i can be expressed in terms of x_j, where $0 \leq j \leq$

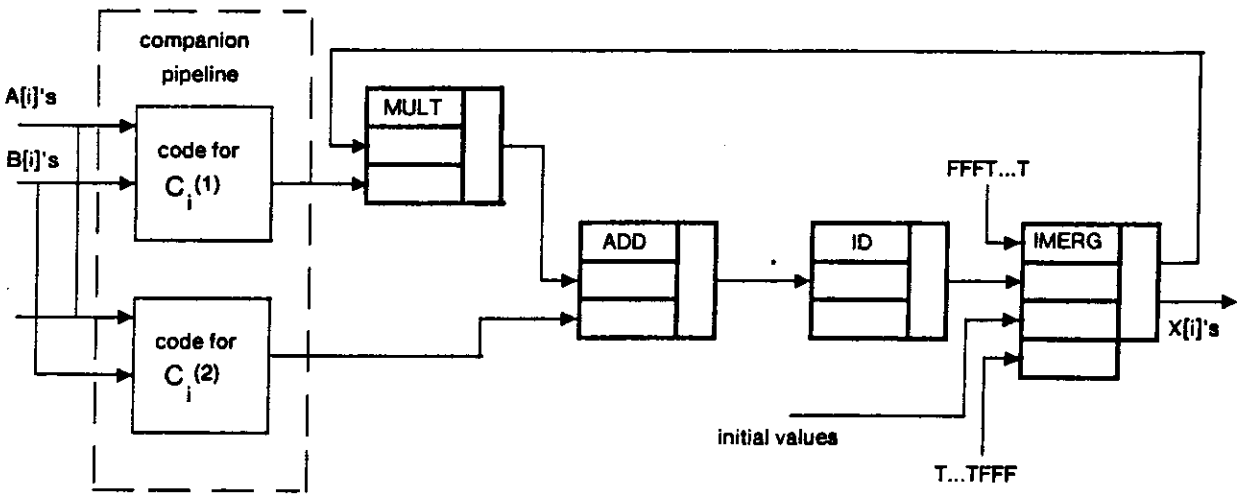


Figure 8. Pipeline Mapping of A Simple for-iter Expression

i and $x_i = F(a(i, i-j), x_j)$. where a(i, i-j) can be expressed by G. The above result can be proved easily from the definition of the companion function.

As we can see from the above analysis, one important factor in applying the new scheme is the introduction of the companion pipeline. The previous work in this area has been to concentrate on its implementation in the context of a conventional machine. There the pipeline is mainly implemented in hardware, such as the pipelined arithmetic and logic unit, with some additional hardware support. The introduction of a hardware companion pipeline is not flexible in such a context. In fact, many different recurrence relations may be used in various computations, and the exact form of the companion pipelines needed for a particular problem is hard to predict. Therefore it is impossible to construct a separate hardware companion pipeline for each possible recurrence relation in the computation. Even if it is possible, the overhead would be very large. On the contrary, the pipeline in the architecture of a data flow machine is software implemented. As a result, it is more flexible to design a piece of data flow program which acts as a particular companion pipeline. This can be performed at compile time, based on an analysis of program structure. Hence, it is much more attractive to apply the new scheme on a data flow machine.

There are some trade off considerations in using this strategy. First, there are many recurrence functions for which no companion function is known. Furthermore, the overhead of backing up of companion functions will grow considerably when the p is big. The complexity of a compiler to analyze the code is also a factor which should be considered.

Now let us return to the problem of classifying Val **for-iter** constructs which have good mapping schemes.

Definition A *simple* **for-iter** expression is a primitive **for-iter** expression such that (1) the recurrence function it denotes has a companion function, (2) the Val expression which computes the companion function is a PE.

Obviously, the code of Example 2 is a simple **for-iter** construct. Consequently, from the results presented in this section, we have the following theorem :

Theorem 3 A simple **for-iter** expression can be mapped into a fully pipelined instruction graph.

8. Fully Pipelined Pipe-Structured Programs

A pipe-structured program in which each **forall** expression is primitive and each **for-iter** expression is simple has an elegant structure; each component is a consumer and producer of array values and has an implementation as a fully pipelined data flow instruction graph. Due to the applicative nature of the Val programming language, the data dependencies among the **forall** and **for-iter** expressions define an acyclic directed graph in which each edge represents a path over which an array value is sent from producer to

consumer. If the instruction graphs for the constituent expressions are connected together according to the acyclic pattern defined by data dependencies of the pipe-structured program, the result is an instruction graph that implements the Val program. Since the component instruction graphs are fully pipelined, the balancing algorithm may be applied to the acyclic interconnection to produce a fully pipelined instruction graph for the complete pipe-structured program.

Theorem 4 For any pipe-structured program in which each **forall** expression is primitive and each **for-iter** expression is simple, a fully pipelined data flow instruction graph can be constructed.

We have developed a formal model of pipe-structured programs for use in studying algorithms for balancing and optimizing corresponding data flow instruction graphs for fully pipelined operation. Interested readers will find a rigorous formulation and analysis of flow dependency graphs in [8]. Some conclusions of this study are:

(1) If the flow dependency graph is acyclic, then a polynomial time balancing algorithm exists and can be constructed. Extra FIFO buffers are usually required for such balancing.

(2) It is possible to reduce the amount of buffering needed to balance a given flow dependency graph. An polynomial time algorithm is presented which can effectively reduce the buffering in many cases.

(3) The optimum balancing of a graph (using minimum number of buffer stages) is equivalent to the linear programming dual of the min-cost flow problem which is known to be solvable in polynomial time [10].

9. Conclution

We have shown how a certain class of programs expressed in the Val programming language can be mapped into machine level programs that operate in a fully pipelined mode, allowing efficient utilization of the units of a high-performance, static data flow computer.

Programs in this class may be thought of as blocks of code connected in an acyclic graph, where each edge represents a producer-consumer relationship between two blocks—one block, the producer, generating the sequence of elements making up an array value, and the other block, the consumer, receiving the elements for use in its computation. Each block corresponds to either a **forall** expression or a **for-iter** expression in the source program that defines a single array value of manifest subscript range. Moreover, we require that each program block be constructed using only *primitive* expressions—nesting of **forall** and **for-iter** expressions and use of certain array constructor operations of the Val language are disallowed.

For programs having this structure, we have shown that machine code can be constructed that performs the specified computation in a fully pipelined mode. The demonstration encompasses all **forall** expressions

that satisfy the structural conditions, but we have only shown how to fully pipeline certain **for-iter** expressions for which a *companion function* for the recurrence relation is known.

Some other techniques for pipelining the implementation of iteration expressions are known—generally involving trading off delay in exchange for achievement of computation at the maximum rate. For example, a recurrence having a cyclic dependence of four operators may be implemented at the maximum rate by introducing a delay (via a FIFO buffer) of length equal to the number of elements in the array being generated.

The extension of this work to array values of multiple dimension is straightforward.

The few benchmark programs we have analyzed carefully all fall into the class of pipe-structured programs. Thus we are hopeful that the extension of these ideas will apply successfully to a large portion of the codes for which execution of a static data flow supercomputer is attractive. Investigating these extensions, and exploring the design of a compiler that will automatically construct fully pipelined code for a large class of Val programs are subjects for further study.

10. References

[1] Ackerman, W. B. and J. B. Dennis. "Val—A Value-Oriented Algorithmic Language Preliminary Reference Manual." Technical Report 218, Laboratory for Computer Science, MIT, Cambridge, MA, 13 June 1979.

[2] Dennis, J. B., Boughton, G. A., and Leung, C. K. C. " Building Blocks for Data Flow Prototypes" Proceedings of the 7th Annual Symposium on Computer Architecture", May, 1980, pp. 1-8.

[3] Dennis, J. B. "Data Flow Supercomputers" IEEE, Computer, Nov. 1980.

[4] Dennis, J. B. and D. P. Misunas. "A Preliminary Architecture for a Basic Data-Flow Processor." *The Second Annual Symposium on Computer Architecture Conference Proceedings*, January 1975, pp. 126-132. Also Computation Structures Group Memo 102, Laboratory for Computer Science, MIT, Cambridge, MA, August 1974.

[5] Dennis, J. B. "Packet Communication Architecture" Computation Structure Group Memo 130, Laboratory for Computer Science, MIT, Cambridge, MA, Aug. 1975.

[6] Dennis, J. B., Gao, G. R., and Todd, K. "A Data Flow Supercomputer" Computation Structure Group Memo 213, Laboratory for Computer Science, MIT, Cambridge, MA, Jan 1982.

[7] Dennis, J. B., Gao, G. R., and Todd, K. "Modeling the Weather with a A Data Flow Supercomputer" IEEE Trans. on Computer, C-33, No. 7, July, 1984.

[8] Gao, G. R. "An Implementation Scheme for Array Operations in Static Data Flow Computer" MS Thesis, Laboratory for Computer Science, MIT, Cambridge, MA, June 1982.

[9] Gao, G. R. "Homogeneous Approach of Mapping Data Flow Programs", Proceeding of 1984 International Conference of Parallel Processing.

[10] Gao, G. R. "Maximum Pipelining Linear Recurrence on Static Data Flow Computers", paper in preparation, 1984.

[11] Kogge, P. M. "A parallel Algorithm for Efficient Solution of a General Class of Recurrence Equations." IEEE Trans. Comput., Vol. c-22, no. 8, Aug. 1973.

[12] Kogge, P. M. "Parallel Solutions of Recurrence Problems" IBM J. Res. Develop., Vol. 18, no. 2, March 1974.

[13] Kung, H. T. "Why Systolic Architectures?" IEEE Computer, Jan 1982.

[14] Montz, L. B. "Safety and Optimization Transformations for Data Flow Programs." Technical Report 240, Laboratory for Computer Science, MIT, Cambridge, MA, January 1980.

[15] Todd, K. W. "High Level Val Constructs in A Static Data Flow Machine" Technical Report 262, Laboratory for Computer Science, MIT, Cambridge, MA, June 1981.

[16] Todd, K. W. "An Interpreter for Instruction Cells." Computation Structure Group Memo 208, Laboratory for Computer Science, MIT, Cambridge, MA, Aug. 1982.