MIT/LCS/TR-280

AN IMPLEMENTATION SCHEME FOR ARRAY OPERATIONS IN STATIC DATA FLOW COMPUTERS

Gao Guang-Rong

This blank page was inserted to preserve pagination.

An Implementation Scheme for Array Operations in Static Data Flow Computers

by

Gao Guang Rong

May 1982

This research was supported by the National Science Fondation
under grant MCS-7915255

Massachusetts Institute of Technology
Laboratory of Computer Science
Cambridge, MA 02139

An Implementation Scheme of Array Operations in Static Data Flow Computers

by

Gao Guang Rong

This report was
submitted to the Department of Electrical Engineering and Computer Science
on 13 May 1982 in partial fulfillment of the requirements for
the Degree of Master of Science

Abstract

The mapping of array operations in VAL programs on a static data flow machine with array memory is studied. The flow dependency graph is introduced as a model of array operations in VAL programs. The balancing and optimization of the flow dependency graphs is presented. The class of well-behaved VAL programs which can be modeled by flow dependency graphs is specified. Schemes for pipelined mapping of **forall** and **for-iter** array operation constructs in well-behaved VAL programs are formulated

Thesis Supervisor: Jack B. Dennis

Title: Professor of Electrical Engineering and Computer Science

key words: parallel processing, flow dependency graph,

pipelined computation, data flow architecture, functional programming,

VAL.

Acknowledgments

I am gratefully indebted to my thesis supervisor, Prof. Jack Dennis for the freedom of research throughout this work and his guidance in the formulation and development of this thesis. Without his encouragement and patience, this document would not be possible.

I wish also to thank Prof. Arvind for his interesting suggestions and discussions during this research.

I am grateful for the very friendly working atmosphere and active academic environment of the Computation Structure Group at MIT. I would like to thank Clement Leung, Dean Brock, Bill Ackerman, Andy Boughton, Ken Todd, Kashev Pingali and Willie Lim for their encouragement and discussions. I also thank Nena Bauman for her critique on my draft of this thesis.

Finally, I am grateful to my friends, my parents and family who always provide me with support, encouragement and love.

Table of Contents

1. Introduction

As the use of computers expands in scientific areas, the demand for greater levels of computing power has risen rapidly. To meet this demand, enormous efforts have been put into two major areas : the improvement of hardware technology for the increasing speed of computations; and the design of appropriate machine architectures for parallel processing.

Although many advances have been made, the appetite for computing power has not been satisfied. Scientists continuously find new applications which beyond the capacity of our most powerful computers. The improvement of hardware technology has certain physical constraints. Conventional multi-processor machine architectures also have some problems. Among others, a serious problem is to devise a suitable scheme to program the data communication between parallel processors. The sequential nature of conventional languages and machine architectures has proved to be not quite successful as an adequate and practical approach.

Researchers have been searching for new machine architectures. Data flow concepts have been proposed as one promising approach. Several data flow machine architectures have been designed and studied. New high level programming languages have been designed as appropriate tools for describing both algorithms and their mapping onto the processing and memory resource of data flow computers.

Array manipulation often forms a major part of scientific computation. As a result, efficient mapping of array operations in a high level language onto a suitable machine architecture is a crucial consideration in language design and implementation.

This thesis is a study of the mapping of large scientific computations onto high performance data flow computers. The study is based on the high level programming language VAL, intended for expressing large scientific programs for data flow computation.

This chapter will proceed with a more detailed look at the programming language VAL (section 1.1), followed by a description of the static data flow machine architecture which is the form of architecture assumed for this research (section 1.2). Section 1.3 presents the form of machine code assumed for the proposed data flow machine. A general discussion of array operations in the context of data flow architectures, is presented in section 1.4. Finally, section 1.5 will give a synopsis of the remainder of the thesis.

1.1 Programming Language VAL

The programming language VAL [1] is a high level language designed particularly for expressing algorithms to run on computers that are capable of achieving parallel execution, especially those machines based on data flow architecture. VAL is also a functional language which lacks the concepts of storage, or rather, makes them hidden from the programmer. In VAL, variables are treated differently from varibles in conventional languages. When a variable is declared in VAL, it is assigned a value which it retains for its lifetime. This is called *the single assignment rule.*

The basic programming unit in VAL is the function. Both the inputs and the outputs to a function should be explicitly declared. When a function is called, arguments are passed to it and a set of one or more values are computed as the result. It is interesting to note that a function is guaranteed access to only those argument values, i.e., no global variables are needed. Furthermore, a function will not change the argument values because VAL is side-effect free. The body of a function consists of expressions. Usually, these expressions are built using the **let-in**, **if-then-else**, **forall**, **tagcase** and **for-iter** constructs of VAL. These expressions are also side-effects free.

One important feature of VAL is that it provides *implicit concurrency* (operations that can be executed independently are evident without the need for any explicit language notation). This allows programmers to focus on the presentation of the algorithm, meanwhile the concurrency can be easily identified by both compiler and reader. This is a very desirable feature, since, in a data flow environment, execution speed depends heavily on a programmer's ability to write programs that contain large amounts of concurrency. It is impractical to ask programmers to specify all concurrency explicitly in the program.

The main target of the mapping of VAL programs is the data flow computer. The machine level program in a data flow machine is a data flow graph. The nodes of the graph represent operations; the directed arcs represent data paths. The execution of such a graph is based on the firing rule. An operator is *enabled* as soon as all its input values are present. The execution (firing) of an operator will consume input values and put output values on each of its output arcs. Figure 1.1 shows a data flow graph representation of a VAL function which computes the sum of squares of two values.

There are many interesting features in the VAL language. This thesis concentrates on the implementation of array operations in VAL language. To be more specific, among the array operations in a VAL programs, we are only interested in those which can be expressed by **forall** and **for-iter** constructs. A general model for a certain class of VAL programs containing such constructs is introduced in Chapter 2. A detailed discussion of the mapping scheme forms the major portion of

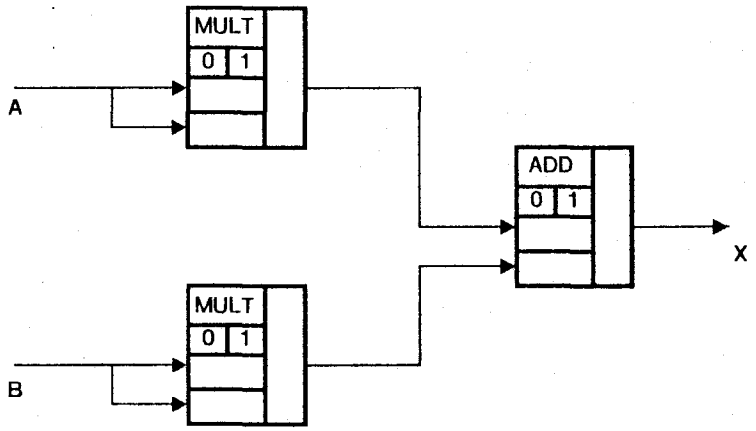


Figure 1.1.

Chapter 3.

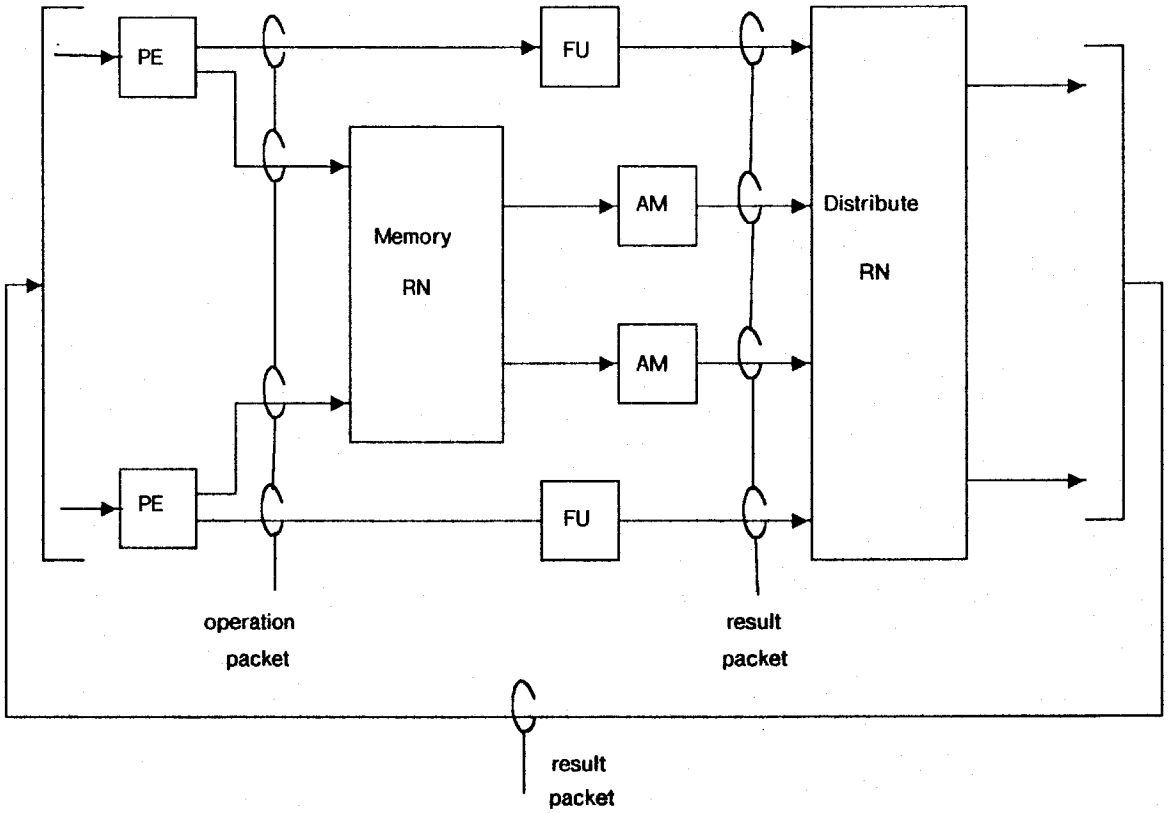
1.2 A Static Data Flow Machine Architecture

Figure 1.2 diagrams a static data flow machine with array memory. It is an extension of Dennis-Misunas Form 1 Data Flow Machine [4] to include an array memory.

This machine architecture is static in the sense that the instructions of machine level programs are loaded into specific memory locations in the machine before computation begins, and only one instance of an instruction is active at a time. In a static architecture, resource allocation is decided mostly during compile time in contrast to other proposed data flow architectures [17] where most of resource management is done at run time.

The system consists of four parts : *processing elements* (PE), *function units* (FU), *array memories* (AM), and *routing networks* (RN). The whole system is a packet communication architecture [5], and there are two kinds of packets in the system : *operation packets* and *result packets.*

Processing elements are used to hold the instructions of data flow programs. When the operand values of an instruction have arrived, the instruction is *enabled.* The execution of an enabled instruction will send an operation packet to one of the routing networks. If an enabled instruction calls for a scalar arithmetic operation (a floating point addition or multiplication, for example), the corresponding operation packet is sent to a functional unit capable of performing that operation. If it calls for an array memory operation for building array values or accessing elements of arrays, the corresponding packet is sent to the array memory. Instruction execution in a functional



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

Figure 1.2.

---

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. The result packets are sent to the processing elements that hold the target instruction through the *distribution* routing network. The use of packet routing networks in data flow computers is discussed in [6].

The array memory units are provided to hold arrays of data values and are accessible from any of the processing elements. For the static architecture, we suggest a one-level memory organization rather than an multi-level memory hierarchy.

In general, the array memory is constructed by several memory modules. The total size of the array memory depends on the application. To simplify the addressing mechanism, we assume that

the number of memory modules is an integer power of two. All memory modules are identical asynchronous packet memories. The use of FIFO buffers is important to the mapping schemes developed in this thesis; their implementation is discussed in Chapter 3.

1.3 Instruction cells

Todd proposed a set of preliminary machine instructions for a static data flow machine which is being designed at MIT. A list of the operation code for the elementary scalar operations can be found in [16]. We will use this instructions set and augment it with appropriate array memory instructions.

An instruction cell consists of following fields :

(1) *opcode* : operation code

(2) *operand* : the place to hold operands values

(3) *destination* : the addresses of target instructions where the result of applying the opcode to the operands is to be sent.

(4) control information

Each destination address is simply the address of an instruction to which the result is to be sent. For each destination address, there is a *use field.* One part of the use field specifies whether the destination cell is to receive an acknowledge [16] or a result, and in the latter case, specifies to which operand to sent the result. Another part is used to provide for switching between alternate destinations under control of a third (boolean) operand.

An instruction cell has another important field which contains the number of acknowledge signals that the cell must receive before it can be enabled. When this number reaches zero, it assures that all the destinations are ready to accept the result, so that the instruction can be fired again. This number must be reset after the instruction is fired. An instruction also contains the information necessary for resetting.

In addition to the scalar instructions, we also suggest two instructions for memory operations, *i.e.*, *read* and *write* as shown in Figure 1.3. The read instruction has only one operand, i.e., the address of the memory location to be read. When executed, it will send a read instruction packet to the corresponding array memory module through routing networks. The write instruction requires two operands : the address of a memory location and the value to be written into that location. When fired, it will send a write instruction packet to the corresponding array memory module where the value part will be stored in the specified memory location.

Besides the read and write instructions, we also need some other machine instructions to assist

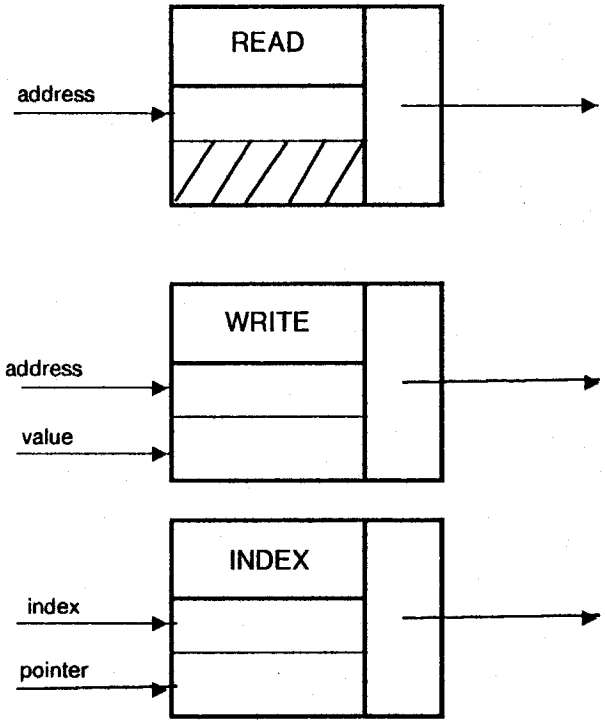


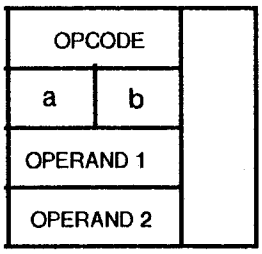
Figure 1.3.

implementation of array operations. For example, to check that the array index is within the subscript bounds, we introduce the *index* instruction, also shown in Figure 1.3. It has two operands, the first operand is the array index value (an integer), and the second is a pointer value points to the starting address of a block of storage in array memory. When the index exceeds the array bound, the **undef** value is returned as the result. The index instruction is usually used together with a read or write instruction to assure that any address sent to an array memory module is within the intended array bounds.

In this thesis several notations will be used to denote machine instructions as shown in Figure 1.4. The reader can find more detailed description of those notations in [16].

1.4 Pipelined mapping of array operations

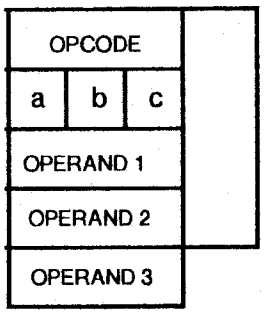
Numerical computation usually involves operations on arrays of data. Problems such as those found in weather forecasting, nuclear physics, thermodynamics, and seismology always seem to exceed the capabilities of conventional architectures. Despite these demands, however, the kernel of such problems is typically more structured than that of other applications. Mathematically, such



(1) without gating power

a : acknowledgement needed

b : reset value

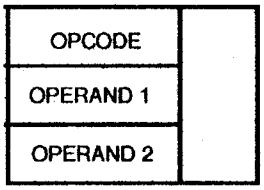


(2) with gating power

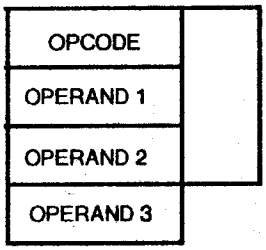
a : acknowledgement needed

b : reset value for ture

c : reset value for false



(3) abbreviation for (1)



(4) abbreviation for (2)

Figure 1.4.

---

problems can usually be expressed in linear algebra with all elements of rows or columns of large matrices manipulated in some identical fashion. A typical piece of a VAL program for such problems is shown below.

```
B : array[array[real]] : =
      forall      i in [1, m+1], j in [1, n+1]
      construct
         if i = 0|i = m + 1|j = 0 |j = n + 1 then A[i,j]
         else 0.25* (A[i,j-1] + A[i,j+1]
                        + A[i-1,j] + A[i+1,j])
         endif
      endall
```

In conventional computers the concept of array is explicitly related with a block of memory locations, and the transformation of array values is done by making successive replacements of element values in the array. In data flow computation, we take a different view, i.e., a computation *constructs* an array value from other array values, scalar quantities and constants. Two representations of array values are used in a static data flow machine. Array values may be hold by a group of destination arcs at some moment or carried by one arc as a sequence of values at successive moments. These two ways of representing array values illustrate the basic space/time trade-off in structuring machine code for a static data flow architecture.

Computer architects have long resorted to a series of design techniques that are classified under the general term of *concurrent operation.* Within this general category are two well recognized techniques *parallelism* and *pipelining.* Parallelism emphasizes concurrency by replicating (often exactly) a part or the whole computation structure (hardware, instruction cells, etc.) many times. High performance is attained by having all structures execute simultaneously on different parts of the problem to be solved. Pipelining, the main interest of this paper, takes the approach of splitting the function to be performed into smaller pieces or stages. The data will flow through the stages of the pipeline at a rate that is independent of the length of the pipeline and dependent only on the rate at which new entries may be fed into the pipeline. In a data flow computer, the scheme of pipelining is very naturally embedded in the machine architecture which will be explained later.

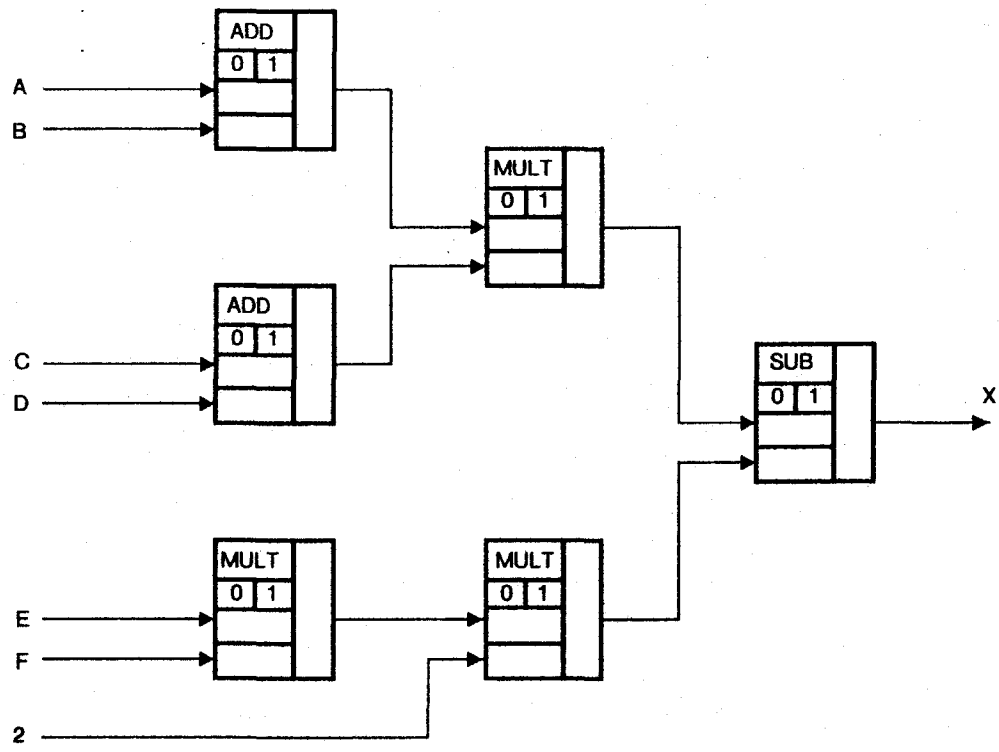
An array of values may be produced (or consumed) either in parallel or in pipelined fashion. In this thesis, the strategy to implement array operation is pipelining. The author has been studying a typical application program, i.e., the GISS weather code [8], which is of interest to NASA. A subset of the code is translated into VAL. Array operations are used extensively in the program, but the structure of the array computation is static, that is, all the arrays manipulated in the program are fixed in size, and their dimensions are not changed during runtime. The substantial parts of the array operations are readily expressed using VAL **forall** constructs. There are also array operations expressed in **for-iter** constructs. We will study the pipelined mapping schemes in the later chapters.

To see how pipelined execution is realized on a data flow machine, a simple example of data flow machine code is shown in Figure 1.5, which corresponds to the following expression :

$$X : \textbf{real} := ((A + B)*(C + D) - 2*E*F)$$

It is observed that successive sets of values can be fed into the inputs and be computed by the graph. In a static data flow machine, an instruction cell must not fire again until all results sent out by the previous firing have been consumed. A data flow graph having the above property is called *safe* and the data/acknowledgment pair is proposed to ensure safety of the execution [14].

The major interest of this thesis is the pipelined mapping of VAL array operations on a static data flow machine. To achieve maximum throughput of a pipeline, we need to solve several basic problems (From now on, we will use maximum pipelining or full pipelining to describe a pipeline with maximum throughput, although no definition is presented until section 2.2.1). First, not all data flow graphs can achieve maximum pipelining. Second, for those that can be fully pipelined the graph often requires some transformation. Third, we always want to do the most efficient transformation to



X = (A + B)*(C + D) - 2*E*F

Figure 1.5.

achieve the maximum pipelining. In the rest of this thesis, we will study these problems in detail and give appropriate solutions whenever possible.

1.5 Structure of the thesis

Having established a basic background, we proceed to study the main topic of the thesis.

In Chapter 2, we will study the global structure of a class of VAL programs in terms of array operations. A model of such VAL program is developed. The balancing and transformation schemes to achieve maximum pipelining for the suggested graph model are formulated. In chapter 3, mapping schemes for VAL **forall** and **for-iter** array operation constructs are studied and developed. Different implementations of FIFO buffers, which are very important for these mapping schemes, are discussed. Then, the class of VAL program which can be modeled and handled by the techniques developed in this thesis is specified. Conclusions are presented in Chapter 4 along with suggestions for further research.

2. The flow dependency graph and its application

In this chapter we will develop a model for studying array operations in a class of VAL programs. We know that the global structure of a VAL program is a set of function modules. The only means for a function module to communicate with other parts of a program is the defined inputs and the returned value which must be explicitly declared in the function header. In this thesis we focus our attention on *intra-function* rather than *inter-function* flow analysis, since the results derived may be extended to the latter case. Furthermore, we are only interested in array construction operations which are expressible using **forall** and **for-iter** constructs. The **forall** construct in VAL is provided for the situation that all elements of the array being created can be computed concurrently. The **for-iter** construct is designed to correspond to tail recursive function definitions. It can be used to express the situation where data dependency exists between successive elements of the array being created.

In section 2.1 we will introduce the concept of *flow dependency graph* to study the global structure of VAL programs. Under certain assumptions, the flow dependency graph is a very good mathematical representation of the global computation structure of a class of VAL programs. This class of VAL programs will be characterized in section 2.2. We will show that to achieve high throughput, buffers should be introduced to balance the flow dependency graph. An algorithm for balancing an acyclic flow dependency graph is presented in section 2.3. The issue of optimum balancing for such a graph is also studied. However, we have not found a good general algorithm for optimizing an arbitrary acyclic graph. What we do is first define some transformation rules necessary for reducing the amount of buffering in a flow dependency graph. Based on those rules, an algorithm is given which can decrease the amount of buffering considerably, yet no guarantee is given that the result is optimum. Then, in section 2.5 and 2.6, we concentrate on a special class of graph, namely well-structured graphs, and show that a good algorithm does exist for their optimization. Some theoretical results about the complexity of these algorithms are formulated.

The schemes introduced in this chapter are only a first step towards a general solution for the balancing and optimization of the pipelined data flow programs. However, the author hopes this work can reveal some important information about the key problems and can serve as a hint for more satisfactory solutions.

2.1 Flow dependency graph for VAL programs

A flow dependency graph is a presentation of the structure of a VAL program. We first state its definition :

Definition. A *flow dependency graph* (FDG) is a directed graph G = (V,E), where :

(1) V is a set of nodes corresponding to VAL program components which can process (i.e. produce or consume) array values,
(2) E is the set of dependency arcs (edges) which express the data dependency between pairs of nodes.

note : all nodes in the graph are value producers or consumers or both. In this thesis we assume they correspond to **forall** and **for-iter** constructs.

We also need further terminologies to describe the graph, most of which are similar to those used in graph theory [9].

An edge (u, v) is called an *input* (or *incident*) *edge* of node v and an *output* (or *adjacent*) *edge* of node u. If (u, v) is an input edge of node v, then nodes u, v are called the *initial node* and *terminal node* of (u, v) respectively. The *input degree* of a node is the the number of its input edges, and the *output degree* is that of its output edges. A node is *multi-input* (or *multi-output*), if its input degree (or output degree) is greater than one. The set of input edges of a node is called its *incident edge list* and the set of adjacent edges of a node is its *adjacent edge list.* A *path* is a sequence of edges $(e_1, e_2 ... e_k)$ such that the terminal node of e_i is the initial node of e_{i+1}. We call the initial node of e_1 the *start* node of the path and the terminal node of e_k the *end* node of the path. Also, node u is *reachable* from v if there is a path from v to u.

The dependency relation between these nodes of an FDG is defined as follows.

Definition Let A, B be two nodes of a FDG. Node A is *dependent* on B if B is referenced in A. The corresponding arc in the graph will be denoted by a directed edge from A to B.

It is important to clearly characterize those VAL programs that can be effectively modeled by a FDG. This class of VAL program has following feature :

(1) All arrays in the program are *static*, *i.e.* array bounds are compile-time computable constants.
(2) All array construction operations in the program are expressible by **forall** and **for-iter** constructs.
(3) For all array constructions represented by **forall** and **for-iter** constructs, good schemes exist for mapping them into fully pipelined data flow machine codes.

A VAL program which satisfies above criteria is called a *well-behaved* program.

Note that in above formulation we are sloppy in using the word "good scheme" for mapping of pipelined data flow graph. It is the author's intention not to go into details of the internal structure of nodes in a FDG at this time. Using the abstraction of nodes, we can study the global structure of a VAL program more easily. In Chapter 3 we will define a class of **forall** and **for-iter** array operation constructs which can be efficiently mapped into pipelined data flow graphs. We also will develop general schemes for such mappings. Consequently, we will present a more formal definition of

well-behaved VAL programs at the end of Chapter 3.

From above definition, the FDG for a VAL program has two types of nodes :

(1) *forall* node (will be called F-node) . It corresponds the **forall** construct which constructs an array.

(2) *For-iter* node (will be called L-node). It corresponds to a **for** iteration loop which constructs an array.

Since F-nodes and L-nodes will produce arrays, we often use the names of those arrays to name the nodes. For example, a F-node produces an array A will be named node A or simply A.

We should point out that our notion of well-behaved VAL programs is applicable to many programs in numeric computation. The kernels of such problems are typically more structured than those of other applications. Following is an example of a well-behaved VAL program. Figure 2.1 shows the corresponding FDG.

```
Function (A : array[real] return array[real])
let
   B : array[real] : =  forall I in [1, IM]
                        construct 5 * A[I] + 4
                        endall;
   C : array[real] : =  forall I in [1, IM]
                        construct exp(A[I], 2) - 7
```

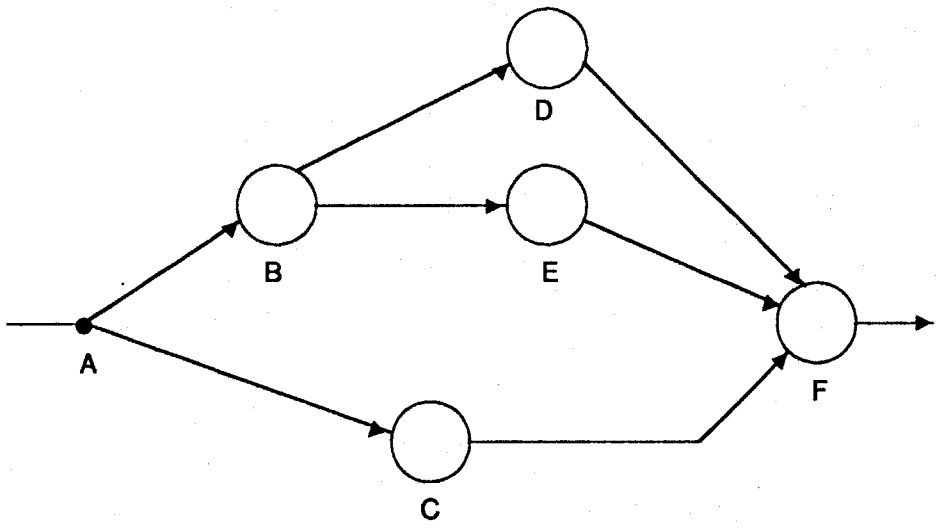


Figure 2.1.

```
                endall;
    D : array[real] := forall I in [1, IM]
                construct
                    if I = 1 then B[I]
                    elseB[I] + B[I-1]
                    endif
                endall;
    E : array[real] := forall I in [1, IM]
                construct 1/2*exp(B[I],3)
                endall;
    F : array[real] := forall I in [1, IM]
                construct C[I]*D[I]*E[I]
                endall;
in F
endfun
```

A few remarks are in order about the difference between the "flow dependency graph" developed in this paper and the "data flow graph" used in many publications [2][3]. Both concepts are introduced for pre-execution data flow analysis and aimed at optimizing the mapping of higher languages to their machine level representation to improve the efficiency of program execution. But the two approaches are different. The data flow graph is mainly used to analyze the conventional high level languages (Fortran, ALGOL, etc.), and it is based on the control graph [13] which is like the flowchart scheme, and is not directly related to the actual data flow in the program. Our concepts are based on the functional type of language, where program execution is the evaluation of mathematical expressions. So nodes in the flow dependency graph are nothing but mathematical value producers or consumers, and the dependency arcs represent the abstraction of real data path. I strongly believe that for functional languages, the FDG is a more straightforward and adequate model for the representation of the computation structure. In particular, if the target of the translation is a data flow machine, the FDG model will become extremely powerful because it is just the "higher level" data flow program for the machine. We will show that the proper reduction and transformation of the flow dependency graph of a VAL program will directly help to produce efficient machine code.

2.2 Balancing of flow dependency graph

In this section we study the issue of flow balancing for a FDG and present an algorithm for balancing an acyclic FDG.

A key parameter in determining the performance of a pipeline is the *latency*, or minimum number of time units needed for separating two successive values of the same input of a pipeline. Another measure is the *initiation rate* of a pipeline, which is the number of values which can be fed

into the input of the pipeline in one time unit. The goal of balancing a FDG is to increase throughput by modifying the graph to allow for maximum pipelining. Let us consider the consequence of presenting the input of the graph with sequences of array elements. We hope that the processing of a second value can begin before the output of the previous value appears, with the best situation being that consecutive array values can pipeline through the graph with the highest initiation rate.

Before proceed to formulate the FDG balancing problem, we should define the criteria of a pipelined graph more carefully. As stated above, we are interested in mapping well-behaved VAL programs on to a static data flow machine, or more precisely, the Dennis-Misunas machine [4]. An arc of a data flow graph in such a machine is a data/acknowledge (d/a) pair. It has been shown that a graph with such arcs guarantees determinate (safe) and deadlock free (live) operation [14]. We assume that the arcs in our FDG model are also such d/a pairs.

Now we are ready to define the time scale used to specify the performance of the pipeline corresponding a FDG. We assume the reader is familiar with the notion of *firing* of a node in a data flow machine. Note that the nodes in a FDG which corresponds to a well-behaved VAL program are defined as those that can be mapped into fully pipelined machine level program for a static data flow machine. As pointed out before, this is a software pipeline with each stage being not a hardware circuit unit but an instruction (operator) of a data flow machine. In general the data flow graph operates asynchronously, however we can analyze it by assuming somewhat synchronous behavior. One important assumption is that during a given unit of time all enabled operators will fire and produce a result. In the context of Dennnis-Misunus architecture, the repeated firings of any particular operator should progress as follows : an operator which fires at time one should receive acknowledges from its successive operators during time two, reenabling it to fire during time three. We will call the above time delay the *execution time* of the operator. In a real data flow machine, different operators in a pipeline will have different execution times. The distribution of execution times is implementation dependent. We will not study these details , but simply assume that the delay time in the pipeline is the maximum execution time of any of its operators. This is taken as one *basic time unit.* Under such assumption, the execution time of any operators in the pipeline can be regarded as a constant (one basic time unit). A graph is *maximum pipelined* (*fully pipelined*) if the delay between an operator's firing and receiving acknowledge signals is one basic time unit. On the other hand, if a graph is *limited pipelined*, the delay between an operator's firing and receiving acknowledge signals may be several basic time units.

The FDG for a real VAL program may be very complex. The stages making up a node in the graph may have different execution times. The number of stages in each node is also different. All these factors make the balancing of a FDG not a trivial task. However, we will show that for some important classes of FDGs good balancing algorithms exist and are useful in compiler design. As a conclusion, we summarize the basic assumptions about FDG's for which we will develop balancing algorithms in the following sections. These assumptions are :

(1) It corresponds to a well-behaved VAL program

(2) It is acyclic. All its arcs are data/acknowledgment pairs and the nodes obey the firing rules as defined in [14].

(3) The execution time of any stage is a constant, called a basic time unit.

A few remarks are necessary for these restrictions. First, to study the balancing of a well-behaved FDG, it is enough to know the number of stages in each node. In a static architecture, when a mapping scheme for a particular node is chosen, its stage number is fixed and is known to the compiler. We will call it the *execution delay* of the node. Second, no good algorithm is known which can balance a general FDG with cycles. Third, we may think of the arcs in a FDG as the data/acknowledgment pairs in the Dennis-Misunas machine, which is very important for the safety of the program. However, we are not interested in the implementation details of the arcs. Instead , we simply assume such data/acknowledgment arcs are used so the graphs are always safe. Finally, these do not include all the assumptions, we will introduce more while developing the algorithms.

It should be stressed that the assumptions presented in this section will be used throughout the rest of the thesis.

2.2.1 Formulation of the balancing problem

Consider a simple FDG as in Figure 2.2, which consists of nodes *I, A, B* and *C*. Suppose the number of stages of nodes *I, A* and *B* are 3, 4 and 6 respectively. Since all arcs are data/acknowledgment pairs, the graph is *safe* in the sense that no arc can have more than one data token on it. The interesting and problematic issue arises when presenting the graph with a sequence of array elements. There are two paths from I to C (see figure), *i.e.* path 1 and path 2. Observe that it takes 10 stages before an input value arrives at node C through path 2. Node *C* can not be fired until all of its inputs are available. As a result, the second set of values can not be fed into the pipeline in 10 time units, because only one value may exist within path 1. So the minimum latency of the graph is greater than 10 stages and the maximum rate is less than 1/10. While the safety avoids the internal conflict of the pipeline, it also imposes severe restrictions on the degree of concurrency. To eliminate this undesirable behavior so that successive values of an array may pipeline through the graph with

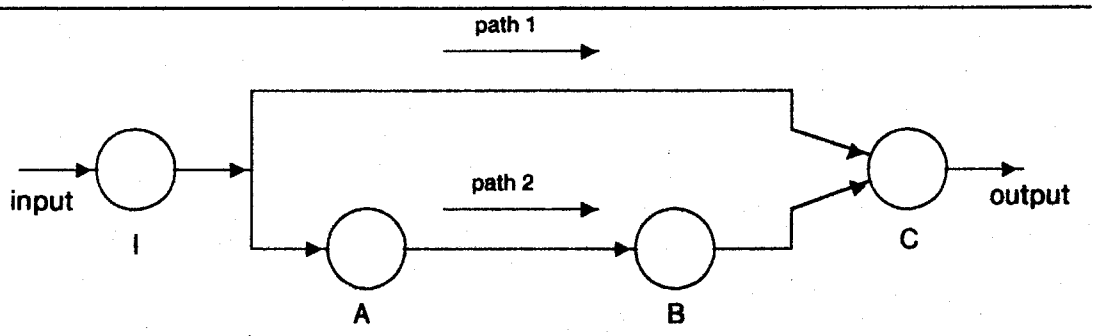


Figure 2.2.

maximum throughput, we introduce some buffering in certain arcs such that they can hold an appropriate number of value tokens. For the above example, if we introduce in path 1 a buffer E of 10 stages delay, the "length" of path 1 and path 2 will be balanced. Thus, the latency of the graph will be decreased to one, and full pipelining could be achieved. The resulting graph is shown in Figure 2.3, where buffers are represented by boxes.

Now we are ready to give a formal description of the balancing problem. Our discussion will be restricted to the *one-in-one-out* acyclic FDG (*i.e.* the FDG has only one input node and one output node). The extension to an arbitrary FDG is addressed in the last chapter. We first introduce the concepts of a weighted FDG.

Definition A *weighted FDG* $G' = \{V,E,W\}$ corresponding to a FDG $G = \{V,E\}$ is a weighted directed graph where W is a function from E to the set of non-negative real numbers.

One sensible way to convert a FDG to a weighted FDG is to assign weights to each of the output (adjacent) edges of a node such that the weight assigned to an edge equals to the stages of the corresponding node. For example, the weight assigned to arc (i,j), *i.e.* $w(i,j)$ equals the number of the stages of node i. In Figure 2.4 we present the weighted FDG for the graph in Figure 2.2.

We also define the *cost* of any path to be the sum of the weights on all edges in the path. As a result, the cost of a path from node i to node j is the time needed for a data token to travel along the corresponding path from i to j.

Definition Let G be a FDG with input node z and let p be an arbitrary multi-input node of G. If the cost for any two different paths from z to p are equal, then G is called a *balanced* FDG. Otherwise G is unbalanced.

It is easy to see that the latency of a balanced FDG equals one, that is maximum pipelining can

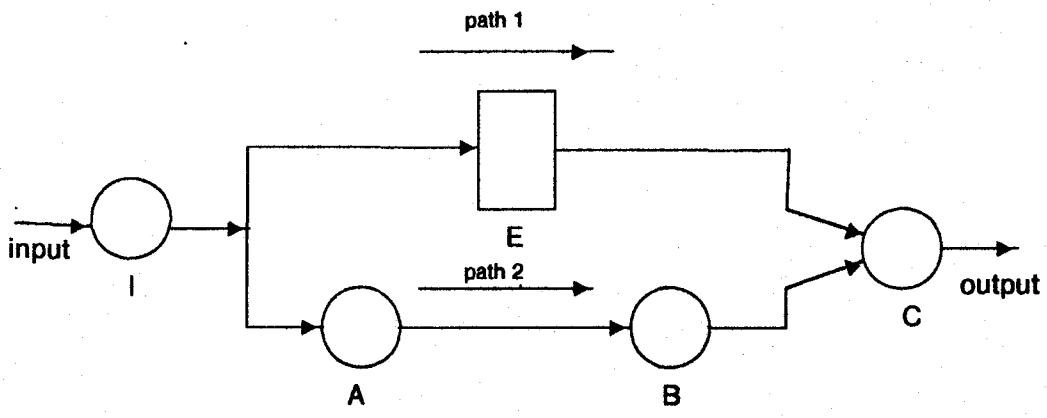


Figure 2.3.

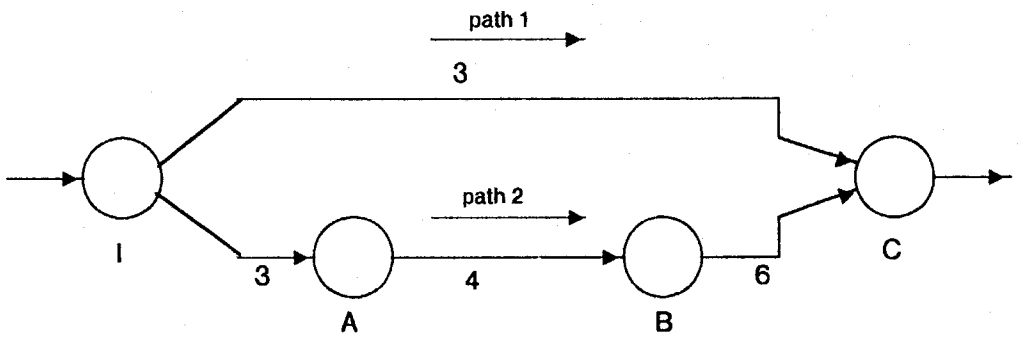


Figure 2.4.

---

be achieved, thus we have the following theorem.

Theorem 2.1 A balanced FDG can achieve the maximum pipelining.

Unfortunately, most FDGs of user programs are unbalanced. To transform one into a balanced graph, appropriate buffers must be introduced. The procedure to modify an unbalanced FDG into a balanced one by introducing buffering is called *balancing* of the FDG. A FDG is *optimally balanced* if the balancing procedure introduces the least amount of buffering.

In the rest of the chapter we will discuss the following questions :

(1) Is it always possible to balance an arbitrary acyclic FDG ? If so, what is the algorithm ?

(2) Is it possible to reduce the buffering necessary for balancing of a FDG ? What is the transformation needed ?

(3) Is it possible to define a special class of graph such that a good optimization algorithm exists ? If it is, what is the algorithm ?

(4) What is the complexity of the algorithms ?

Unless explicitly stated, our further discussion is concentrated on the kind of FDG which satisfies all assumptions introduced so far. We will show that polynomial time algorithms exist for such problems, and the results are summarized in theorem 2.2 to 2.6.

2.2.2 An algorithm for balancing of FDGs

In this subsection we will answer the first question, that is we will give an algorithm which can be used to balance any acyclic FDG. From the definition of a balanced FDG we can note that the key of balancing a graph is to make different paths from the input to any particular multi-input node have equal costs. The most convenient places to introduce buffering are the arcs which are input edges of those multi-input nodes.

Intuitively we can travel the FDG starting from the input node and examine the cost of the paths to each multi-input node by following a depth-first search strategy. The cost for the paths from the input to all nearest multi-input nodes are recorded. Then a comparison is made and the maximum cost of the paths for each those nodes are found. Then we can compute and add appropriate amount of buffering for all other paths to balance the parts of the graph which have been traveled and proceed to walk toward the further multi-input nodes. Eventually, the whole FDG will be balanced.

The balancing algorithm we will present can be applied to any acyclic weighted directed FDG. Let G = $\{V,E,W\}$ be a weighted FDG. Let M be the set of all multi-input nodes in G. Starting from the input node, we walk though the graph according the following : an edge (v,u) is processed and placed in a set T if node u has not been previously visited when we are at node v considering edge (v,u). A stack is maintained which will remember the cost of a path from the input node to the most recently visited node. The push and pop operations of the stack are defined as usual, and top(stack) is the top element of the graph. We assume for each node $v \in M$, a cost list C is maintained which can record the cost of paths from the input node to v. For example, $C(u,v)$ is the cost of the path from the input node to v where the last edge on the path is (u,v). We use B to denote the buffers for balancing the FDG, *i. e.* $B(u,v)$ denotes the buffer on edge (u,v).

An algorithm for balancing a FDG is given below.

Algorithm 2.1 Balancing of a directed FDG

input : A FDG G = (V,E,W) represented by the adjacency list $L[v]$. An incidence list $I[v]$ is also maintained for each node v, where $v \in V$.

Output : A balanced FDG G' = (V,E,W') with buffers added to the incident edges of multi-input nodes.

Method. The recursive procedure SEARCH(v) adds edge (v,u) to T if node u is first reached during the search by an edge from v. We assume top(stack) and elements of C and B are initialized to 0. The entire algorithm is as follows :

1. $T \leftarrow \emptyset$;
2. let z be the input node
3. do SEARCH(z)
4. construct a FDG G' = (V,E,W'), where
   $w'(v,u) = w(v,u) + B(v,u)$

end

The following is the recursive procedure SEARCH(v) which does a depth-first search of the FDG starting from v, locates the buffers for balancing and computes the amount of buffering needed.

SEARCH (v)

begin

```
5.      for each u ∈ L[v] do
6.          if (v,u) ∉ T then
            begin
7.              add (v,u) to T
8.              push (top(s) + w(v,u))
9.              if u ∈ M then
10.                 C(v,u) = top(s)
11.                 if for all (x,u) ∈ I[u], C(x,u)
                        are defined then
12.                     let R = max{C(x,u) | (x,u) ∈ I[u]}
13.                     for all (x,u) ∈ I[u] do
                        begin
14.                         B(x,u) = R - C(x,u)
15.                         top(s) ← R
16.                         goto line 19
                        end
17.                 else
18.                     goto line 20
                    endif
                endif
19.             SEARCH(u)
20              pop
            end
21.         endif
        endfor
22.  end
```

It is easy to show that G' is a balanced FDG.

Proof of Algorithm 2.1 First, we claim that the algorithm will terminate. A call to SEARCH (v) will test all possible output edges of a node in its adjacency list (line 5,6). When an edge is not yet included in T, it will be put in T (line 7). Then a test is made to see if the node is a multi-input node. If it is, a check is done to see whether the cost of all of the paths from v to this node has been computed. If it is not a multi-input node or if it is a multi-input node with all path costs computed, a recursive call to search the terminal node is executed (line 19). Otherwise, the algorithm will continue to test the rest of the edges in the adjacency list of v, if any exist. Since the graph is finite, eventually all edges will be included in T and the algorithm will terminate.

We claim that G' is balanced because the costs of each of the paths from input node z to a particular multi-input node are the same. This is true since the algorithm checks the cost from z to each of its nearest multi-input nodes and introduces the right amount of buffers at their incident

edges (line 12 - 14). Then the algorithm proceeds to balance the path from those nodes to still further multi-input nodes *etc.* This leads to the following theorem :

Theorem 2.2 If a FDG is a directed acyclic graph, then it can be balanced using Algorithm 2.1.

Now let us study the complexity of Algorithm 2.1. Assume the number of nodes of a FDG is n. The time spent in SEARCH(v), exclusive of recursive calls to itself, is proportional to the number of vertices adjacent to v. Since G is acyclic and (v, u) is put in T the first time SEARCH(v) is called, there is only one call for each v. Thus the time spent in SEARCH is $O(e)$ where e is the number of edges in the graph. We know that if an acyclic directed graph has n nodes and e edges, then $e \leq n(n-1)/2$. Hence we have the following result,

Theorem 2.3 Algorithm 2.1 requires $O(n^2)$ steps to balance a FDG with n nodes.

Before closing this subsection, we introduce a concept which is useful in the subsequent discussions. In general, we have :

Definition Let G = (V,E,W) be a weighted FDG and G1 = $(V,E,W1)$ be a weighted graph, where the weight $W1$ corresponds to the buffering introduced on E. Then G1 is called a *buffering graph* of G. Furthermore, a FDG G' = (V,E,W') can be composed from G and G1 such that

$$w'(u,v) = w(u,v) + w1(u,v)$$

If G' is a balanced FDG, then G1 is a *balanced buffering graph* for G.

When drawing a buffering graph, we usually leave the edges with 0 buffering unlabeled.

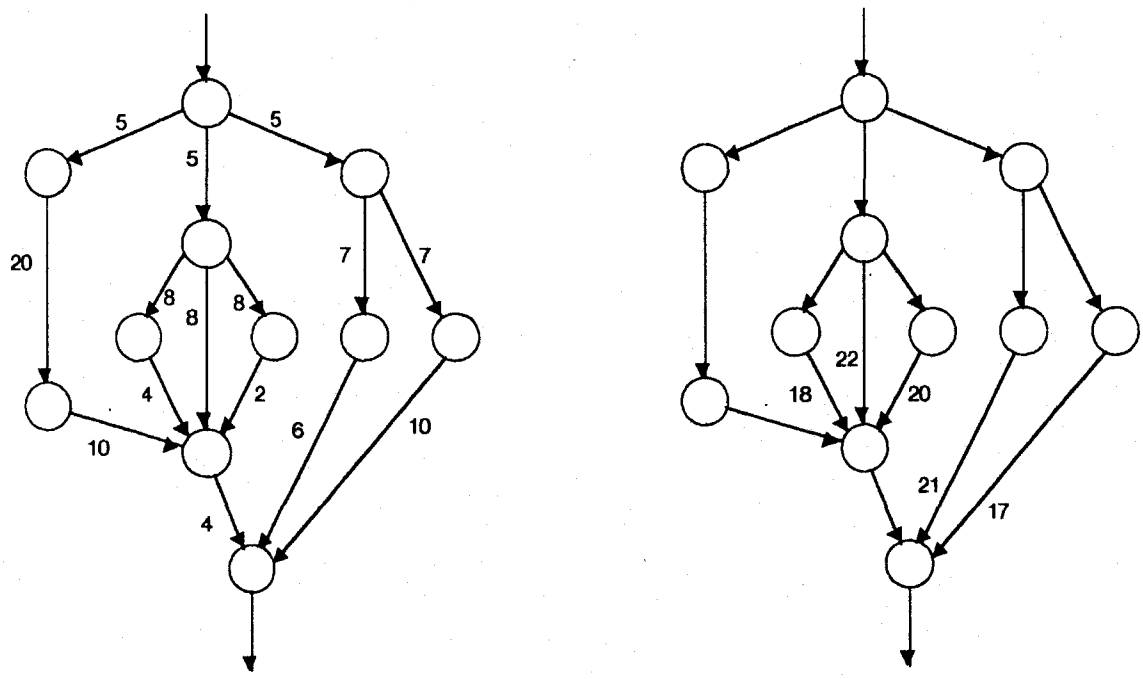
A balanced buffering graph can be easily constructed from the result of Algorithm 2.1, in which all the buffers are attached to the input edges of some multi-input nodes. In general, we say a buffering graph is *normalized*, if all its buffers are located on the input edges of some multi-input nodes.

Figure 2.5 shows the result of applying Algorithm 2.1 to a FDG. We will proceed to discuss the transformation of balanced buffering graph for a FDG in next section.

2.3 Transformation of a balanced buffering graph of a FDG

As stated in the introduction to this chapter, the way to balance a FDG is not unique. There may be several balanced buffering graphs for a FDG. For example, Figure 2.6 shows a different balanced buffering graph for the FDG G in Figure 2.5. We can see from the example that two balanced buffering graphs for the same FDG can be very different both in the position and in the amount of buffering introduced. So we need the following definition.

Definition Two buffering graphs are *equivalent* with respect to a FDG G if both are balanced



G

G'

G : a weighted FDG

G' : a balanced buffering graph for G which is the result
of applying algorithm 2.1 to G
total amount of buffering is 98

Figure 2.5.

---

buffering graphs for G.

It is possible to transform a balanced buffering graph for a FDG by adjusting the position and amount of its buffering. If the result of the transformation is an equivalent buffering graph for the same FDG , the transformation is an *equivalent transformation.* It is easy to prove the following two lemmas for equivalent transformations of a FDG.

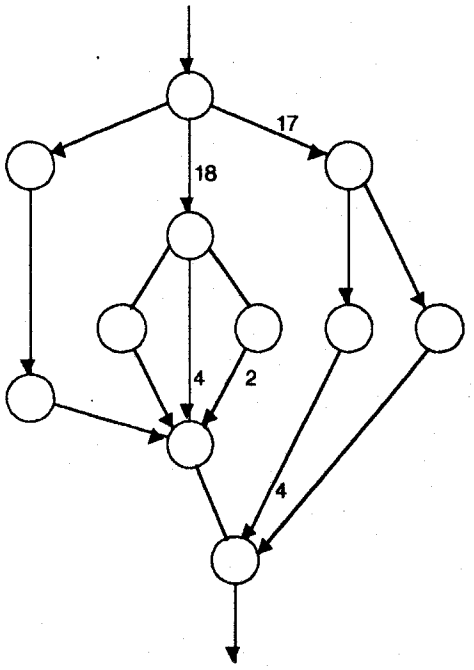
Lemma 2.1. If in a transformation of a buffering graph, the only changes are :

(1) Combine two or more buffers on the same chain together to form a new buffer which has the same stages as the sum of those buffers.

(2) move a buffer along a chain

(3) a combination of (1) and (2)

then the transformation is an equivalent transformation.



Another balanced buffering graph for G in Figure 2.5

The total amount of buffering needed is 45

Figure 2.6.

Note: A *chain* of a FDG is a directed path such that the incoming and outgoing degree for all nodes on the path is equal to one, except for the start and end nodes of the path. Furthermore, a chain is a *pretty* chain if the only buffer on the chain is located on the start edge (*i.e.* the edge adjacent to the start node).

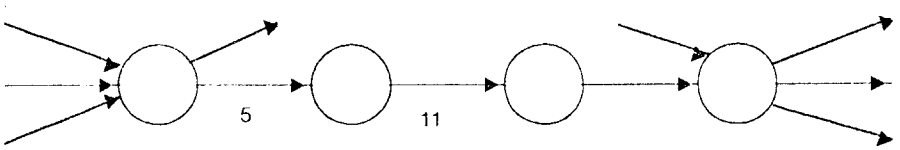
Figure 2.7 shows an example of applying lemma 2.1.

Consider the case where v is a node in a buffering graph which has two adjacent arcs (v,u_1), (v,u_2). The buffers associated with them are b_1 and b_2, with $b_1 \leq b_2$. We may transform the FDG by replacing the two buffers by two new buffers c_1 and c_2, where

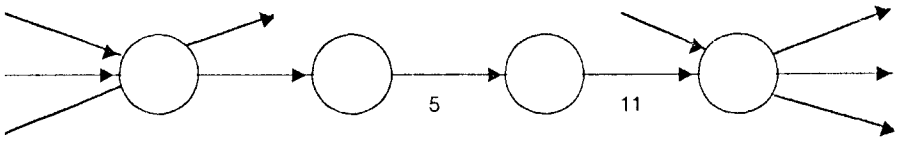
$$c_1 = b_1$$
$$c_2 = b_2 - b_1$$

The result of applying the above transformation is shown in Figure 2.8.

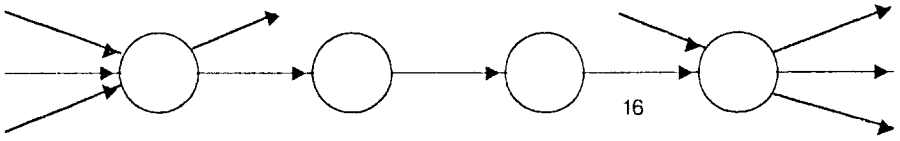
We can easily extend this transformation to the case where a node has multiple outputs. Let v be a node with n output edges (v,u_1), (v,u_2),..., (v,u_n). The buffering associated with each edge is b_1,



G1



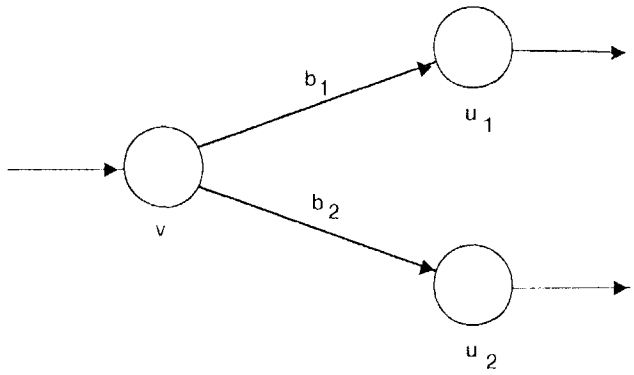
G2



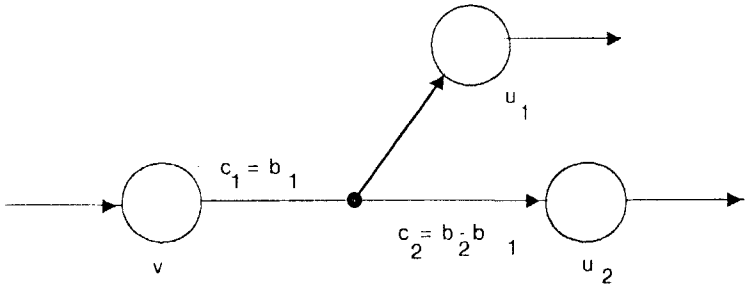
G3

G1,G2 and G3 are all equavalent, G2 and G3 are results of applying lemma 2.1 to G1

Figure 2.7.



G1



G2

G2 is the result of applying lemma 2.2 to G1

Figure 2.8.

---

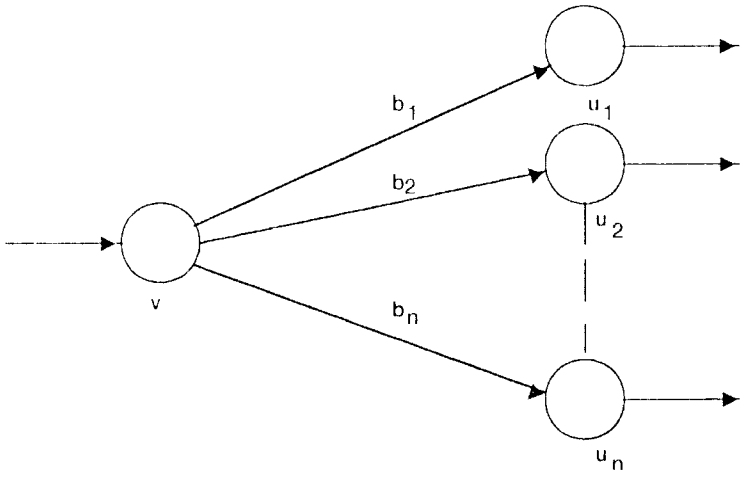
b_2,...,b_n, where

$$b_1 \leq b_2 \leq ... \leq b_n$$

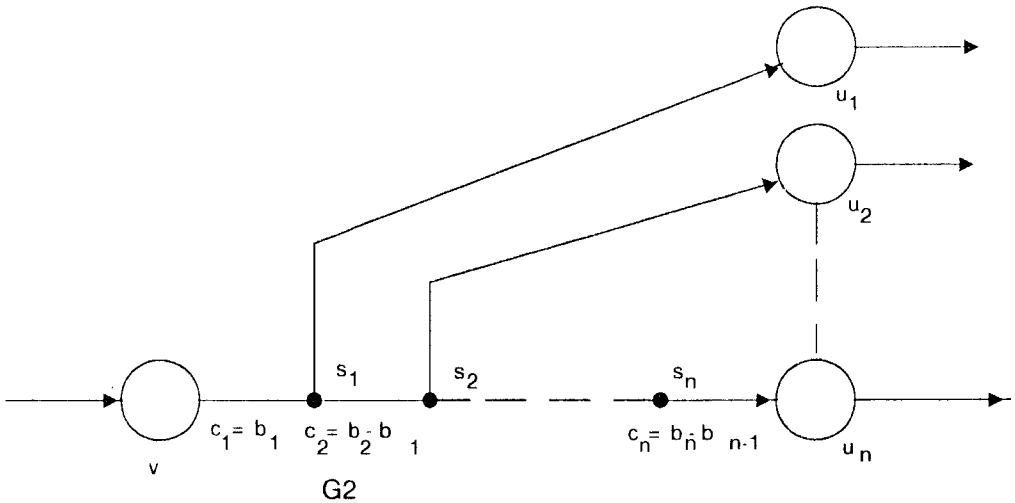
For convenience, we use following notation :

$$c_1 = b_1$$
$$c_2 = b_2 - b_1$$
$$\vdots$$
$$c_n = b_n - b_{n-1}$$

then we can do a transformation as shown in Figure 2.9. For simplicity, we will call this a *serializing transformation*. Sometimes we need to add some *switch node* to the buffering graph as shown in



G1



G2

G2 is the result of applying lemma 2.1 to G1

Figure 2.9.

Figure 2.9 (node s_1, s_2...). The resulting buffering graph is equivalent to the original one if we think of the switch node as one which has no effect when counting the cost of a path passing it. Now we have the following lemma.

Lemma 2.2 A serializing transformation is an equivalent transformation.

The proof of this lemma is obvious and is omitted

If we define the number of buffers by counting the edges on which buffers are introduced, then it is easy to see that, in general, the number of buffers may not be changed by applying a serializing transformation. However, the total amount of buffering is considerably reduced. In fact, the total amount of buffering reduced equals $(b_1 + b_2 + ... + b_{n-1})$. Applying the rule to an extreme case where $b_1 = b_2 = ... = b_n = b$, we end up with only one buffer where the amount of buffering equals b as shown in Figure 2.10. Both the number of the buffers and the total amount of buffering are decreased by a factor of n.

Finally we can transform a buffering graph by repeatedly using lemma 2.1 and lemma 2.2; the result is still an equivalent transformation. This observation leads to the following result.

Lemma 2.3 If a transformation is performed by using only the transformations defined in lemma 2.1 and lemma 2.2, then it is an equivalent transformation.

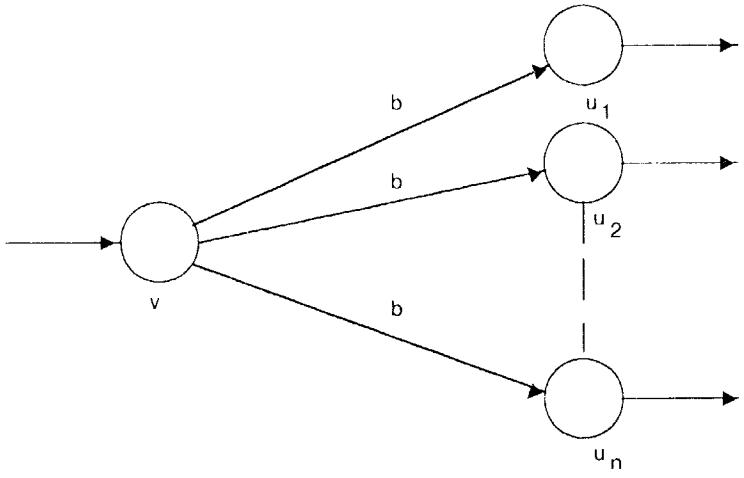
The above concepts and techniques for transformation are very useful in achieving optimal balancing of a FDG which will be discussed in the rest of the chapter.

2.4 Reducing the buffering of a balanced FDG --- An algorithm

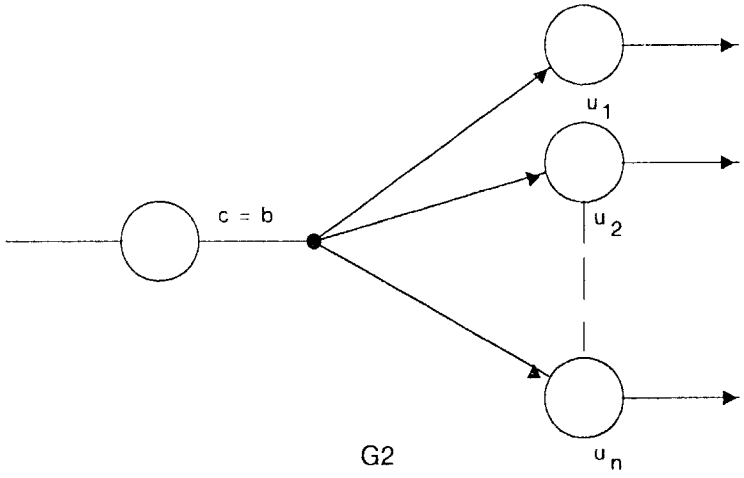
In this section, we will study how to use the tools developed in last section to reduce the buffering of a balanced buffering graph. An algorithm is given which can reduce the buffering needed to balance a FDG, but it is, in general, not an optimal balancing algorithm. An example will be given which illustrates that the optimum balancing of a FDG, even though the graph is acyclic, is intrinsically difficult.

In section 2.2 we have introduced an algorithm which can derive a balanced graph for a FDG. However, the amount of buffering introduced is, in general, much more than actually needed. Referring back to Figure 2.5, we note that the result graph can easily be transformed into the FDG in Figure 2.11, which uses less than 1/3 of total buffering. The way to implement the transformation is applying lemma 2.1 and lemma 2.2 to the balanced buffering graph. The result graph is balanced since the transformation is equivalent. However, it uses much less buffering than before both in the number of different buffers and the size of each buffer needed.

From the mechanism of the two lemmas, we know that applying the transformation of lemma



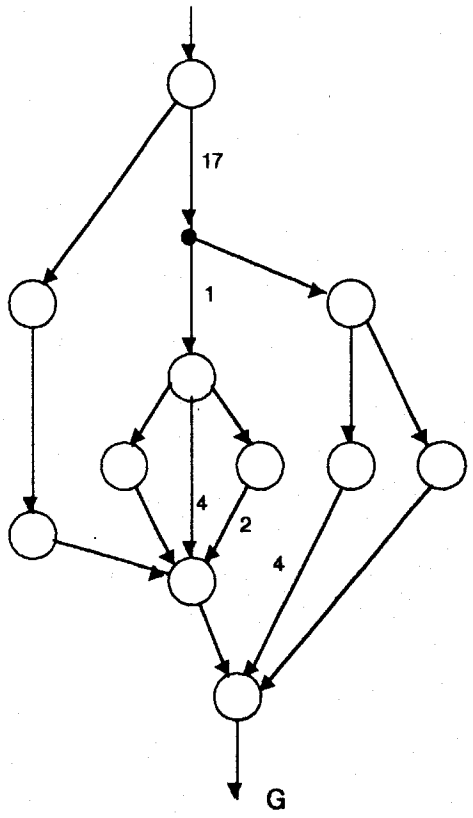
G1



G2

G2 is the result of applying lemma 2.1 to G1

Figure 2.10.



G : the result of applying algorithm 2.2 to the buffering graph in Figure 2.5
the total amount of buffering is reduced to 28

Figure 2.11.

2.1 can only cut the number of buffers needed, but the total amount of buffering is not changed. The serializing transformation defined in lemma 2.2 can decrease the total amount of buffering. In order to apply lemma 2.2 to a multi-output node, we often have to first adjust the position of some buffers by applying the transformation of lemma 2.1 until chains that start with this node become pretty chains. Note that after applying lemma 2.2, the graph is changed and the new graph can be processed again. So we should apply lemma 2.1 and lemma 2.2 to the graph repeatedly and properly to reduce effectively the buffering needed. Now we will present a simple algorithm which can achieve this goal.

Algorithm 2.2 An algorithm for reducing the buffering of a balanced FDG

input : A balanced and normalized buffering graph G1 = $\{V,E,B\}$, which is derived by applying algorithm 2.1 to a FDG G = (V,E,W).

output : A buffering graph G' = (V,E,B'), which is equivalent to G1 with respect to G. The buffering is reduced to a minimum under the restriction that no buffering can be propagated back across a multi-output node which is also a multi-input node.

Method Alternately applying the transformation defined by lemma 2.1 and lemma 2.2 until no more application can be made, without moving buffering across a multi-input node. Initially, M contains all multi-input nodes. A set S is maintained for all nodes to be tested if lemma 2.2 can be applied. S is also initialized to $\emptyset$. The entire algorithm is as follows :

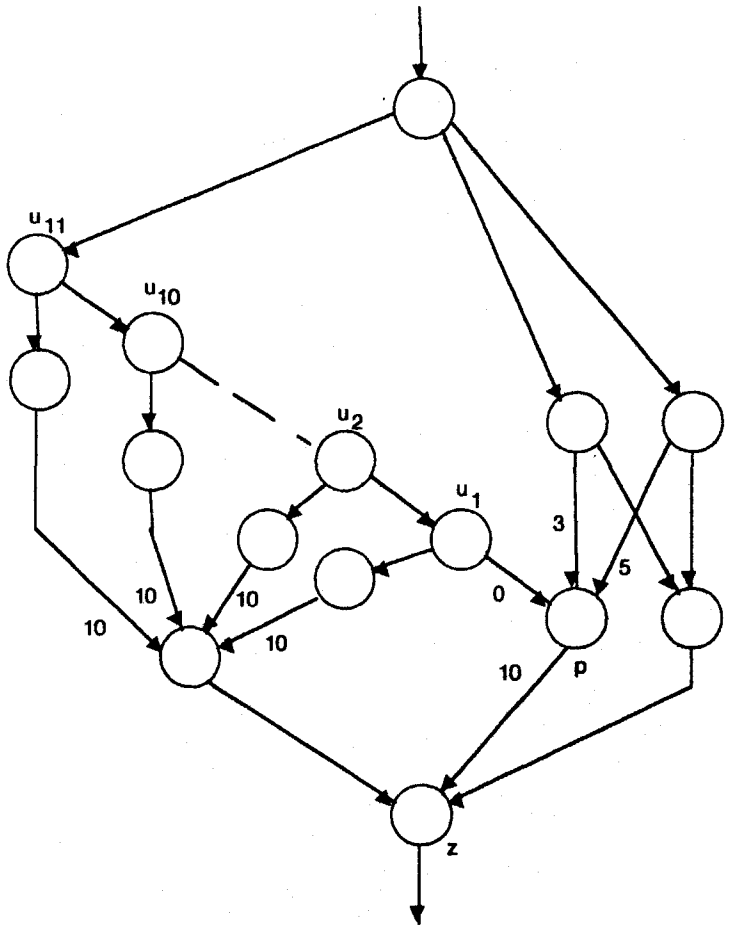
```
begin
1.    for all v ∈ M in G1 do
2.    apply lemma 2.1 to each chain with end node v until
      it becomes a pretty chain
3.    mark the start edges of those chains that are "pretty"
4.    S ← all start nodes of above pretty chains
5.        for all u ∈ S do
6.            if u is a multi-output node
              begin
7.                    apply lemma 2.2 to u
8.                    if it has input degree one than
                          begin
9.                        apply lemma 1 to the chain with end
                          node u until it becomes pretty
10.                       add start node of the chain to S
                          end
                  endif
              end
              endif
11.   return the result buffering graph G'
end
```

Intuitively, it is easy to see that the above algorithm will reduce the buffering needed to balance G, since the only reasonable way to cut the buffering needed is to apply lemma 2.2 to a multi-output node. However, the effects of applying lemma 2.2 to a node depend on the situation of the chains which have this node as their start nodes. Only if all such chains become pretty, can we succeed in cutting the maximum amount of buffering for the node under consideration. Note that in the algorithm we repeatedly apply lemma 2.1 and lemma 2.2 until they can not be applied anymore. Under the restriction that no buffering can be propagated back across multi-input nodes (line 8) the algorithm best reduces the buffering needed for balancing.

We may wonder whether it is reasonable to disallow movement of buffering across a

multi-input node. At first glance, it is reasonable, since moving a buffer back through a multi-input node, would increase the buffering needed by k times, where k is the input degree of the multi-input node. However, this is not always true ! Consider a part of a normalized buffering graph shown in Figure 2.12. Node p has three input edges and one output edge which has weight of 10. If we move the buffer back through p, each of its input arcs would gain a buffer of 10, so a total of 20 stages more buffering is introduced (see Figure 2.13). However, if we look at the problem more carefully, we note that the buffer introduced on (u_1,p) is useful for the serializing transformation of node u_1, u_2...u_{11}. This can reduce the buffering by a total of 100 stages. As a result we may save buffering up to 100 - 20 = 80 stages (see Figure 2.14).

In general, it is very difficult to predict the effect of propagating a buffer back through a



total buffering : 118

Figure 2.12.

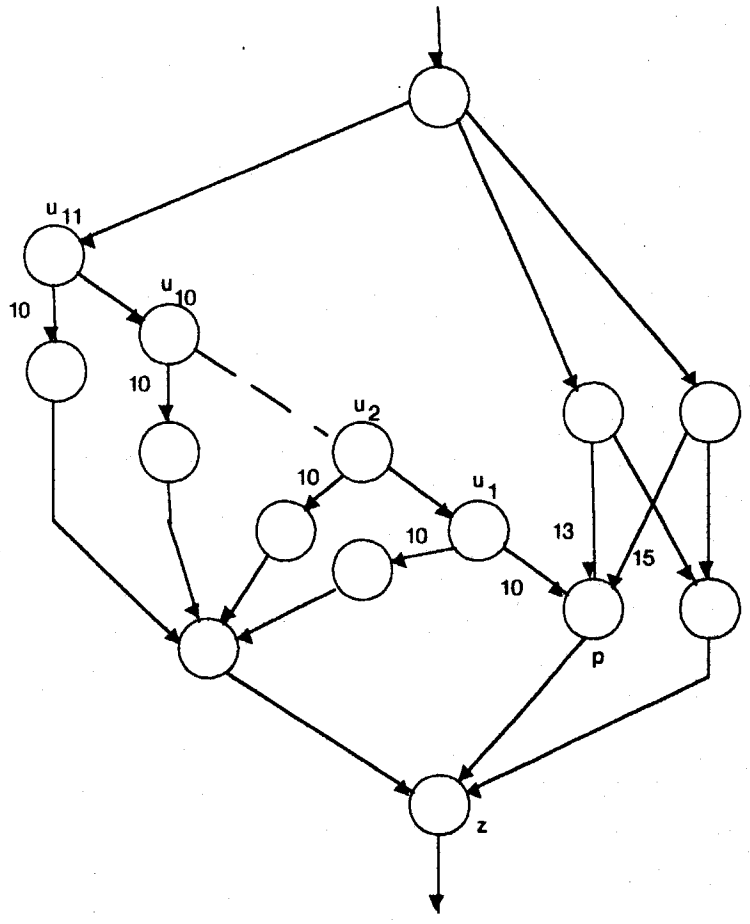
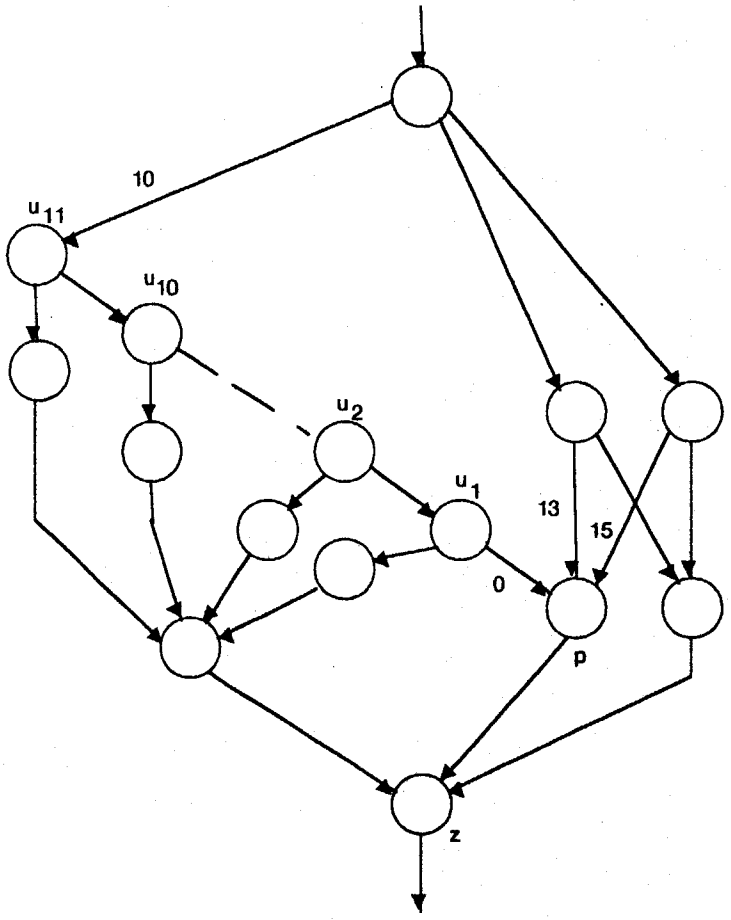


Figure 2.13.

---

multi-input node, since the influence may be far reaching. It is also difficult to decide how much buffering should be transferred. For example, if in Figure 2.12 the buffer on (p,z) has been changed to 50. Then, it is obvious that we should move an amount of 10, not all 50, buffering back. We can note that the above arguments are far from a mathematical reasoning. It does show, however, that the result of applying Algorithm 2.2 is not guaranteed an optimal one. It also shows that optimization of an arbitrary FDG, even if it is acyclic, remains a very hard problem. No good algorithm is known for this purpose. Fortunately, for a particular class of acyclic graphs a good optimization algorithm does exist. In this thesis I will introduce one algorithm for the optimal balancing of a well-structured FDG (to be defined in next section) and study its complexity.



total buffering : 38

Figure 2.14.

2.5 Well-structured FDG

In this section we will study the optimization problem more closely and introduce some important ideas; in particular, the class of well-structured FDG's will be defined.

Consider the structure of a balanced buffering graph G = (V,E,B). Assume M is the set of all multi-input nodes of G, and M' is the set of all multi-output nodes. We note that all buffers introduced using algorithm 2.1 are located on the input edges of nodes in M and the lemma 2.2 can only be applied to nodes in M'. Obviously those nodes in M and M' are the key nodes in an optimum balancing procedure, so we should study them more closely to facilitate our subsequent discussion.

Definition Let $u \in M$ and $v \in M'$, u is *directly reachable* from v if u is reachable from v and

there exists at least one path p from v to u such that no node $z \in M$ is on the path p between v and u (assume $z \neq u$ and $z \neq v$). Path p is called a *directed path* from v to u.

Note :

(1) If u is directly reachable from v, we write $v \mapsto u$.

(2) the set of all nodes which are directly reachable from v is denoted by DR(v).

For example in Figure 2.15, *c, k, u1, u2* are in DR(v). However, node i is not included in DR(v).

Sometimes we are interested in the subgraph consisting only of the parts of the graph which are covered exactly by the direct paths from v to u. Thus we have the following definition.

Definition Let $u \in M$, $V \in M'$ and $v \mapsto u$. Let G' = (V',E') be a subgraph of G such that V' and E' are the sets of all nodes and edges on some direct paths from v to u. Then G' is a *directly reachable section* from v to u and is denoted by SDR(v,u). The union of SDR(v,u) for all $u \in$ DR(v) is a *directly reachable graph* of v and is denoted by GDR(v).

The directly reachable section in above definition has interesting properties. If we delete node u from SDR(v,u) and attach new nodes one for each of its input edges, the resulting graph is a tree. This leads to the following lemma.

Lemma 2.4 Let $u \in M$, $v \in M'$ be two nodes of a buffering graph G = (V,E,B) and $u \in$ DR(v). Let G' = ($V'E'B'$) be SDR(v,u) and E'' is the set of all input edges of u in SDR(v,u). We construct a graph G1 = ($V1$,$E1$,$B1$) from G' such that node u is deleted and for each (z,u) in E'' a new end node is added. Then G1 is a directed tree with v as its root.

proof Assume G1 is not a tree. Then there must be at least a node $z \in V1$ other than u such that z is a multi-input node. However this is a contradiction to the assumption that G' = SDR(v).

In the following, G1 is named a *directly reachable tree* from v to u and is denoted by TDR(v,u). In addition, v is named the *root* of the tree. Sometimes we also call v the root of the corresponding directly reachable section. For example, Figure 2.15 also shows the directly reachable sections SDR(v, *u1*) and SDR(v, *u2*), and Figure 2.16 shows their corresponding trees.

In graph theory, trees usually have very nice properties which make some algorithms more straightforward. In an optimal balancing scheme, the same thing happens. As we will see, a good algorithm exists for optimization of a balanced buffering graph which is a tree (see OPTIMIZE-1 in the next section). Since each directly reachable section in a FDG corresponds to a tree, it can be optimized easily.

An interesting question is what is the relation between those directly reachable trees ?

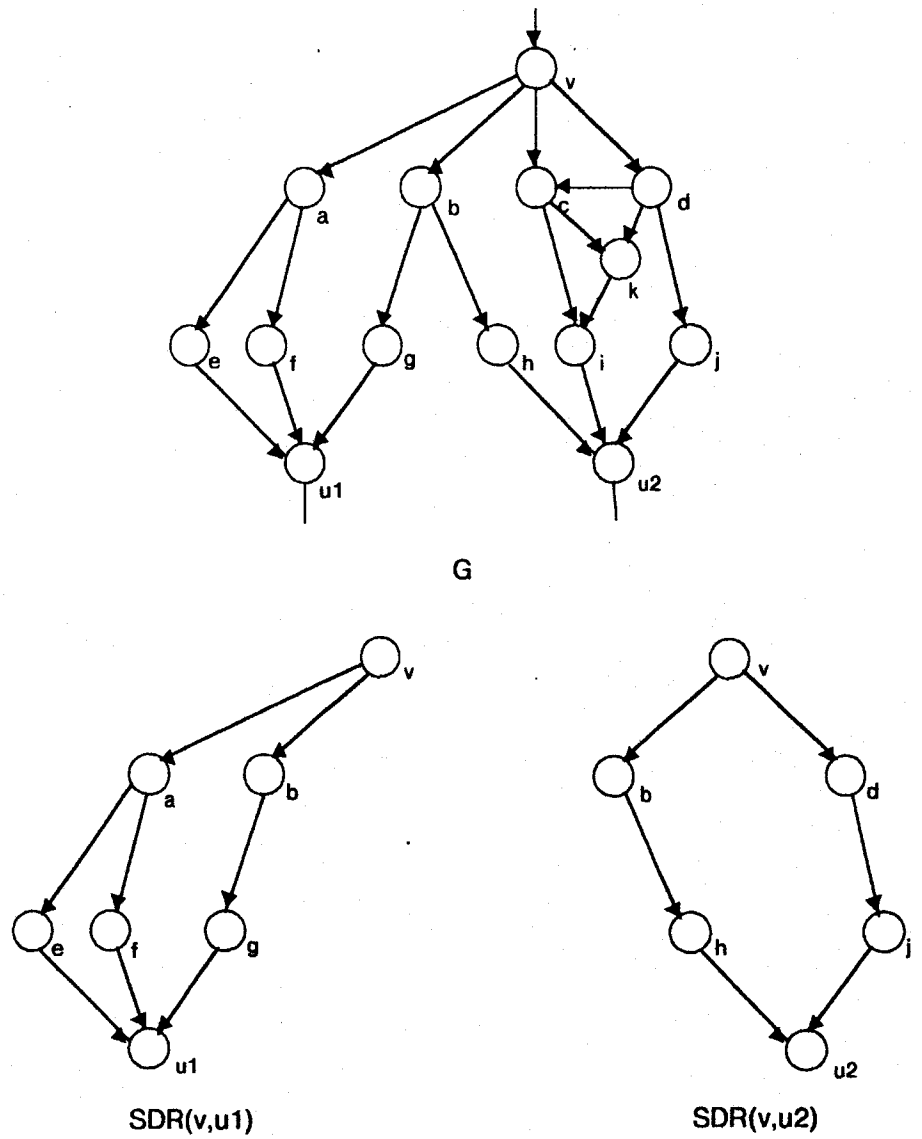


Figure 2.15.

Intuitively we might conjecture that the optimal balancing of one such tree is independent of that of others. If this is the case, the optimization for a balanced buffering graph can be performed separately for each individual tree. As a result, the optimal balancing algorithm can be made much simpler. Furthermore, the optimization can be done fully in parallel which will increase its efficiency in a multiprocessing environment. Unfortunately, this conjecture is not true for general cases. However, a particular class of graphs does have the nice property that the optimization of its directly reachable trees is independent of each other.

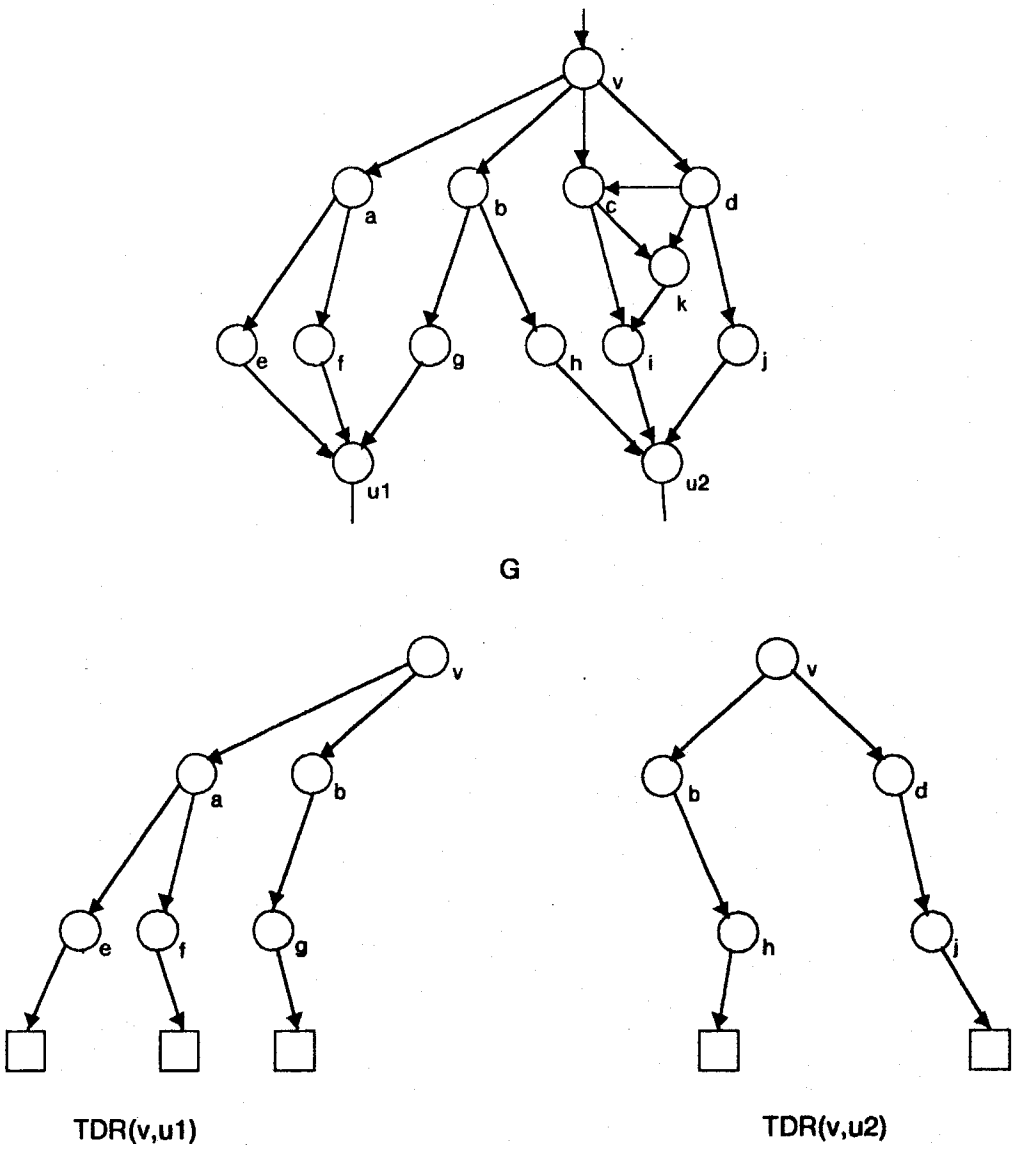


Figure 2.16.

Definition Let G1 = SDR(v,u) be a directly reachable section of v with respect to u in G and G1 = {$V1$,$E1$}. If for any path P that starts from v and ends with some node not in SDR(v,u), either $P \cap E1 = \emptyset$ or $P \wedge E1$ is a direct path from v to u, then G1 is called an *independent* directly reachable section. Furthermore, the corresponding tree is also *independent.*

Figure 2.17 shows some independent sections in a FDG.

We can define a relation between those independent sections as follows.

Definition Let G1 = SDR(v,u) and G2 = SDR(v',u) be two independent sections. G1 is *included* in G2 if G1 is a subgraph of G2.

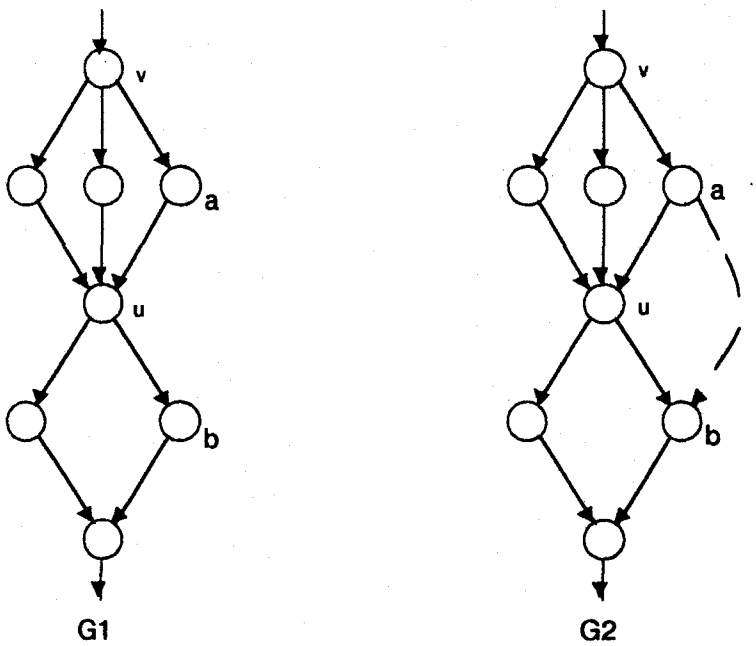
Now we can distinguish one particular class of independent sections, which we are interested in the optimization problem.

Definition An independent section is *maximum* if the graph has no other independent section which includes it.

For simplicity, we will frequently use *maximum independent section* to denote maximum independent directly reachable section as logn as no confusion is introduced.

Conceptually in an optimizing procedure, we can replace each maximum independent section in a graph by an edge, *i.e.* replace SDR(v,u) by an edge (v,u), where SDR(v,u) is a maximum

---



SDR(v,u) in G1 is an independent section

SDR(v,u) in G2 is not independent because of edge (a,b)

Figure 2.17.

independent section. This can be done because the optimization of an independent section does not depend on the rest of the graph. Furthermore, we can simplify the effects of such a section on the optimization of other parts of the graph by attaching an appropriate amount of buffering on the new edge. This procedure is called *m-reduction*, since it reduces multiple input edges of multi-input nodes, thus reducing the complexity of the result graph for further optimization. If the result graph of a *m*-reduction still contains some independent sections, then the m-reduction can be recursively applied to them. Now we can define a class of FDG for which we will present an algorithm for optimization.

Definition If *m*-reduction can be applied to a FDG G and its successive result graphs until no more multi-input nodes are left, then G is a *well-structured* FDG.

Figure 2.18 shows the examples of a well-structured graph and a graph which is not well-structured. Figure 2.19 shows the results of successive application m-reduction on the well-structured graph G in Figure 2.18, where, for simplicity, we omit the weights.

Obviously, the result of applying m-reduction to a well-structured graph is still a well-structured graph.

In the following section we will prove that a well-structured FDG can be optimized by a polynomial time algorithm.

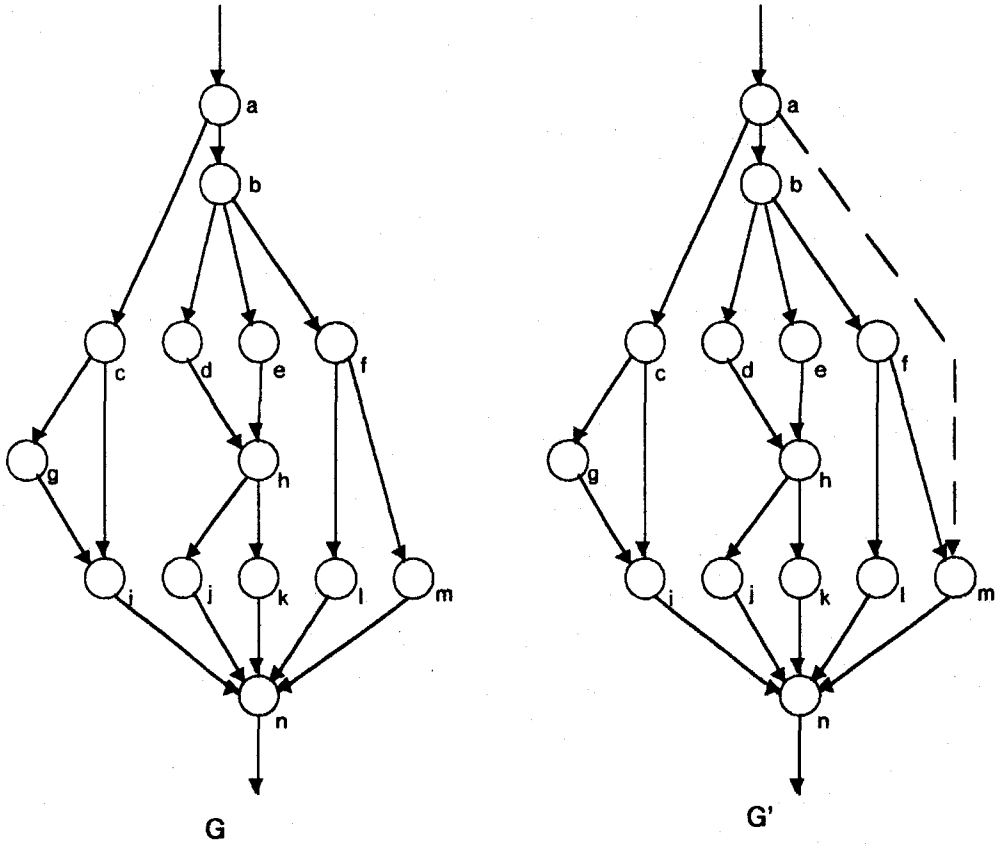
2.6 Optimization of well-structured FDG

In this section we will present an algorithm for optimization of a well-structured FDG. But before state the algorithm, we first prove some interesting results.

2.6.1 Some interesting properties of well-structured FDG

Let $u \in M$ and $v \in M'$ be two nodes in a balanced buffering graph G' for a well-structured FDG G, where *M and M'* are the sets of multi-input and multi-output nodes respectively. Furthermore, assume SDR(v,u) represents the maximum independent section of v. During an optimization process, the only way SDR(v,u) can interact with rest of the graph is through v, u. That is, during an optimization procedure, some buffers may be propagated in from u or some may be propagated out through v (as a result of applying lemma 2.2). Obviously, this does not change the balancing of SDR(v,u). By optimizing SDR(v,u) separately from other parts of the graph, one can specify the maximum amount of buffering which can be propagated out through v. This is defined in the following definition.

Definition Let SDR(v,u) be a maximum independent section in G' which is a balanced buffering graph of a well-structured FDG G. Assume SDR(v,u) is optimized separately from other parts of G'. The *specific weight* of SDR(v,u) is the maximum amount of buffering that can be



G is a well-structured FDG

G' is not a well-structured FDG

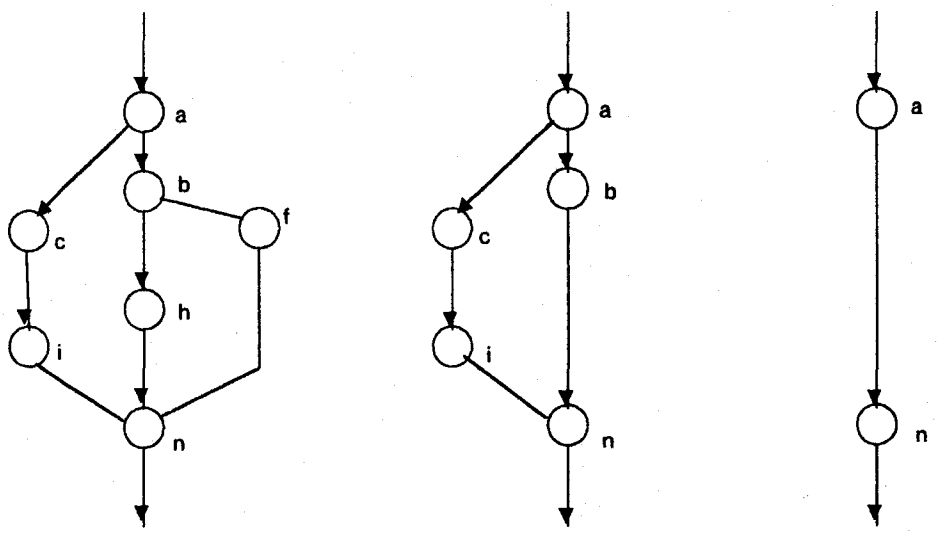
Figure 2.18.

---

propagated out through v while keeping it optimum balanced, without any buffering being propagated into the section through u.

Figure 2.20 shows the specific weight of a maximum independent section.

The specific weight of SDR(v,u) defined above does not depend on the optimization in other parts of the graph. It is the maximum amount of buffering which can be contributed by the section SDR(v,u) itself to the rest of the graph. Of course, during the optimum balancing process some buffering may be propagated in through u. But this has no effects on the balancing of the section, as soon as the section has already been balanced. It is easy to check that the same amount of buffering can also be propagated out of v. This is because SDR(v,u) is independent and v is the only port. So we can extend the m-reduction for a FDG to its balanced buffering graph as follows. Let G' be a balanced buffering graph for a well-structured FDG G. We apply m-reduction to G'. In addition to



G1,G2 and G3 are the successive results of applying m-reduction to G in Figure 2.18

Figure 2.19.

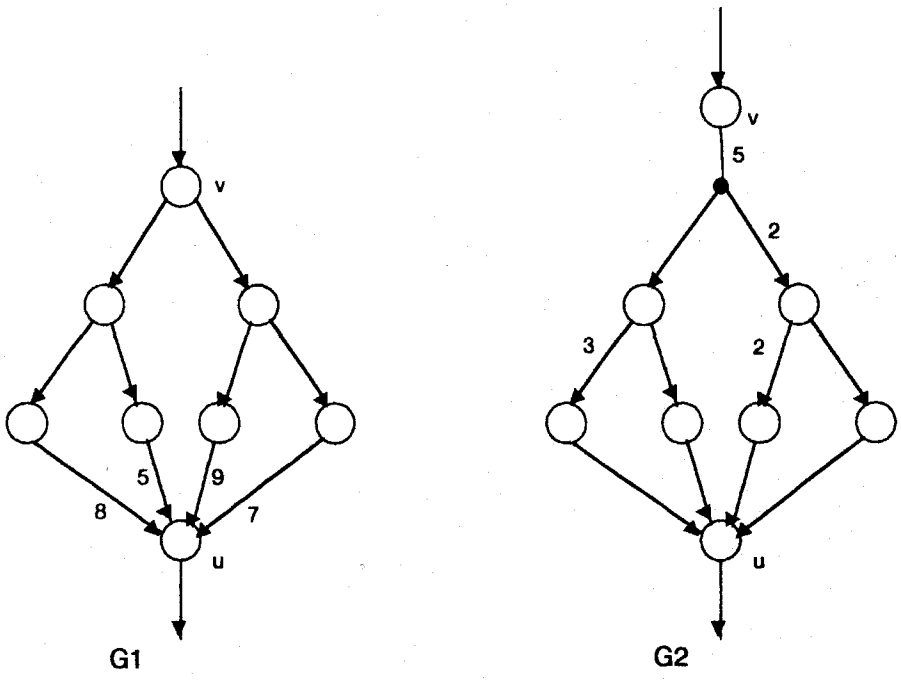
---

replacing each maximum independent section SDR(v,u) by an edge (v,u), we add a buffer on (v,u). The amount of buffering equals to the specific weight of SDR(v,u) in G'. If such procedure is applied to all maximum independent section in G', we say G' is *m*-reduced. The result graph behaves the same in term of the optimization of the rest of the graph. So we have the following lemma.

Lemma 2.5 Let G' be a balanced buffering graph of a well-structured graph G and let SDR(v,u) be a maximum independent section from v to u. Then SDR(v,u) can be replaced by an edge (v,u) with amount of buffering equal to the specific weight of SDR(u,v), without any influence on the optimization of other parts of the graph.

Obviously, we can reconstruct G' from the result graph of *m*-reduction simply by a reverse substitution. Of course, we should somehow remember, in each step of the m-reduction, the parts of the graph which have been m-reduced in order to perform the reconstruction later.

From lemma 2.5, we know that an *m*-reduction of one independent section in a buffering graph for a well-structured FDG does not influence the optimal balancing of other parts of the graph. So we can optimize a well-structured graph (say G) in the following way. We first perform optimization separately for each maximum independent section SDR(v,u) in G'(balanced buffering graph for G),



G1 shows an independent section SDR(v,u) of a FDG

G2 is the result of optimization of SDR(v,u)

the specific weight of SDR(v,u) is 5

Figure 2.20.

---

and get its specific weight. Then the *m*-reduction is applied to the graph. The reduced section of the graph can be somehow remembered, and the information is useful in reconstructing the original graph. After the whole buffering graph is *m*-reduced, we can apply a similar optimization and *m*-reduction process to the new graph. Since G is well-structured, the optimization and *m*-reduction processes can be carried out until there are no more multi-input nodes in the result graph. Finally, we can reconstruct the optimized buffering graph for G easily from the result. The algorithm for optimization of a well-structured FDG is presented in the next subsection.

2.6.2 An algorithm for optimization of a well-structured FDG

Now we can present the algorithm for optimum balancing of a well-structured FDG. Since we are interested in introducing adequate buffering, the algorithm is applied to a buffering graph instead of the FDG itself.

Algorithm 2.3 Optimization of a well-structured FDG

Input A balanced and normalized buffering graph G' = (V',E',B') which is the result of applying Algorithm 2.1 to a well-structured FDG G = (V,E,W).

Output A balanced buffering graph G1 = ($V1$,$E1$,$B1$) for G which is optimized.

Method Through each iteration of the main loop, procedure OPTIMIZE is called to optimize all maximum and independent sections in the graph separately. Then the *m*-reduction is applied to the result graph. Procedure NORMALIZE is to normalize the result graph so that it is ready to be fed as input to OPTIMIZE again. The repeated optimization and *m*-reduction process will be continued until the graph contains no more multi-input nodes. Then the optimized balanced buffering graph is constructed and returned as output.

begin

1. $G_0 \leftarrow G'$
2. OPTIMIZE (G_0)
3. let R be the result graph of line 2, apply m-reduction on R and the result is R'
4. **if** the set of multi-input nodes in R' is empty **then** **goto** 8 **endif**
5. Normalize (R')
6. $G_0 \leftarrow$ the result of line 5
7. **goto** 2
8. reconstruct the balanced buffering graph G1 for G
9. **return** G1 as result.

Following is the procedure OPTIMIZE(G) which can perform optimization on all maximum independent sections in G by optimizing their corresponding directly reachable trees separately. Let $M = \{u_1,u_2 \ldots\ u_m\}$ be the set of all multi-input nodes in G. Let $\mathcal{T} = \{T_1, T_2 \ldots T_s\}$ be the set of directly reachable trees in G, where each tree corresponds to a maximum independent section in G. The procedure OPTIMIZE-1(T) optimizes a directly reachable tree. Applying OPTIMIZE-1 to optimize all trees in $\mathcal{T}$ will eventually achieve the goal.

OPTIMIZE(G)

begin

10. **for** all i from 1 to *s* **do**
11. OPTIMIZE-1(T_i)
12. reconstruct the buffering graph G1 = ($V1$,$E1$,BI) from optimized trees in $\mathcal{T}$ by properly deleting all leaves in T_i.
13. return G1 as the result.

Note The "properly" in line 12 means delete all leaves of T_i and replace them by u_j, where T_i corresponds TDR(v,u_j) in G.

Following is the procedure OPTIMIZE-1(T) which performs the optimization of a tree T which is the input of the procedure. We assume L to be the set of leaves of T where $L = \{l_1, l_2 ... l_k\}$. The weight of each edge (x, l_i) is removed and assigned to the corresponding leaf l_i for each i and denoted by B(i). In addition, the weight of the leaves are sorted, and we have $B(l_1) \leq B(l_2) \leq ... \leq B(l_k)$. A list $B1$ is maintained for all multi-output nodes $v \in T$ (includes root), which is initialized to 0 for all such v. At the beginning, all multi-output nodes are unmarked, except the root which is marked "stop".

OPTIMIZE-1(T)

```
begin
14.   i ← 1
15.   C ← B(l_i)
16.   start from l_i and trace back until a node u
      marked "stop" is reached
      begin
17.       let P be the set of all multi-output nodes on the path
          from u to l_i , except u and l_i
18.       for each z ∈ P in descending order do
          begin
19.           mark z "stop"
20.           B1(z) ← C
          end
21.       put a buffer (C - B1(u)) on the last edge of the
              backward path.
      end
22.   i ← i + 1
23.   if i ≤ k goto 15
24.   apply lemma 2 to all nodes marked "stop"
end
```

Finally, we present the procedure NORMALIZE(G). This procedure will normalize a buffering graph G, *i.e.* it will end up with an equivalent buffering graph of G such that for each maximum independent section SDR(v,u) in G, all buffers are located on some input edges of node u. Let $\mathcal{S} = \{S_1, S_2 ... S_q\}$ be the set of all maximum independent sections in G. Let $\mathcal{R} = \{r_1, r_2 ... r_q\}$ be the set of roots of those sections in $\mathcal{S}$. The procedure NORMALIZE-1 normalizes a particular section in $\mathcal{S}$. Applying NORMALIZE-1 to all sections in G will eventually achieve the normalization of G.

NORMALIZE (G)

```
begin
25.   for i from 1 to q do
26.   NORMALIZE-1(r_i)
27.   return the result graph
```

The following is procedure NORMALIZE-1(r), which normalizes a directly reachable section SDR(r,u). An adjacency list L is maintained for each node in SDR(v,u). As in Algorithm 2.1, a set D is maintained to hold the edges which have been examined during the normalization process. The stack used in the algorithm is also similar to that in algorithm 2.1.

NORMALIZE-1 (r)

```
    begin
28.   for each z ∈ L[r] do
29.   if (r,z) ∉ D then
      begin
30.       add (r,z) to D
31.       push (top(stack) + B(r,z))
32.       if z = u then
33.           B(r,z) ← top(stack)
          endif
34.       NORMALIZE-1 (z)
35        pop
      end
      endif
      endfor
36. end
```

2.6.3 The correctness of the algorithm and its complexity

The key in proving the algorithm is to show that OPTIMIZE(G_0) (line 2) correctly implements the optimization for all maximum independent sections in G_0. By lemma 2.5, if this is proved, we can conclude that G1 constructed in line 8 is an optimized balanced buffering graph for G. To show OPTIMIZE is correct, it is enough to show that OPTIMIZE-1(T_i) (line 11) correctly performs the optimization for T_i. This is formulated in the following lemma.

Lemma 2.6 OPTIMIZE-1 optimizes its input directly reachable tree.

To make the proof simpler we introduce some additional notations.

Definition Let u,v be two nodes in a direct tree where u,v are multi-output nodes or leaves. If there exists a path from v to u, then v is called an *ancestor* of u; in addition if the path is a chain, then v is called an *immediate ancestor* of u. Also, if node v is an ancestor for both node u_1 and u_2, we call v a *common ancestor* node of u_1 and u_2.

Now let P denote the set of all multi-output nodes and leaves in a directly reachable tree. P can be partitioned into disjoint subsets $P_0,P_1...P_m$ such that

(1) $P_0 = \{r \mid r$ is the start node of G$\}$

(2) if node $v \in P_i$ and v is an immediate ancestor of u, then $u \in P_{i+1}$.

Furthermore, if $v \in P_i$ and $u \in P_j$ with $j \geq i$, then the *distance* between v and u is j - i and denoted by Dist(v,u). In words, P_j is the set of multi-output nodes or leaves which have distance j from root r.

Before beginning to prove the lemma, let us first do some analysis. Consider the loop from line 15 to line 23 in OPTIMIZE-1. We note that after each iteration through the loop one buffer, which corresponds to the weight of one leaf in T, is located on an output arc of some multi-output node $u \in P$. As a result, all buffering is put on some output arcs of multi-output nodes. In addition, it is easy to see for all multi-output nodes u, $Bl(u)$ would be assigned some weight during the execution of the procedure. We make the following claim :

Claim 1 After execution of main loop in OPTIMIZE-1, $Bl(z)$ equals the minimum buffering which can be introduced on the path from r to z.

Proof of claim 1 Let AN(z) denotes the set of all possible leaves which have z as its common ancestor and $l_w \in$ AN(z) be the leaf with minimum weight. Node z will only be assigned weight once in line 20. The algorithm guarantees that the weight assigned to z equals to $B(l_w)$. However, this is exactly the minimum amount of buffering which can be introduced between root r and z. So claim 1 holds.

Now we can proceed to make another claim.

Claim 2 All buffers introduced by OPTIMIZE-1 are the minimum amount buffers which can be introduced there.

Proof of claim 2 Let $x \in P_{j+1}$ and let $y \in P_j$ be an immediate ancestor node of x. Then from claim 1, $Bl(x)$ and $Bl(y)$ are respectively the minimum buffering which can be introduced on the path from r to x and r to y. Let l_w be the leaf with minimum weight in the set AN(x). Clearly we have $B(l_w) = Bl(x)$. Now if $B(l_w) > Bl(y)$, then we are sure some buffers which are on a leaf in AN(y) but not in AN(x) has already been propagated to or through y. But the algorithm guarantees that, a buffer of $Bl(y) - B(l_w)$ will be put on the output edge, say e, which is the start edge on the path from y to x. From claim 1, this is the minimum amount of buffering that can be introduced there. On the other hand, if $B(l_w) = Bl(y)$, it means the leaf l_w is the one in AN(y) which has minimum weight. According the algorithm, it would be propagated back through x and y. Hence, no buffer can be put on the output edge e of y. This also satisfies claim 2. It is easy to see that the algorithm will examine all nodes in P and add buffers on its output edges when necessary. Therefore claim 2 must hold.

Finally, applying lemma 2.2 will optimize the buffering on output edges of each multi-output nodes. We note that applying lemma 2.2 to one such node has no influence on other nodes. That is no buffering can be propagated out across the node, since one of the output arcs must have zero buffering. Thus we have proved lemma 2.6.

Now it is easy to see that the procedure OPTIMIZE(G) will optimize all directly reachable trees separately, since each tree in G will be optimized correctly by OPTIMIZE-1.

From lemma 2.5, applying *m*-reduction to the result of OPTIMIZE (line 3) will not affect the optimization of each separate tree. The result of reduction will be normalized by calling procedure NORMALIZE, which can make the graph ready for the next cycle of applying OPTIMIZE. If the algorithm eventually terminates, the graph reconstructed in line 8 must be an optimized buffering graph.

One last thing remaining to be proved is the termination of the algorithm. Obviously, line 3 takes finite time steps since the graph to be processed is finite. It is also easy to show NORMALIZE (line 5 and line 25-36) takes finite time steps for the same reason. To prove OPTIMIZE(line 2) takes a finite amount of time requires a few words. Since *s* (line 10) is finite, we need only to show OPTIMIZE-1 takes finite steps. To show this is true we note that *k* (line 23), the number of leaves in the tree, is finite. Through each iteration of the main loop in OPTIMIZE-1 (line 15 to 23) the weight (*i.e.* amount of buffering) of one leaf is propagated back until a node marked "stop" is met. This only takes a finite amount of time since the longest distance it can travel is bounded by *h*, where *h* is the height of the tree. Since *h* is finite, we can deduce that the OPTIMIZE-1, and hence OPTIMIZE, will terminate in finite time.

Now we go back to see the main algorithm. Since each time a *m*-reduction is performed, the total number of input-edges of all multi-input nodes in the graph is decreased by at least one. We know that the graph is well-structured, so the m-reduction can be carried out until no more multi-input nodes exist. Thus the test in line 4 will eventually go to line 8, and the algorithm will terminate. So we have the following theorem.

Theorem 2.4 Algorithm 2.3 correctly implements the optimization of the buffering graph for a well-structured FDG.

To derive the time complexity of the whole algorithm let us first prove the following claim.

Claim 3 OPTIMIZE(G) takes $O(n^2)$ steps to optimize all maximum independent sections in G.

Proof of claim 3. In OPTIMIZE, procedure OPTIMIZE-1 is called *s* times where *s* is the number of maximum independent sections (and their corresponding trees) in G. As we can see, the time spent on each iteration of the main loop in OPTIMIZE-1 is proportional to the multi-output nodes that it passes during the backward traveling. However this is the number of nodes which has been marked during the iteration. We know that a node can only be marked once. Moreover, all multi-output nodes would be marked by some iteration of the main loop. Hence, the time spent for the main loop is $O(n_i)$, where n_i is the number of nodes in T_i. In line 24, lemma 2 is applied to all nodes marked "stop". For a node with *d* output degrees, applying lemma 2 requires $O(d)$ steps.

Hence line 24 requires a total of $O(e_i)$ steps, where e_i is the number of edges in T_i. Since T_i is a tree, we have the important property : $e_i = n_i - 1$. Thus, one call to OPTIMIZE-1(T_i) costs $O(e_i)$ steps.

In the algorithm, OPTIMIZE would be called for each T_i in the G'. Since all those T_i's are mutually independent, the time for execution of the whole procedure is $O(e)$, where e is the total number of edges in G. Expressed in n, the number of nodes in G, we have the result of claim 3.

Now we can analyze the whole algorithm. Clearly, line 3 in the main loop of the algorithm takes at most $O(n^2)$ steps since the number of maximum independent sections is less than the number of edges in the graph. Also it is easy to see that NORMALIZE in line 5 will use $O(n^2)$ steps. This is because NORMALIZE-1 will use $O(e_i)$ steps to normalize a directly reachable tree T_i, where e_i is the number of edges in the tree (note that each edge in the graph is computed only once).

As we mentioned before, through each iteration in the main loop, the total number of input edges of all multi-input nodes in the graph will be decrease by at least one. Since the loop will terminate when there are no more multi-input nodes left, we conclude that the number of iterations executed for the loop is at most $O(n^2)$. Consequently, we can deduce that the complexity of the whole algorithm is $O(n^4)$. So we have the following theorem.

Theorem 2.5 Algorithm 2.3 for optimizing a buffering graph for a well structured FDG with n nodes requires $O(n^4)$ steps.

By combining Theorem 2.4 and Theorem 2.5, we can derive the following theorem.

Theorem 2.6 There exists a polynomial time algorithm which can balance and optimize a well-structured FDG.

3. The Mapping Scheme for for-all and for-iter array Operation Constructs

In chapter 2, we have introduced the FDG model, as a powerful tool for studying the mapping of array operations on a static data flow machine. The analysis and results apply only to the class of well-behaved VAL programs, for which we did not give a precise definition. Intuitively, in a well-behaved VAL program, all array values are produced or consumed by **forall** or **for-iter** constructs, and the internal structure of such constructs can be mapped into fully pipelined data flow graphs. In general, however, the mapping of **forall** and **for-iter** constructs into fully pipelined data flow machine code is not always possible. Moreover, the mapping itself, if it is possible at all, is not a trivial matter. In this chapter, we will define the class of **forall** and **for-iter** constructs which can be effectively mapped into fully pipelined data flow graphs. It is beyond the scope of this thesis to give sound and complete definitions which cover all possible cases. The definitions we give are easily understood and applicable to programs widely used in numerical computations. The mapping of **forall** constructs is, under some restrictions, relatively simple. We will develop good schemes for such mapping, including a discussion of implementing FIFO buffers on a static data flow machine. It is more difficult to deal with the mapping of **for-iter** constructs. The corresponding data flow graph, in general, is not acyclic. We have not found an algorithm which can balance an arbitrary data flow graph with cycles to a fully pipelined fashion. Fortunately, for an important class of **for-iter** which are quite often used in numerical computation programs, we do have a good mapping scheme. This is the class of **for-iter** constructs expressing certain recurrence relations which have associated functions with certain composition properties. For such recurrence functions, we can construct an elegant and efficient solution. We will show that it is particularly suitable for implementation on a static data flow machine by taking advantage of software pipelining.

Finally, using the concepts and results developed in this chapter, we will present a more rigorous definition of well-behaved VAL programs.

3.1 The strategy of mapping forall constructs

One of the most important forms of concurrency available in VAL comes from the **forall** expression. There are two types of array operations which can be expressed in terms of **forall** constructs. One is the array construction operation, where the results of individual computation are made components of an array which will be the value returned by the whole expression. Another type is array reduction operations; an example is the adding of all elements of an array concurrently to produce a single result value which is the sum. For our purpose, we are only interested in the first type of **forall** construct which represents array construction operations.

Typically, a forall construct has three basic parts : *range specification, environment expansion* and *result accumulation.* The range specification of a **forall** is a contiguous integer interval or a cross product of contiguous intervals that defines the dimension and range of the array being constructed. The body of the **forall** expression contains an environment expansion which will be executed for each

element of the array constructed by the **forall**. The body also contains an accumulation part which is located after the key word **construct**. The following is an example of a **forall** array operation construct.

```
B : array[array[real]] : =
      forall      i in [1, m+1], j in [1, n+1]
         G : integer : = if A[i,j] = 0 then 1 else A[i,j] endif;
         H : real : = 1/G + 2;
      construct
         if i = 0|i = m + 1|j = 0 |j = n + 1 then A[i,j]
         else G + H**2 + (A[i,j-1] + A[i,j+1]
                        + A[i-1,j] + A[i+1,j])
         endif
      forall
```

This **forall** expression requires as input an array A and constructs a new array B. The dimension and bounds of array B is specified by the first line of the expression which is the range specification section. The environment expansion section defines two local variables G and H. This section executes once for each array element in the range specified by identifiers i and j. The expression following the key word **construct** is the accumulation section which is also evaluated once for each array index. It does not impose any extra sequencing constraints or any particular form of run time concurrency. One thing worth mentioning is the index expressions i-1, i+1, j-1, j+1. In typical numerical applications, many algorithms manipulate arrays of values with each of its indices denoting a point in a physical area or space. The value of an array element represents some parameter associated with the corresponding point. New values of an element are often determined by the original values of its adjacent elements. As a result, indices such as i-1, i+1, etc. are extensively used in related VAL programs. In mapping of **forall** constructs, we should take this into consideration and work out good solutions.

To exploit the embedded parallelism, two basic schemes for mapping **forall** array constructs onto the static data flow machine architecture exist. The first, known as the *parallel scheme*, generates a copy of the program body for each array element as shown in Figure 3.1. In this scheme, many copies of the **forall** body may execute concurrently. Of course, this scheme can be implemented only if the range is known when the machine code is loaded into the PE's. The second scheme, *the pipeline scheme*, generates the array elements by operating the body of the **forall** construct in a pipelined fashion. The program could be compiled to treat the environment expansion section and the accumulation section together as a pipeline, feeding the successive values of the array range indices into the front of the pipeline as shown in Figure 3.2. A third scheme, which combines the above two, is also possible. In this scheme the whole array is partitioned into several subsections. All subsections can be executed in parallel and each individual subsection is executed in pipelined fashion.

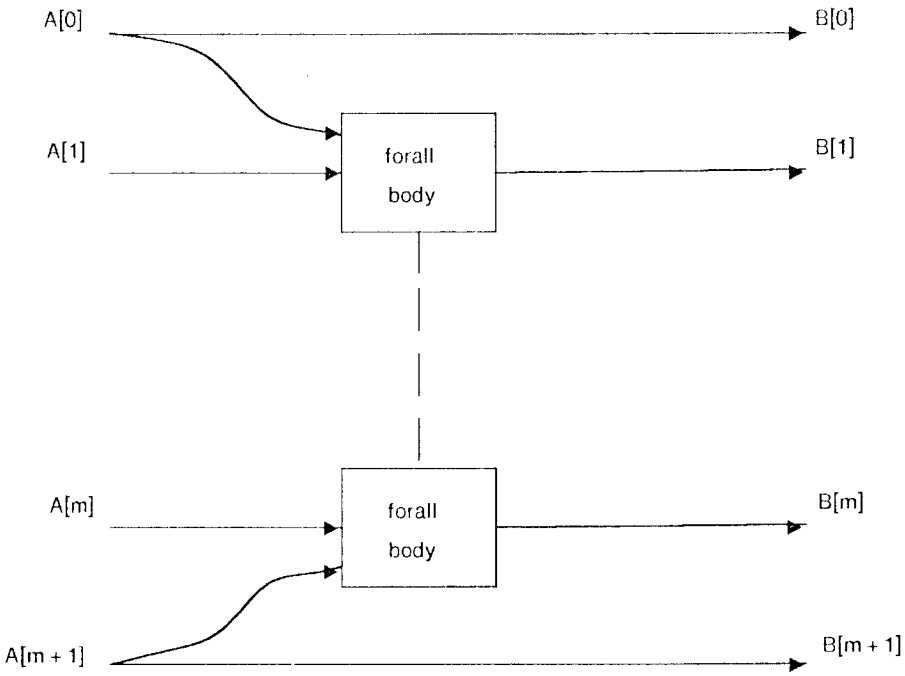


Figure 3.1.

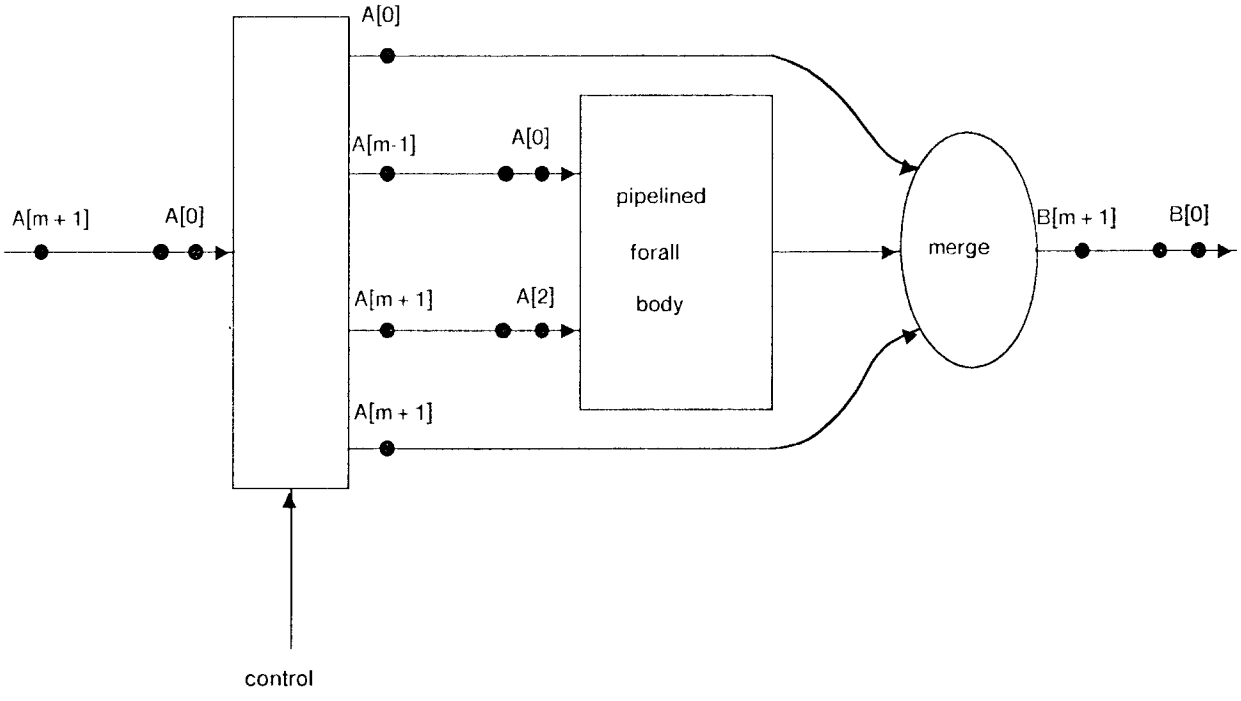


Figure 3.2.

In the following we will concentrate to the pipeline scheme which is the major interest of this thesis.

3.2 Well-behaved forall constructs

In this section we will study the structure of VAL **forall** array operation constructs more carefully in terms of the pipelined mapping. Based on the study, a class of **forall** constructs is defined which can be mapped into fully pipelined machine code on a static data flow machine. Some concepts and results presented in this section will be helpful in understanding of the concept of well-behaved VAL programs introduced in chapter 2.

Not all VAL expressions can be mapped into fully pipelined data flow programs. For example, we do not have, in general, good algorithms for transforming a data flow graph with cycles into a fully pipelined one. As stated before, a **forall** construct is composed of several sections each of which is a VAL expression or consists of VAL expressions. Therefore, in order to define the class of **forall** constructs which can be mapped into fully pipelined machine codes, we should first study the similar nature of more primitive VAL expressions. Now let us begin by introducing some new concepts.

Definition A *primitive well-behaved* VAL expression (will be denoted by PWE) is recursively defined as follow :

(1) A constant is a PWE

(2) An identifier is a PWE

(3) If E1 and E2 are PWEs, then (E1 op E2) is a PWE, where op is an arithmetic or relational operators of VAL.

(4) If A is an identifier that denotes a one-dimensional array, then A[I] is a PWE. Furthermore, if E is a PWE, then A[J : E] is also a PWE. This can be extended to multidimensional arrays.

(5) If E is a PWE, then f(E) is a PWE where f is a VAL reserved word for a system defined function such as SIN, COS, etc.

(6) If E1, E2, E3 are PWEs, then **if** E1 **then** E2 **else** E3 **end** is a PWE.

(7) Let E be a VAL **let-in** construct expressed as

Let
    ⟨*env-expansion*⟩
in
    E0
endlet

If ⟨*env-expansion*⟩, the environment expansion part, contains only PWEs and E0 is also a PWE, then E is a PWE.

(8) a **tagcase** expression is a PWE, if all its arms contain only PWEs.

(9) Nothing else is a PWE

We are not interested in design and implementation of a parser for PWEs in VAL, therefore the grammatical strictness and syntactical completeness of the above definition is not our major concern. The importance of our notion of PWE is to put adequate restrictions on program structure so that a good scheme can be developed to map the program into fully pipelined data flow graphs.

We claim that a PWE corresponds to a data flow graph with no cycles. The correctness of this claim can be easily derived from the above definition. The PWEs constructed by (1) through (4) are clearly cycle free, from the tree structure of such expressions. To show it holds for case (5), we need to make the following assumption : all system defined functions can be mapped into acyclic data flow graphs. Case (6) is also true, since the translation of a VAL **if-then-else** expression into acyclic data flow graphs is straightforward. A simple **if-then-else** in VAL has the form :

if x = a **then** f(x) **else** g(x) **endif**

Figure 3.3 shows how to translate it into acyclic data flows, assuming that the body of functions f and g can be mapped into such graphs. Cases (7) and (8) are also obvious if we notice the recursive nature of the definition. As a result, we have the following lemma.

Lemma 3.1 A primitive well-behaved VAL expression can be mapped into a data flow graph with no cycles.

The transformation of an acyclic data flow graph into a fully pipelined one has been studied in

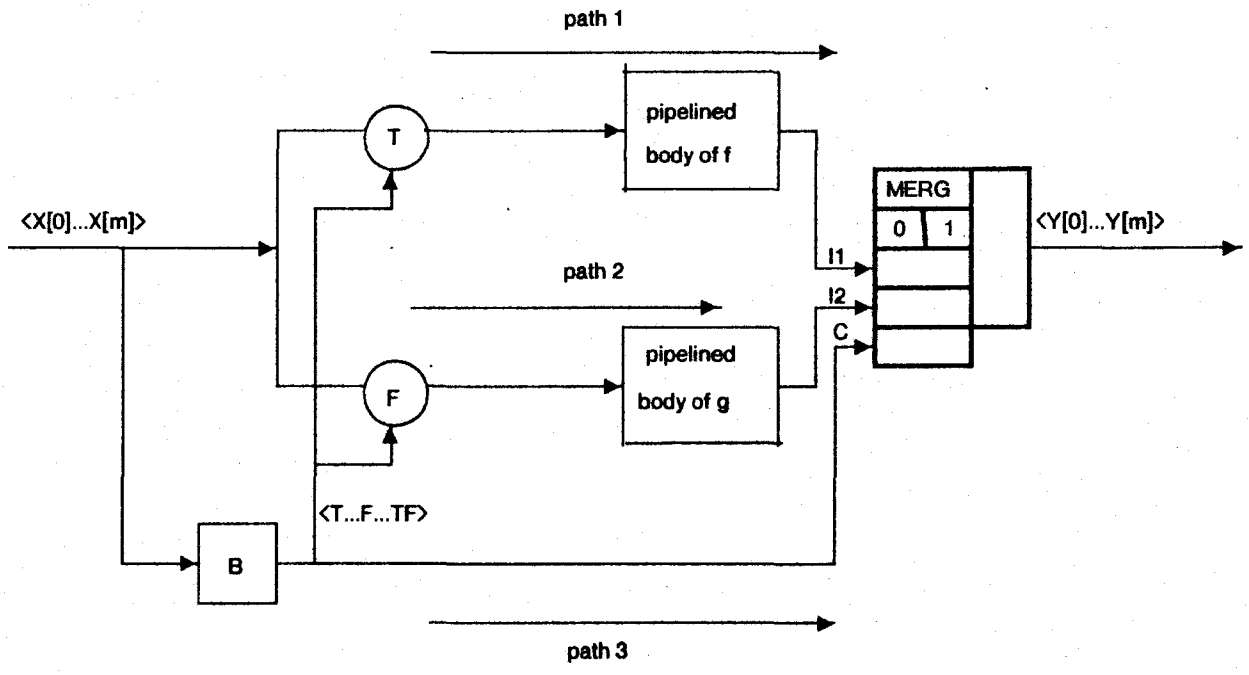


Figure 3.3.

[14]. The basic approach and technique developed in chapter 2 for balancing a FDG can also be applied to a machine level data flow graph. To show that a graph denoting a PWE can be balanced to achieve full pipelining we should go back to its recursive definition. Again, the balancing of cases (1) through (4) are straightforward. For case (5) we also need an assumption that all system defined VAL function can be mapped into fully pipelined data flow graphs.

A few remarks are needed to justify case (6). Let us examine a simple example just described. A full pipelined data flow graph corresponding this expression is shown in Figure 3.4. The key to understanding the balancing of such a graph is the function of the merge gate. Notice that the firing rule for a merge gate is quite different from other operators. A merge gate, as shown in Figure 3.4 has three inputs : two data inputs I1, I2 and a control input C. If a true value is present on C and a data value is also present on I1, the gate fires and the value on I1 is forwarded to the result arc, while leaving the second operand on I2, if any, untouched. Conversely, if the control value is false, the second operand value, if present, will be used and gated to the result arc. As we can see, for a merge gate to fire, the control operand must be present. Furthermore, one of the data operands should be present according the value of the control operand. Recall that the balancing algorithm requires that different paths from the input to the output should have the same length. In the case of Figure 3.4, this means we should balance the two data paths (path 1 and path 2) and control path (path 3). If the two data paths have different lengths, the balancing algorithm should equalize all three paths by adding appropriate buffers.

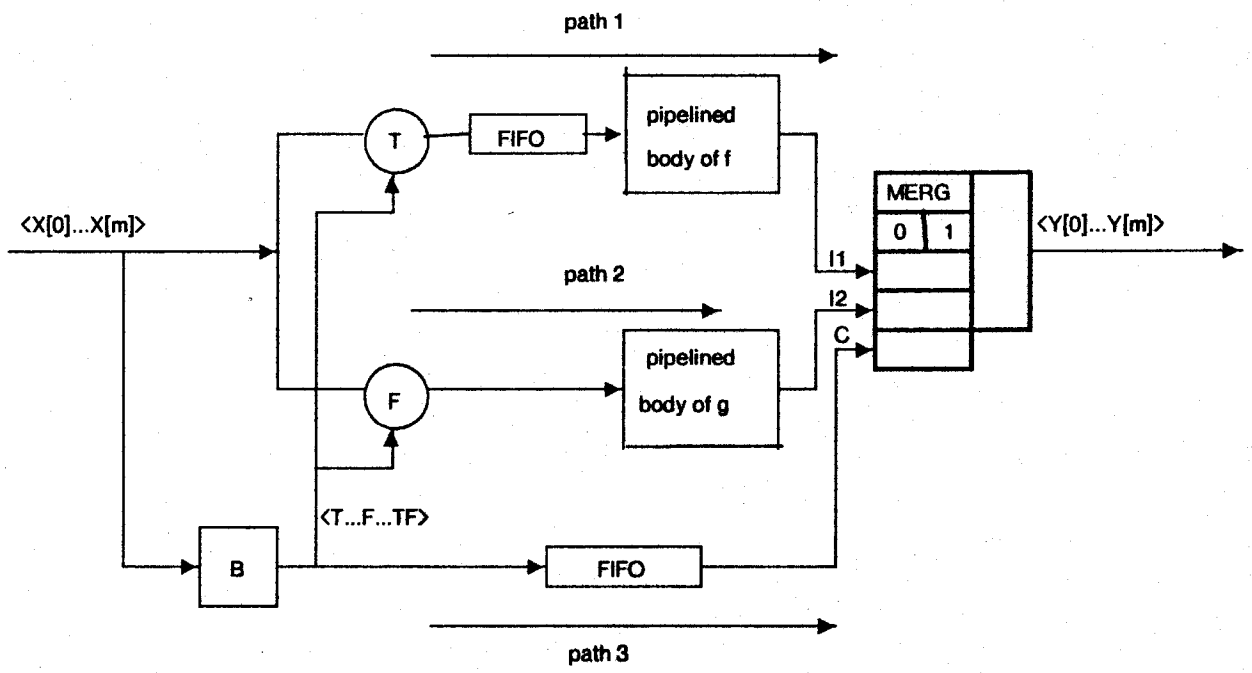


Figure 3.4.

For cases (7) and (8), the balancing can be achieved because of the recursive nature of PWE's definition. Consequently, we come up with following theorem which is very important in the rest of this thesis.

Theorem 3.1 A primitive well-behaved VAL expression can be mapped into a fully pipelined data flow graph.

Based on the above analysis and results, we can proceed to study the class of **forall** array operation constructs which can be mapped into fully pipelined data flow graphs. We first introduce the following concept.

Definition A **forall** array construction expression is *well-behaved*, if its environment expansion part and accumulation part contain only PWEs.

To facilitate the study of a well-behaved **forall** construct, we can restructure the expression. Let us study the following example :

```
A : array[real] : =
   forall  i in [1, n]
      x1 : real : = g1(y1,y2...ym);
      x2 : integer : = g2(y1,y2...ym);
         .
         .
      xk : real : = gk(y1,y2...ym)
   construct
      f(x1,x2...xn)
   endall
```

Figure 3.5 illustrates the possible structure of the above expression. E1 and E2 are PWEs which correspond to the environment expansion section and accumulation section respectively. The input to the whole expression is y1 through ym. they are fed in E1 and E2. The array index values <1,2...n> are also produced and fed into E1 and E2. Eventually, array A is being produced as the result value. Obviously, the above expression can be restructured as shown in Figure 3.6, where E1 and E2 are combined into E3. Since E1 and E2 are PWEs, we conclude that E3 is also a PWE. From Theorem 3.1, E3 can be mapped into fully pipelined data flow graphs. Consequently, the whole **forall** expression is fully pipelined. Note that in the above argument we assume that the range specification part can produce a sequence of array index values in fully pipelined fashion. In addition, we assume a scheme exists to construct the result array. We will discuss these issues through examples in the next section.

Although the example we give is a **forall** expression which returns a one-dimensional array, similar results can also be extended to **forall** expressions which produce multi-dimensional arrays.

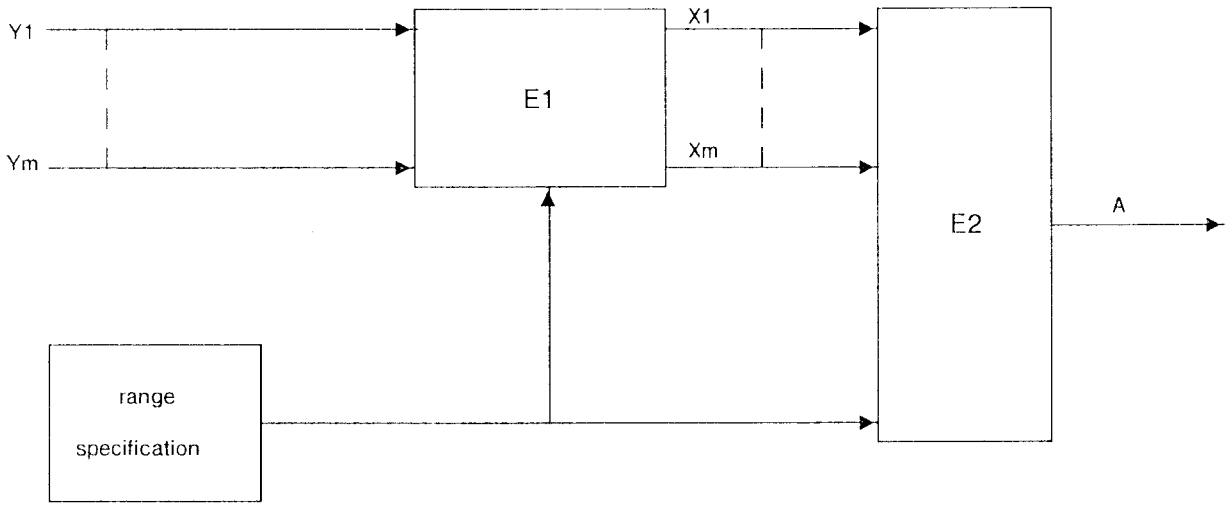


Figure 3.5.

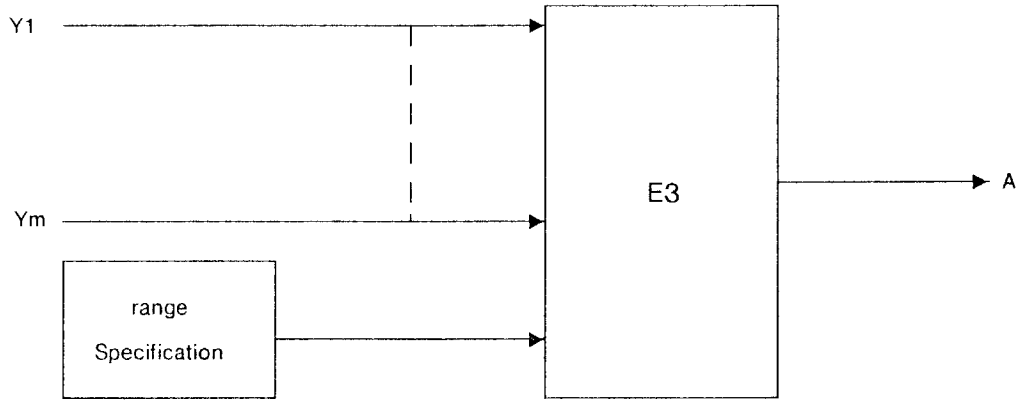


Figure 3.6.

Hence, we have the following theorem.

Theorem 3.2 A well-behaved **forall** construct can be mapped into a fully pipelined data flow graph.

Finally, we should point out that PWEs are not the only expressions which can be mapped to fully pipelined data flow graphs. Furthermore, well-behaved **forall** array operation constructs are also not the only **forall** constructs which can be mapped to fully pipelined machine codes. However, if an expression is known belong to the above category, we can deduce immediately that it can be mapped to a fully pipelined data flow graph. This fact is very important for a compiler, since many important decisions can be made earlier to improve the overall efficiency of the mapping scheme.

Another point is that our definition of PWE and well-behaved **forall** are not invented only by imagination. In fact, they are the abstractions of expressions and array construction operations frequently found in real application programs. As in GISS codes, some important parts of the computation consisted entirely of such well-behaved expressions. In the following sections we will develop the schemes to actually implement the mapping.

3.3 A mapping scheme for well-behaved forall constructs

From Theorem 3.2, we know that a well-behaved **forall** expression can be mapped into a fully pipelined data flow graph. Now the question is whether we can have a mapping scheme which effectively implements such a mapping. In this section we will discuss both important and practical issues in performing a real translation. An interesting approach for generating the desired sequence of array indices is presented. The relation of the mapping of an individual **forall** array operation expression to the structure of the overall VAL program is addressed. Since FIFO buffering is extensively used to handle the interaction between array producers and consumers, a separate section is devoted to discussing the implementation of FIFOs on a static data flow machine. There may be many mapping schemes for a well-behaved **forall** construction which achieve the stated goal. We hope that the one developed in this section can provide an efficient solution to the above problems.

3.3.1 Generation of array index values

We understand from previous sections that, in order to map a **forall** array operation construct into pipelined code, we need sequences of array index values, say ⟨1, 2...n⟩ for a one dimensional array, to drive the body of the expression (E3 in Figure 3.6). One naive way to produce the sequence is to generate an array of index values and store this array somewhere in the instruction memory or in the array memory. These indices can then be produced in fully pipelined fashion very easily. However, in many programs for numerical computation, various sequences of array indices are used, and the efficient generation of those indices becomes very important. Let us study this problem through examples.

First let us study a very simple well-behaved **forall** construct as follow :

```
B : array[real] : =
    forall   i in [1, m+1]
    construct
```

```
        if i = 0|i = m + 1 then A[i]
        else 0.5*(A[i-1] + A[i+1])
        endif
    endall
```

We can map the above code to a fully pipelined data flow graph as shown in Figure 3.7. The sequence of element values of array A, i.e. A[1], A[2]...A[m+1] is presented at the input and fed into three T operators. Three sequences of boolean control values are generated and sent to the control inputs of the T cells. The result values which come out from the T cells as follows :

T cell number	control input	result
#1	T...TFF	A[0], A[1],...,A[m-1]
#2	FFT...T	A[2],A[3],...A[m+1]
#3	TF...FT	A[0],A[m+1]

These are just the right sequences of values which the **forall** body should consume.

The final construction of the result array is done through the merge gate. The control unit also provides a sequence of boolean values to the merge gate so that the array elements are produced in the right order. To balance the graph several FIFOs are inserted as shown in the figure. The implementation of FIFOs will be discussed later.

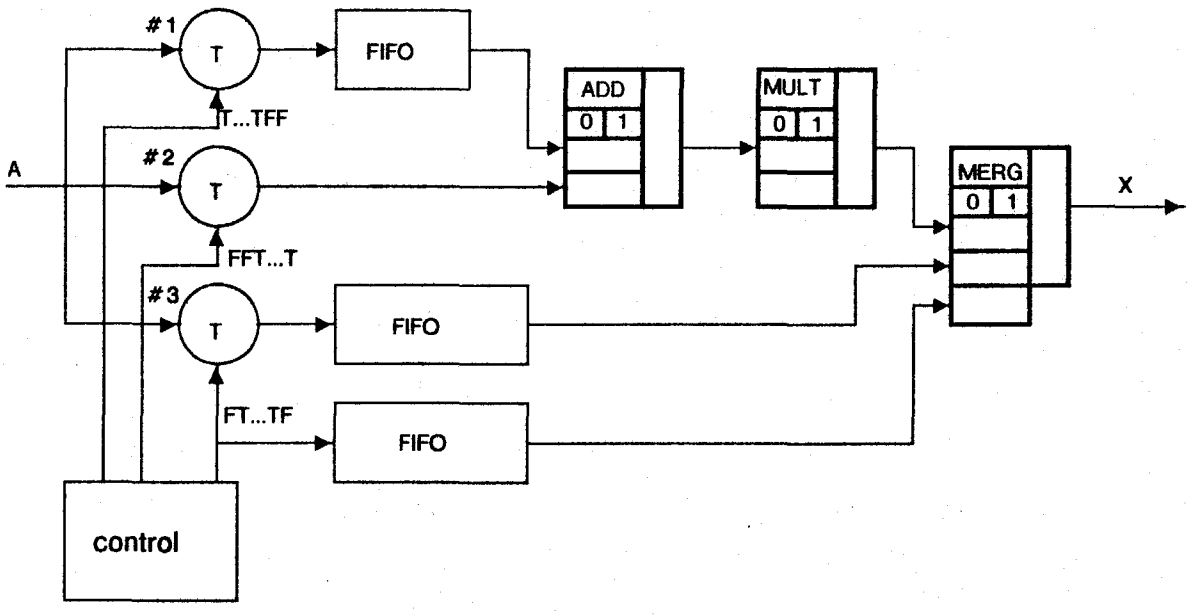


Figure 3.7.

Now we can proceed to deal with more sophisticated examples which involve multi-dimensional arrays. Let us examine an interesting example which first appeared in section 1.4. For convenience, we reproduce the program here :

```
B : array[array[real]] : =
    forall      i in [1, m+1], j in [1, n+1]
    construct
       if i = 0|i = m + 1|j = 0 |j = n + 1 then A[i,j]
       else 0.25* (A[i,j-1] + A[i,j+1]
                  + A[i-1,j] + A[i+1,j])
       endif
    endall
```

Each element of the new array B[i,j] is computed from the elements of array A in the four adjacent positions except at the boundary. Figure 3.8 shows the machine code for the program which computes row i of array B where $1 \leq i \leq m$. Note that this code is similar to the code in Figure 3.7, except that the three input arcs are driven by different streams of values, *i.e.*, row i-1, i and i+1 of A.

Figure 3.9 illustrates the generation of different rows of array A in a pipelined fashion. As we can see, the code structure of Figure 3.9 is much the same as that shown in Figure 3.7 and Figure 3.8. The only difference is the amount of FIFO buffering needed to balance the graph, *i.e.* here long FIFO buffers are required to delay rows of array elements.

It is easy to see that the above translation scheme will produce a fully pipelined data flow graph if the body of the **forall** construct is a PWE, which is true for a well-behaved **forall** expression. We can also apply the same approach to the case of multi-dimensional arrays, where much longer FIFO buffers may be required.

3.3.2 The mapping of forall and the producer-consumer interaction

In Chapter 2 we have discussed the FDG model of the global structure of a VAL program and have developed the methods of balancing the machine code for a well-behaved VAL program. For the sake of simplicity, we assumed that each node in the FDG presents a **forall** or **for-iter** construct which can be mapped to fully pipelined data flow graphs. In this chapter we present a scheme to actually implement the mapping of a well-behaved **forall** array operation construct. Now we will exploit briefly the relation of the mapping of an individual **forall** and the overall program structure.

It is very important to realize that our primary goal is to achieve the maximum throughput of the whole program. In a pipelined mapping scheme, any individual **forall** array operation construct can be viewed as a producer, a consumer, or both. The efficient mapping of the interaction between producers and consumers is crucial to the overall performance of the program. We should keep this in mind when considering any mapping scheme for an individual **forall** construct.

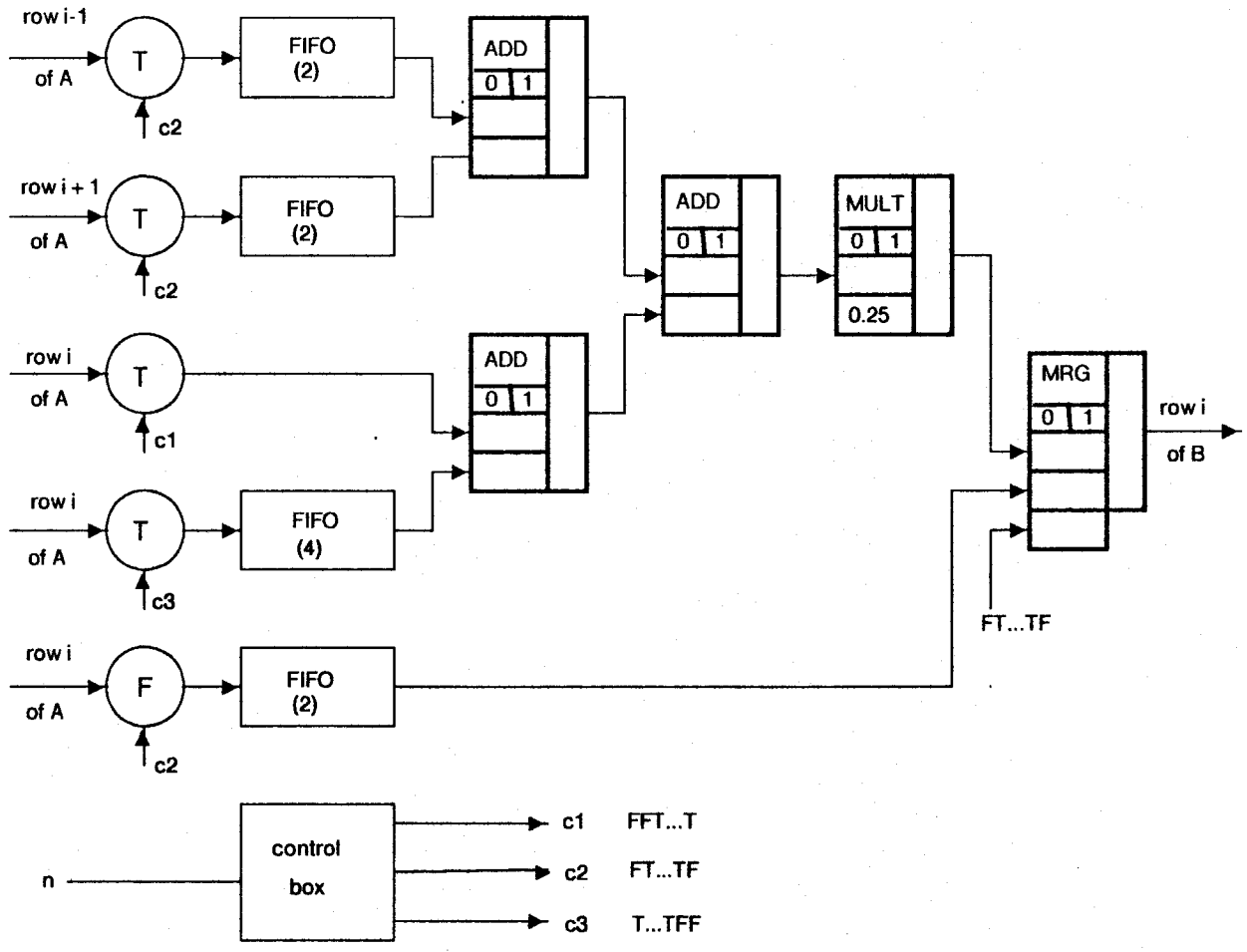


Figure 3.8.

The analysis of the FDG for a VAL program needs the information about the execution delay of its individual nodes. The structure of a **forall** node and its mapping scheme will help to decide this parameter. Based on this information, the FDG model can be processed and balanced, and the appropriate amount of buffering will be assigned to each arc which is a producer-consumer interaction link. Hence, the input and output in the data flow graphs for a **forall** construct are, in general, connected to the FIFO buffers. The implementation of these buffers should be considered as parts of the mapping of the **forall** constructs. In fact, the efficient implementation of the buffers will have significant impact on the performance of the **forall** mapping and on the throughput of the whole program. In the next section, we will discuss different ways of constructing FIFO buffers on a static data flow machine.

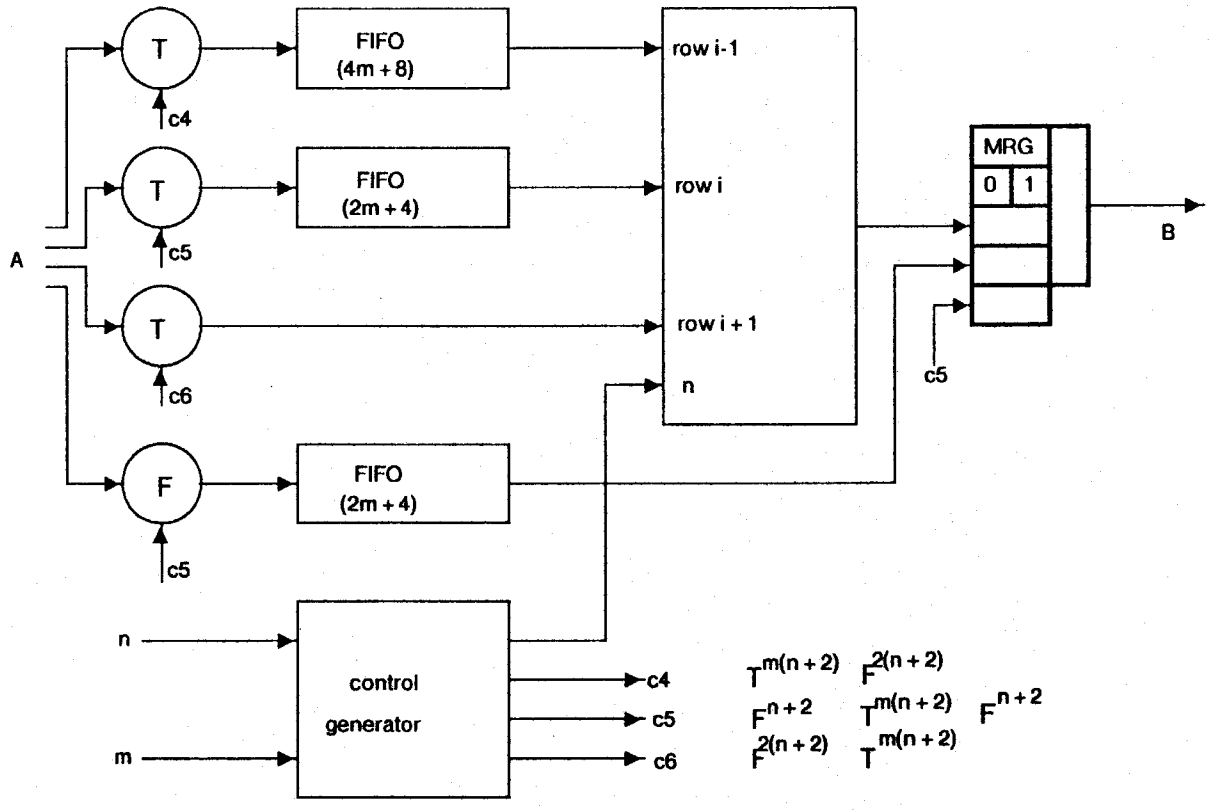


Figure 3.9.

3.4 Implementation of FIFO on a static data flow machine

As indicated in previous sections, FIFO buffers play a very important role in the pipelined mapping schemes for VAL array operations. They are extensively used as buffering both between array producer and consumer arcs of a FDG and inside the data flow graph of an individual node. The building of a FIFO on a data flow machine is very different from that on a conventional machine. In this section, we will present two different ways of constructing a FIFO buffer on a static data flow machine. Both schemes have their own advantages and can be applied to different situations.

3.4.1 FIFO buffering in processing units

In a static data flow machine architecture, one way to implement FIFO buffering is in the processing elements. A FIFO buffer can be built by cascading identity instructions as shown in Figure 3.10. A sequence of data values arrives at one end, pipelines through the identity instructions and finally comes out from the other end of the buffer. In a Dennis-Misunus machine, the delay of the FIFO is one half of the number of the identity instructions in the buffer. Each processing element has its own instruction memory which is loaded with data flow machine codes before

Figure 3.10.

execution. The identity instructions used to implement the FIFO also reside in the instruction memory. In fact, we are using instruction memory as local buffers for the pipeline. Moreover, the buffering is simulated by software (instructions), so the amount of buffering can be flexible. The only requirement is to be able to decide the length of the FIFO buffering at compile time. This does not impose further restrictions on a well-behaved VAL program which is to be mapped on a static data flow machine.

This scheme of building FIFOs has the following advantages. First, the structure of a FIFO is quite simple and is easy to implement. Second, a FIFO can be put into the same processing element as the part of the machine codes where it is being used, thus eliminating the overhead of sending instruction packets across the processing units. This is very desirable in terms of speed and efficiency, since the traffic of result and acknowledge packets in the routing networks is very busy. The major deficiency of this scheme is that the number of instruction cells used increases linearly with the length of the buffer. When the FIFO is long, the number of identity instructions used would be very big and this scheme is not a good choice. As a result, this implementation is only suitable for the case where the FIFO is comparatively short. For example, the FIFOs in Figure 3.7 can be constructed using this scheme.

3.4.2 FIFO in array memory

Another approach to implement a FIFO buffer is to use the array memory. Assume a FIFO buffer is needed between producer A and consumer B, where A and B each correspond to a two dimensional F-node in a FDG. Some free storage block (or blocks) in array memory should be allocated for the buffer of the array data values produced by A and subsequently consumed by B. In the following discussion we will use *B-block* to name the storage blocks in array memory which acts as buffers associated with some producers and consumers.

One important question is how to implement a FIFO buffer using B-blocks. There are several

possible ways to answer this question.

Assume, in the above case, the size of the array processed by A and B is *m* by *n*. In the conventional view of arrays and memory, a block of storage of size *mxn* must be allocated for the array produced by A. From our point of view, however, this is certainly not a reasonable approach. The amount of storage allocated for the buffers between A and B is decided by the entire structure of the program and the mapping scheme chosen for implementation of the pipelined machine program. It is very unlikely that the delay buffering introduced by this scheme will equal exactly the size of the array. As a result, the conventional scheme will have poor performance. It is often more flexible if we can make the B-blocks much smaller than the size of array being buffered to facilitate the memory management and make efficient use of storage space.

We propose a second way to implement FIFO buffer using B-blocks, that is we partition the whole array into small pieces and make one B-block of the same size as one piece. Then the whole buffer can be constructed from several B-blocks. For the two-dimensional example described earlier, a possible way is to partition the whole array into *m* rows. Now we can choose the size of each B-block equal to the size of one row, in this case, *n*. Another way to partition the array is to make the size of each B-block equal to the size of a column. There are also other possible ways to partition the array and choose the size of a B-block.

For the present we assume that the buffering needed may be provided by several B-blocks. In fact, if the buffering needed is not an integer multiple of the size of the B-block, we may implement the extra buffering by identity instructions in PE's, while still keeping the major part of the buffer, to be implemented in the array memory, equal to an integer multiple of the B-block size.

Figures 3.11, 3.12 and 3.13 present three schemes of constructing the buffer using B-blocks. All schemes use a FIFO queue, *i. e.* Q1, to hold the pointers to the storage for B-blocks. They are different in the way of interaction with free storage pointer pools.

In the first scheme (Figure 3.11), which will be called the *static* scheme, a second FIFO queue Q2 is used which has the same size as Q1. Initially Q2 holds several pointers to the free storage for B-blocks allocated as the buffering storage, and Q1 is empty. The number of initial pointers in Q2 is decided by the size of the buffer to be constructed. As the producer produces values for elements of a row, it gets a pointer from Q2 and stores them at increasing address in the corresponding B-block. When the last element of the row has been written, it puts the pointer into Q1 to be processed by the consumer and begins to process the values of the next row using a new pointer from Q2. The consumer will process a stream of pointer values from Q1 and read the values for successive rows of the array out of the corresponding B-blocks. The pointers will then be sent back to Q2 to be reused. After the whole process is finished, Q2 will hold the pointers, as it did before execution.

The static scheme is suitable to the case when all the run time storage usage of array memory

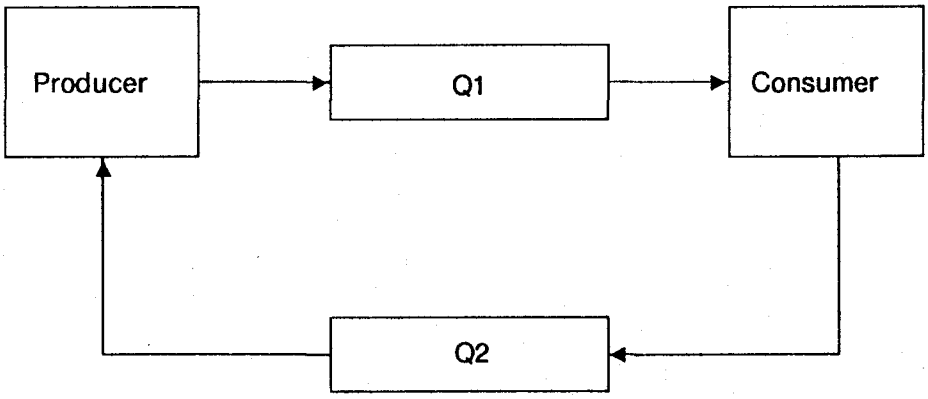


Figure 3.11.

---

can be decided before the actual execution of the programs. That is, the program has a "nice" structure, such that it is possible for a smart compiler to analyze the program and derive exact information of run time storage usage for the array memory.

The static scheme has two important features. First, all free storage allocation for the array buffers is completed at compile time. Second, the run time memory management is done automatically by the object program itself. The B-blocks allocated to implement a FIFO buffer for a producer-consumer pair can be reused again by the same pair. That is, after processing one set of inputs (arrays), Q2 is automatically initialized and ready to process the next set of inputs. No specific machine instructions or system software is needed for runtime storage managements. This not only simplifies the machine architecture but also increases the runtime execution efficiency.

The main disadvantage of this scheme is that a B-block, once allocated to a producer-consumer pair during compile time, can not be reused by other parts of the programs, even if it is no longer needed by the original pair. Because of the asynchronous nature of data flow computation, we can not assure that all code blocks are busy at the same time. As a result, the static scheme may waste a considerable amount of memory space in some cases.

The second scheme (Figure 3.12), which will be called *semi-static* is different from the static scheme in the following aspects. First, the input B-block pointers to the producer come from a control box, namely box 1 in the diagram, which allows the initial allocated pointers to come from a free storage pool through some runtime mechanism. Furthermore, the pointers, after being processed by the consumer, are sent to the input of another control box (box 2) where they will be routed to one of its two ports under control. If the whole process is not finished, the pointers will be sent to Q2 waiting to be reused by A. Otherwise the pointer will be deliver to the release mechanism where they will be returned to free storage pool.

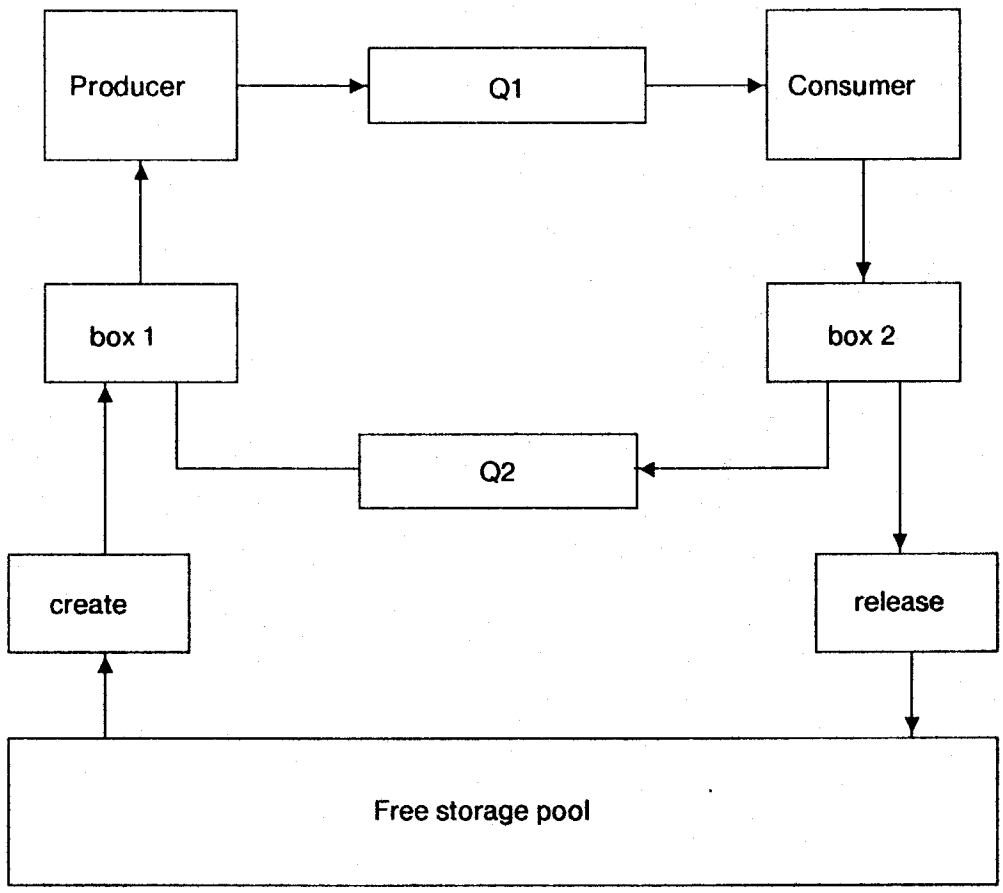


Figure 3.12.

In the semi-static scheme parts of the storage allocation are done at run time. For an array memory, the runtime memory management is simply free storage manipulation in the array memory address space without considering the interlevel transfer of information as in a machine with memory hierarchy.

There are two ways to implement the memory managements. First, we can extend our instruction set to include new instructions which correspond to the create and release operators in the scheme. The machine architecture should also be augmented by an array memory controller which can handle these new instruction packets and perform the memory management algorithms. A second, perhaps more attractive way is to implement the create and release operators as special functions. Firing of the two operators are equivalent to function calls. The system software will include these functions and handle the memory management.

The advantage of the semi-static scheme over the static scheme is that it allows simple runtime manipulation of free storage. When a FIFO buffer is needed at runtime, some B-blocks are allocated to it and may be reused in the corresponding producing/consuming process. When the FIFO buffer

is idle, the B-blocks are returned to free storage pool through a release mechanism, and may be used by other parts of the programs. Also, this scheme for runtime free storage management is very simple compared to other known schemes, even without using reference counts. Once again, this is because the memory management is closely related to the program structure and the runtime storage allocation becomes straightforward.

The third scheme (Figure 3.13), which will be called the *dynamic* scheme, is quite different. It does not have the second FIFO queue as in the previous schemes. After being processed by the consumer, the pointers for the B-blocks will be directly returned to the free storage pool.

Although the dynamic scheme supports more powerful mechanisms to manipulate the free storage, its overhead associated with both machine architecture and system software is also high. For the static data flow machine, the static and semi-static schemes are more attractive. For many VAL application programs, we can make the compiler smart enough to analyze the storage usage of FIFO buffers and the storage allocation of B-blocks can be performed efficiently at compile time. We can note that no recursion is allowed in the present version of VAL programs. For nested or multiple function calls in such programs, we can expand each function call by substituting in the code of the function body at the calling site. Remember that the assumption for the static data flow machine is that all machine level programs should be loaded into the instruction memory before execution. Under such restrictions, the static scheme and semi-static scheme have enough power to accomplish the necessary memory management. As an example, most of the array FIFO buffering in GISS code can be implemented by the static scheme with high performance. We also point out that no fragmentation problem will occur in any of the above schemes, since the B-blocks are fixed in size and would be taken from the corresponding storage pools.

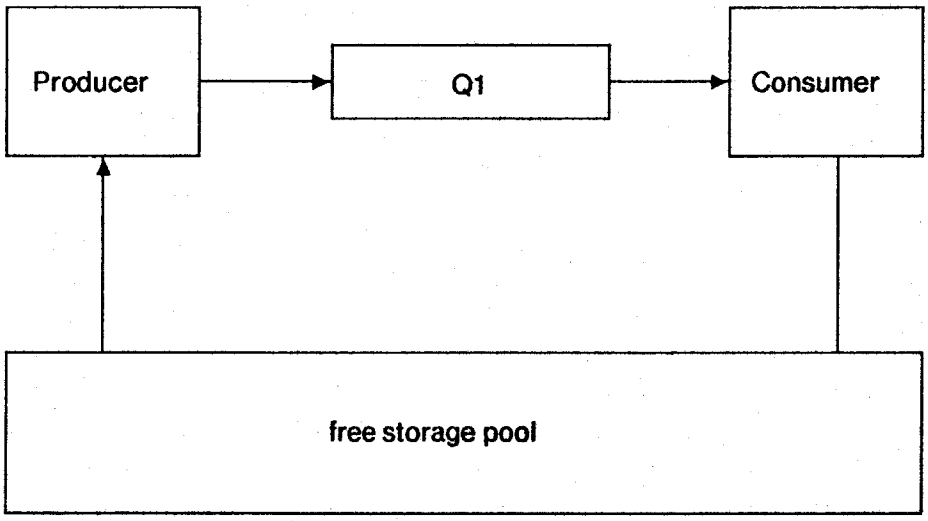


Figure 3.13.

As discussed at the beginning of this section, the inefficiency of some memory management schemes for conventional machines is due to the lack of adequate analysis of memory usage in term of program structure. Furthermore, the concepts of arrays in the conventional language also prevent necessary information from being conveyed to facilitate such analysis. On the contrary, the memory management schemes introduced above are closely related to the underlying program structure itself. Our view of array memory is nothing but temporary buffer for array values. The functionality of the VAL language and the target static data flow machine architecture make it possible for the compiler to do the necessary analysis of program structure and get all important information on runtime storage usage of buffers associated with each code block. Just as in the case of hotel management, if the room distribution is done not only based on the knowledge of the number of rooms used by a whole team of customers but also on detailed information about the behavior and schedule of each individual customer and the relationship of those customers, then clearly the room distribution plan can be made more clever. This is a good example for understanding why our schemes above are more efficient than the conventional ones.

Before leaving this section, we should point out that other schemes of implementing FIFOs are also possible. For example, the conventional ring buffer scheme can be used either in array memory or in the PE's. Our purpose in this section is only to show that FIFOs can be efficiently implemented in a static data flow machine.

3.5 The problems in mapping of for-iter constructs

In this section, we study the mapping of VAL **for-iter** array operation constructs into fully pipelined data flow graphs. We will explore the difficulties with which one is confronted when trying to develop a fully pipelined mapping scheme for them. The solution to such problems will be discussed in the following section.

The **for-iter** is another major construct in VAL which construct arrays. It can be used to express the way to construct an array by permitting sequential looping. In a **for-iter** expression, values can be transmitted from one pass through the loop to the next. This transmission is performed by defining loop parameters and then binding new values to them just prior the beginning of the next pass. VAL does not allow the use of global variables, and inside a loop objects can only be bound to the identifiers defined locally to the loop.

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

```
B : array [real] : =
   for  I : integer : = 1;
            A : array[real] : = array_empty[real]
   do  if   I ≤ N     then  iter   A[I :  if I=1 then 0
                                          else f(A[i-1])
                                          endif]
```

```
                    enditer
            endif
        endfor
```

A **for-iter** construct consists of two parts : *loop initialization* and *loop body*. In the loop initialization part, which is located between **for** and **do**, all loop parameters are declared and assigned initial values. In each execution of a **for-iter** expression, this portion is evaluated only once. Between the **do** and **endfor** is the body of the **for-iter** expression, which is repeatedly evaluated for each loop index, producing a sequence of array elements. It is the **iter** clause in the body that causes reevaluation of the whole loop body.

As in the case of **forall** expression, we are only interested in some subset of **for-iter** constructs which can be mapped into fully pipelined data flow graphs. A first restriction of such subset is given by following definition.

Definition A **for-iter** construct is *simple*, if both its initialization part and body consists of primitive well-behaved VAL expressions.

Obviously, the example described above is a simple **for-iter** array operation construct.

Several researchers have studied the translation of **for-iter** array operation constructs into pipelined data flow graphs. A scheme of mapping **for-iter** constructs is proposed by Montz [14] and elaborated by Todd [16]. If the **for-iter** expression under consideration produces an array, then the above scheme can generate the array values in a pipelined fashion. Figure 3.14 illustrates the data flow graphs for a **for-iter** array construction expression translated using Montz's scheme.

A feature of a data flow graph corresponding to a **for-iter** construct is that it contains cycles. Due to the existence of cycles, the graph produced by Montz's scheme, in general, is not fully pipelined. Let us consider the behavior of the code in Figure 3.14 more carefully. Assume the body of function f is mapped into fully pipelined data flow graphs, and the execution time of f is *df* stages. The feed-back link between the input and the output of f prevents the whole graph from being fully pipelined. This is because the value of A[I] depends on the value of A[I-1]. As a result, the calculation of A[I] can not start until the computation of A[I-1] is finished and the value of A[I-1] is available at the input of f. Since the time of calculation of f is df, the initialization rate of the pipeline can not be higher than 1/df, which is not optimum when $df > 1$. If $df \gg 1$, the throughput of the pipeline will be seriously degraded. Furthermore, the throughput of the program depends on the length of the "pipe", which is very undesirable.

Stoy [15] suggested another approach for mapping **for-iter** constructs which provides a way of handling different activations of the same **for-iter** construct concurrently. But it does not address the problem of full pipelining.

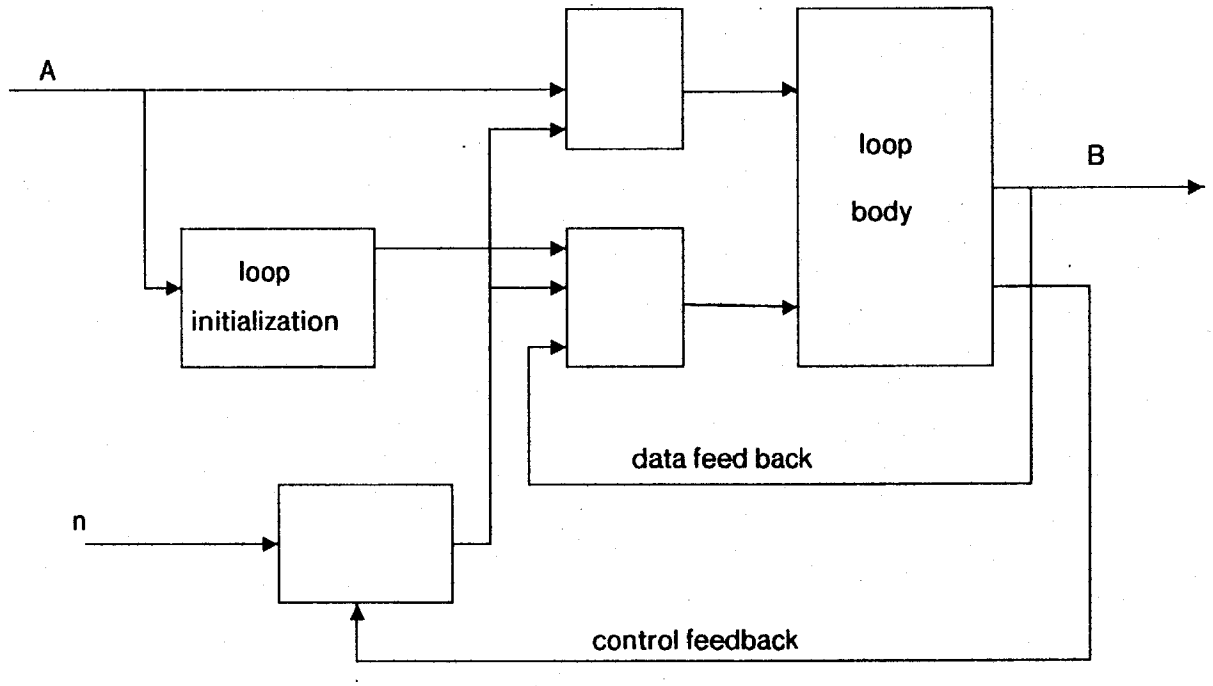


Figure 3.14.

The difficulty we meet here is that the balancing algorithm developed in this thesis so far and in [14] does not apply to the case where the graph is not acyclic. To further classify the problems which can be handled and to develop a scheme for them, we need to study them in a more formal context. This is the topic of next section.

3.6 A New Scheme for Implementing For-iter Array Operations

The most common problems involving **for-iter** array operations are recurrences, which accepts as input a sequence of values $a_1, a_2 \cdots a_n$, and produce as output another sequence of value $x_1, x_2 \cdots x_n$, where for some m>0, each x_i is a function of a_i and x_{i-1} through x_{i-m}. That is :

$$x_i = f(a_i, x_{i-1} \cdots x_{i-m}) \qquad (3.1)$$

Mathematically, we have the following definition.

Definition : The function f in (3.1) is called a *recurrence function*, and a_i is called *parameter vector.* Also we call the recurrence expressed by (3.1) m^{th} *order recurrence.* If $m=1$, it is named *first order recurrence.*

An example of a first order recurrence relation is shown below :

$$\begin{aligned} x_i &= A_i x_{i-1} + B_i \\ &= a_i(1) x_{i-1} + a_i(2) \\ &= f(a_i, x_{i-1}) \end{aligned} \qquad (3.2)$$

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. In general, different recurrence relation may have different parameter vector lengths. However, for any particular problem a parameter vector $\mathbf{a}_i$ must have constant length for all *i*. 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 this problem would ideally accept each new input $\mathbf{a}_i$ and feed it into the pipeline as soon as it is available. Unfortunately, a direct translation scheme, in general, can not achieve this result.

Many authors have studied this problem, yet no general scheme is known that works for an arbitrary recurrence relation. However, for a certain class of recurrence relations, a good solution has been proposed [10][11]. Most previous work in this field is concentrated on the implementation of the proposed solution on a conventional machine architecture. However, our emphasis is in applying this approach to the mapping of **for-iter** array operation constructs on a static data flow machine. We will show the advantages of applying it to a data flow machine as compared to a conventional machine.

In the proposed approach, the basic recurrence equation for computing x_i from $x_{i-1}...x_{i-m}$ is *backed up* on itself by substituting appropriate copies of the original equation (3.1) for one or more of the $x_{i-1}...x_{i-m}$ terms :

$$\begin{aligned} x_i &= f(a_i, x_{i-1}...x_{i-m}) \\ &= f(a_i, f(a_{i-1}, x_{i-2}...x_{i-m-1}), x_{i-2}...x_{i-m}) \\ &= f(a_i, f(a_{i-1}, x_{i-2}...x_{i-m-1}), \\ &\quad f(a_{i-2}, x_{i-3}...x_{i-m-2})...f(a_{i-m}, x_{i-m-1}...x_{i-2m})) \end{aligned}$$

This procedure removes the dependence of x_i on x_{i-1} completely, thus easing the pipelining problem. Further backup can remove the dependence on x_{i-2}, x_{i-3}.... In general, use of backup will cause the number of instruction cells to increase substantially. For example, even a single backup for elimination of x_{i-1} will double the number of instruction cells and this will also double the length of the pipeline.

Fortunately, for many common problems found in real applications, the recurrence function f can be transformed to avoid the above problem. 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 :

$$f(\mathbf{a},f(\mathbf{b},x)) = f(g(\mathbf{a},\mathbf{b}),x) \qquad (3.3)$$

then the function g is termed a *companion function* for f.

In fact, any first order recurrence with a companion function can be backed up as follow :

$$\begin{aligned} x_i &= f(a_i, f(a_{i-1}, x_{i-2})) \\ &= f(g(a_i, a_{i-1}), x_{i-2}) \\ &= f(g(g(a_i, a_{i-1}), a_{i-2}), x_{i-3}) \\ &= f(g(g(\ldots g(a_i, a_{i-1})\ldots a_{i-df+1})), (x\text{-}df)) \\ &= f(c_i, x_{i-df}) \end{aligned} \qquad (3.4)$$

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

To expand the pipeline computing f into a complete system requires adding an additional pipeline that computes c_i from a_i. The output of this new pipeline will then feed into the f block directly. This pipelined will be named the *companion pipeline* in the rest of this thesis. Figure 3.15 shows this implementation scheme. If we construct the new pipeline properly, it is possible to keep the whole pipeline running at maximum throughput.

Now let us consider the example program in (3.2) which is a first order recurrence relation, and show how to apply the above scheme to it. Now the function f has a execution delay of 2 (df=2). We can transform it as follows :

$$\begin{aligned} x_i &= A_i x_{i-1} + B_i \\ &= a_i(1) x_{i-1} + a_i(2) \end{aligned}$$

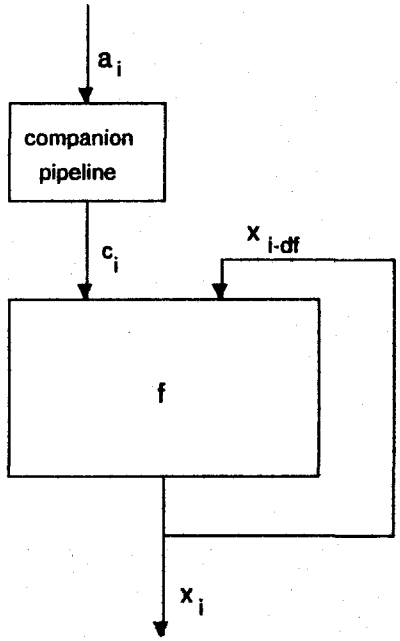


Figure 3.15.

$$= a_i(1)[a_{i-1}(1)x_{i-2} + a_{i-1}(2)] + a_i(2)$$
$$= a_i(1)a_{i-1}(1)x_{i-2} + (a_i(1)a_{i-1}(2) + a_i(2))$$

We can see that the function f of (3.2) has companion function g, where

$$g(a, b) = (a(1)b(1), a(1)b(2) + a(2))$$

As a result, f can be expressed as :

$$x_i = f(a_i, x_{i-1})$$
$$= f(a_i, f(a_{i-1}, x_{i-2}))$$
$$= f(g(a_i, a_{i-1}), x_{i-1})$$

Figure 3.16 shows the fully pipelined data flow graph for f.

Figure 3.17 illustrates another example, which corresponds to (3.4) with df = 4. Now we have :

$$x_i = f(c_i, x_i\text{-}4) \qquad (3.5)$$

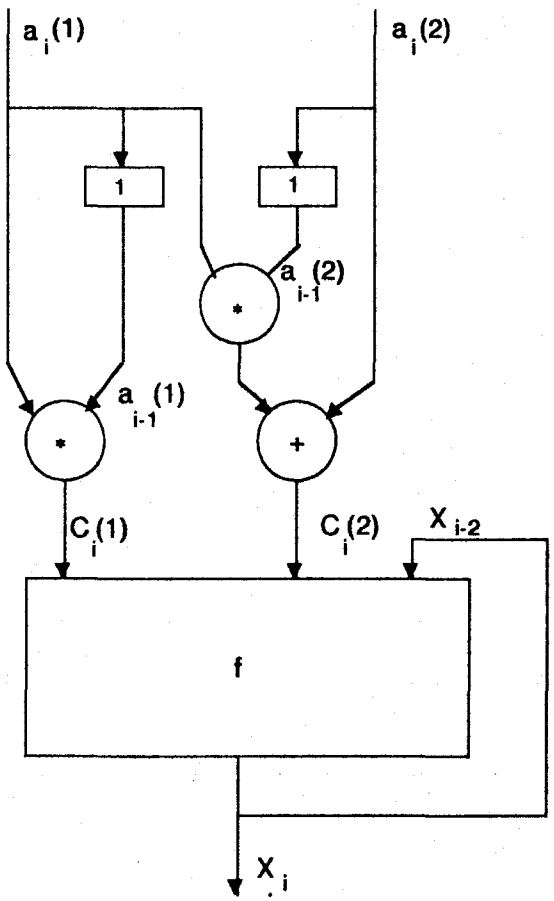


Figure 3.16.

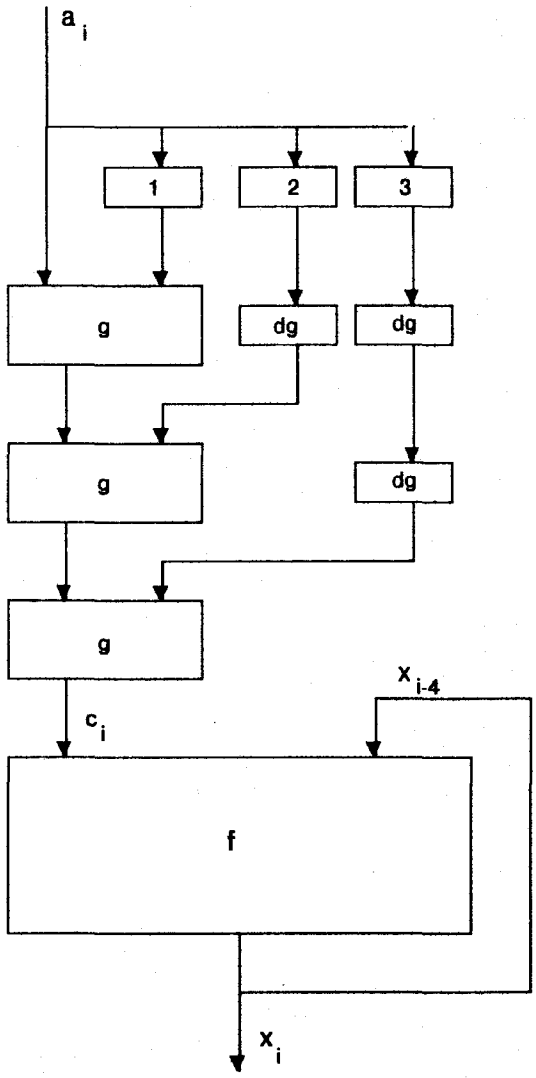


Figure 3.17.

---

where

$$c_i = g(g(g(a_i, a_{i-1}), a_{i-2}), a_{i-3})$$

Note also that the execution delay of g is *dg*.

If the function g is associative, we can have a tree-like arrangement as in Figure 3.18. Here (3.4) is rewritten as :

$$c_i = g(g(a_i, a_{i-1}), g(a_{i-2}, a_{i-3}))$$

This is similar to the so-called *log-sum* reduction used for many years in parallel computers [12].

Fortunately, it can be proved that the companion function g is indeed associative. Furthermore, if a recurrence function f has a companion function g as defined in 3.3 then any x_i can

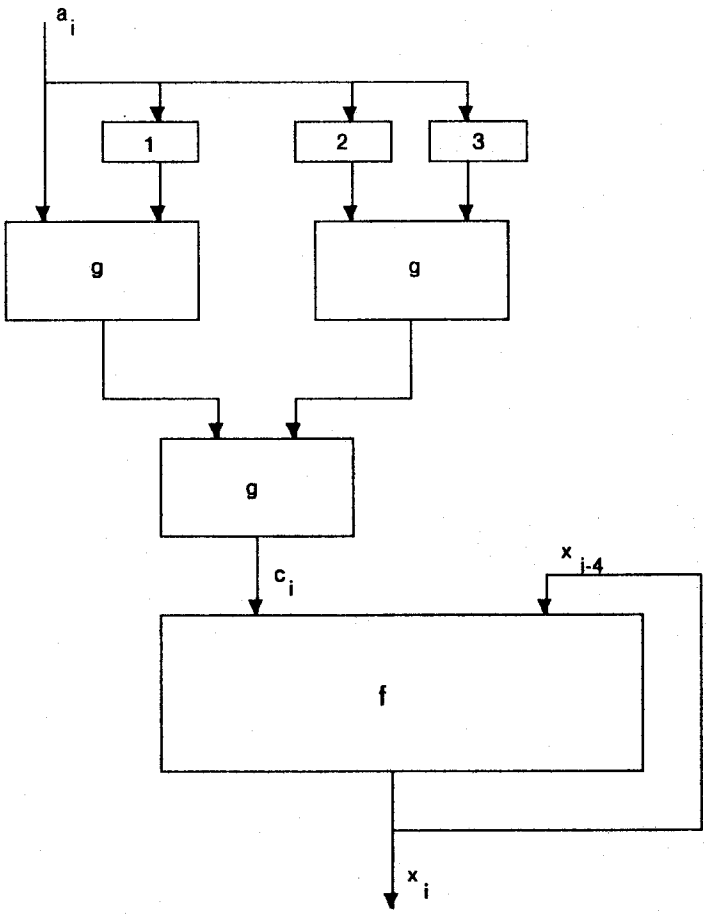


Figure 3.18.

---

be expressed in terms of x_j, where $0 \leq j \leq i$ and

$$x_i = f(a(i, i\text{-}j), x_j).$$

Note that a(i, i-j) can be expressed by g as in (3.4). The above result can be proved easily from the definition of the companion function.

Therefore, if the number of stages in f is *df*, we can construct a pipeline consisting of $\log_2 df$ levels of g arranged as in Figure 3.19, which is a fully pipelined program.

Note that companion function concept is not limited simply to first order recurrences. For m^{th} order recurrence, there exists a similar result [13], which will not give in detail here.

In [11] we can find a list of recurrence problems that are known to have companion functions. In our experience, quite a few recurrence problems found in real applications fall in this class. In particular, all linear recurrence relations have their companion functions. A short proof can be found along the same line as our analysis for (3.2). A VAL compiler should be able to detect those

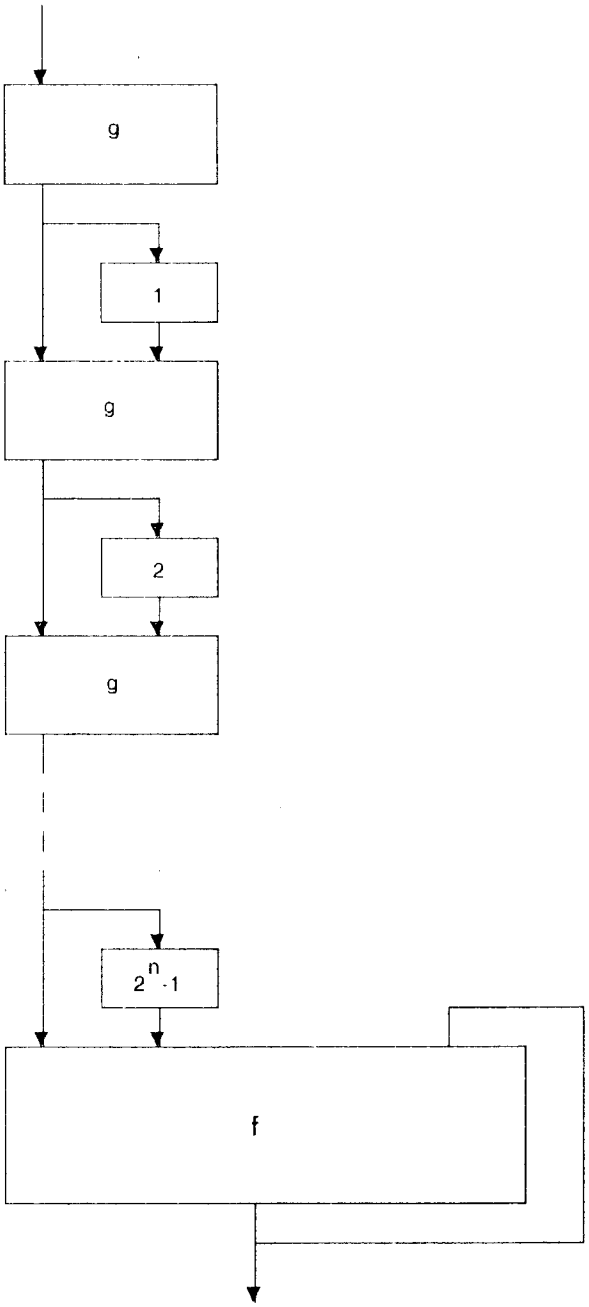


Figure 3.19.

recurrences and direct the graph generator to construct appropriate data flow graphs.

As we can see from the above analysis, one important factor in applying the new scheme is the introduction of the companion pipeline. Since many different recurrence relations may be used in various computations, the exact form of the companion pipelines needed for a particular problem is difficult, if not impossible, to predict. 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, usually as the pipelined arithmetic and logic unit, with some additional hardware support. The introduction of a companion function is not flexible in such a context. On the contrary, the pipeline in the architecture of a data flow machine is software implemented. As a result, it is very easy 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 relatively easy 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 df is big. The complexity of a compiler to analyze the code is also a factor which should be considered. Nonetheless, the author believes, the new scheme is a good start for further study of the mapping of the VAL **for-iter** array operation construct into fully pipelined data flow graphs.

Finally, let us return to the problem of classifying VAL **for-iter** constructs which have good mapping schemes.

Definition A **for-iter** expression is *well-behaved*, provided (1) it is simple, (2) the recurrence function it denotes has a companion function, (3) the VAL expression which computes the companion function is primitive well-behaved.

Consequently, from the results presented in this section, we have the following theorem :

Theorem 3.3 A well-behaved **for-iter** array operation construct can be mapped into a fully pipelined data flow graph.

Now we are ready to review the notion of well-behaved VAL programs. Using the concepts introduced in this chapter and the related results, we can define the notion of well-behaved VAL program more precisely.

Definition A *well-behaved* VAL program is a VAL program which has the following properties :

(1) All arrays in the program are *static*, *i.e.* array bounds are compile-time computable constants.

(2) All array construction operations in the program are expressible by **forall** and **for-iter**

constructs which are well-behaved.

From theorem 3.2 and 3.3, we know that a well-behaved VAL program can be mapped on a static data flow machine in a fully pipelined fashion. As a result, we have the following theorem :

Theorem 3.4 A well-behaved VAL program can be mapped into fully pipelined data flow graphs.

4. Conclusions

4.1 Summary of the thesis

The aim of this thesis has been to study the problem of mapping array operations in the high level language VAL on a static data flow machine. The main interest is producing pipelined data flow machine codes which can run with maximum throughput. Since data driven instruction execution is radically different from its conventional counterpart, new models and schemes for its translation should be investigated and developed.

The main contributions and accomplishments of this thesis are :

(1) The specification of a class of VAL program, i.e. well-behaved VAL programs, for which both a good model and efficient schemes exist to map the array operations into fully pipelined machine codes on a static data flow architecture.

(2) The development of flow dependency graph model (FDG model) for the class of well-behaved VAL programs. A polynomial time algorithm is presented for balancing a general acyclic FDG to achieve maximum pipelining. The issue of reducing the buffering in a balanced FDG is addressed. An algorithm for reducing the buffering based on equivalent transformation of balanced FDG is formulated. Although no general algorithm is known for optimization of a balanced FDG, a class of FDG (well-structured FDG) is defined for which a polynomial time algorithm exists for its optimization.

(3) The establishment of mapping schemes for well-behaved **forall** and **for-iter** array operation constructs. In particular, a new scheme for mapping **for-iter** constructs to achieve maximum pipelining is proposed. This is based on the concepts of companion function for a recurrence function. The construction of FIFO buffer on a static data flow machine is also discussed which is crucial for efficient implementation of above mapping schemes.

4.2 Suggestions for furthur research

For further research, I suggest following areas.

The first is the extension of the FDG model presented in chapter two. This may include the generalization of the balancing algorithm to multi-input and multi-output FDG and the extension of the model to inter-function data flow analysis. Another interesting topic would be the existence of polynomial time optimization algorithms for a general FDG model. My present conjecture is that this belongs to the NP-complete problems.

The second area would be the further improvement of companion function schemes for mapping **for-iter** array operations. The useful class of recurrence functions found in most application programs which have companion functions should be investigated and classified. A practical algorithm for a VAL compiler to recognize such classes of functions and to produce the efficient

codes should be devised.

The third interesting area would be the design of practical usable algorithms for a VAL compiler to implement the analysis of a well-behaved VAL program based on the FDG model. The abstract algorithms developed in chapter two should be revised or improved to be implemented in a real VAL compiler.

Finally, it would be a very challenging task to implement the array mapping schemes developed in this thesis on a static data flow machine. Only after we have successfully implemented the mapping of a practical application program on a real data flow machine, can we be in a good position to assess the theoretical model, to evaluate the scheme and the technique, to suggest further extensions and improvements for them.

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] Allen, F. E. "Program Optimization" *Annual Review of Automatic Programming*, 5, 239-307, 1969.

[3] Cocke, J. and Schwartz, J. T. "Programming Languages and Their Compilers." Courant Institute of Mathematical Science, New York, N.Y., 1970.

[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., 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.

[7] Dennis, J. B., C. K. Leung, and D. P. Misunas. "A Highly Parallel Processor Using a Data Flow Machine Language." Computation Structures Group Memo 134-2, Laboratory for Computer Science, MIT, Cambridge, MA, January 1977 (revised June 1980).

[8] 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.

[9] Harary, F. "Graph Theory", Addison-Wesley, Reading, Mass. 1969.

[10] 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.

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

[12] Kuck, D. E. "ILLIAC IV Software and Application Programming" IEEE Trans. Comput., Vol. c-17, no. 8, Aug. 1968.

[13] Matthew, S. H. "Flow Analysis of Computer Programs" Elsevier. North-Holland Inc., 1977.

[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] Stoy, J. E. "Functions in the Form 1 Data Flow Machine." Unpublished communication.

[16] Todd, K. W. "An Interpreter for Instruction Cells." Unpublished communication, January 1981.

[17] Weng, K. S. "An Abstract Implementation For a Generalized Data Flow Language" Technical Report 228, Laboratory for Computer Science, MIT, Cambridge, MA, February 1980.