A Pipelined Code Mapping Scheme for Static Data Flow Computers

by

Gao Guang Rong

S. B., Tsinghua University, Peking (1968)

S. M., Massachusetts Institute of Technology (1982)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

August 28, 1986

Signature of Author _____
Department of Electrical Engineering and Computer Science
Aug 28, 1986

Certified by _____
Jack B. Dennis
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Departmental Graduate Committee

A Pipelined Code Mapping Scheme for Static Data Flow Computers

by

Gao Guang Rong

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 1986 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

Computers built on data flow principles promise efficient parallel computation limited in speed only by data dependencies in the calculation being performed. We demonstrate how the massive parallelism of array operations in numerical scientific computation programs can be effectively exploited by the fine-grain parallelism of static data flow architecture. The power of such fine-grain parallelism derives from machine-level programs that form large pipelines in which thousands of actors in hundreds of stages are executed concurrently. Each actor in the pipe is activated in a totally data-driven manner, and no explicit sequential control is needed.

This thesis studies the principles of program mapping techniques that can be made to achieve high performance for numerical programs when executed on a computer based on data flow principles. A simple value-oriented language is specified as the source language to express user programs. The key of the program mapping techniques is to organize the data flow machine program graph such that array operations can be effectively pipelined. Program transformation can be performed on the basis of both the global and local data flow analysis to generate efficient pipelined data flow machine code. A pipelined code

mapping scheme for transforming array operations in high-level language programs into pipelined data flow graphs is developed. The optimal balancing of data flow graphs is investigated and an efficient solution is formulated. Based on the pipelined code mapping scheme, the pragmatic issues of compiler construction and efficient architecture support for pipelining of machine level data flow programs are addressed.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering

Keywords: applicative programming, parallel computation, pipelining

Acknowledgments

I would like to thank my thesis supervisor, Jack Dennis, for his guidance and support during the course of this research. He has provided not only an intellectually exciting environment for research, but also the guidance which opened my interests in many aspects of modern computer science.

My other thesis readers, Arvind and John Guttag, have been very helpful in this work, and I have enjoyed interesting discussions with them on many topics in the field of computer science.

The friendly and stimulating atmosphere in the Computation Structures Group have been a constant source of support. I would particularly like to thank Clement Leung, Bill Ackerman, Andy Boughton, Dean Brock, Keshav Pingali, Tam-Anh Chu, Willie Lim, and David Culler for their contributions to this work and for their friendship.

Many friends have provided support and help when they are needed. In particular, I would like to thank Robyn Fizz for many suggestions to improve the English of the thesis.

Finally, I am very grateful to my wife, Ping Gao and my son Ning Gao for their patience, companionship and love during the difficult years. In particular, Ping Gao has typed the manuscript of the entire thesis for many long night hours.

CONTENTS

1. Introduction

The past decade has seen sustained efforts in the design and implementation of high-performance supercomputers. They are the highest performance machines available for automatic computation. The most successful supercomputers of the present day are conventional von Neumann stored program computers based on sequential instruction execution. However, there exists a mismatch between the amount of parallelism available in many large scientific computations and the amount of concurrency which can be efficiently supported by computers based on von Neumann architecture. The sequential nature of the von Neumann architecture, in which a sequential control mechanism is used to schedule instruction execution, creates the so-called *von Neumann bottleneck* [13].

To overcome this bottleneck, a variety of innovative architecture concepts has been developed to improve the performance of von Neumann computers. The instruction overlap architecture allows concurrent execution of several instructions; the cache speeds up the memory accesses; pipelined processors exploit the parallelism expressed in vector operations, and array processors make it possible to organize multiprocessors working in parallel in a lock-step fashion.

In spite of these advances, the demands for ever increasing computing power have not yet been satisfied. If future generation supercomputers are to achieve significant gain in performance over current designs, they will have to process large number (hundreds or thousands) of basic operations concurrently. The arrival of VLSI technology causes a dramatic change in cost performance trade-offs for solving large problems on a computer. High levels of concurrency will be achieved by machines consisting of many instruction processing units, function units and memory units, which become practical only by the advent of VLSI technology. To effectively organize and realize these high levels of concurrency presents a major challenge to computers built on von Neumann architecture [11]. Attempts to eliminate the von Neumann bottleneck have lead to the introduction of novel forms of computer architecture such as data flow computers.

Despite the widely recognized feature of proposed data flow model of computation — its ability to exploit parallelism at the level of atomic operations, skepticism exists concerning their efficiency in high performance scientific computation [51]. A long-standing issue has been the efficient mapping of massive parallelism in array operations. A major goal of this thesis is to show that the power of *fine-grain* parallelism supported by data flow architecture can be effectively utilized in handling arrays. In structuring data flow machine code, we have found pipelining of array operations a very attractive way to exploit such massive parallelism.

Most compilers for conventional vector processors do some form of analysis to identify parallelism in serial code written in conventional programming languages such as Fortran. After the parallelism is detected, the compiler will generate machine code for the vector operations using inherently sequential scheduling mechanism available in the von Neumann processor architecture. Therefore, in order to perform a successful program mapping, a vectorizing compiler must overcome two forms of the von Neumann bottleneck: in the source language and in the machine architecture. While considerable progresses has been made in parallelism detection of user programs [69], the bottleneck originating from the instruction scheduling of machine architecture makes the vector code generation phase extremely machine dependent. It may require sophisticated analysis to overcome the barrier caused by problems like conditional statements and vector register allocation. As a result, such efforts are often performed under a strategy emphasizing local optimization rather than global optimization. Therefore, even when massive parallelism in user programs has been successfully detected, we still need an effective code mapping strategy to organize the computation such that the parallelism can be best handled and exploited by a highly parallel computer architecture. The von Neumann bottleneck has been the major obstacle for conventional vector compilers, and so far there are no easy solutions to the problem.

The program mapping scheme developed in this thesis transforms the user program

such that the massive parallelism in array computation can be effectively matched with the power of fine-grain parallelism supported by data flow architecture. The goal of such optimization is to achieve effective pipelining of the data flow machine code. Such a pipelined program restructure and transformation strategy requires both local and global analysis and optimization. This adds a new and interesting dimension of optimization problems faced by a compiler.

In developing the code mapping scheme, we assume functional languages are chosen as high level programming languages for writing user programs because they encourage an applicative style of programming in expressing parallel computation. In particular we choose to use a functional language with features which are useful in expressing array operations with certain regularities frequently found in highly parallel numerical computation.

This chapter gives an introduction to the various aspects of the thesis research and outlines its scope.

1.1 Massive Parallelism of Array Operaions in Numerical Computation

A major driving force in the development of high-performance computers has been scientific computation. In applications such as weather forecasting, aerodynamic simulation, nuclear physics, seismology and signal processing, enormous computing power is required to handle massive parallelism in an efficient manner. Problems in scientific computation are usually expressed in linear algebra with all data structured as elements of arrays. The kernels of such array computation typically demonstrate certain regularities. In the computation, the bulk of the elements of an array are processed in a regular and repetitive pattern through the different phases of program execution. For example, references to array elements are usually organized in an iteration loop in which the array elements are accessed by a simple indexing scheme, as illustrated in the following Fortran program.

```
      DO 10 J = 1, N
      X(1,J) = A(1,J)
10    X(N,J) = A(N,J)

      DO 20 I = 2, N-1
      X(I,1) = A(I,1)
20    X(I,N) = A(I,N)

      DO 30 I = 2, N-1
      DO 30 J = 2, M-1
30    X(I,J) = (A(I,j-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))/4
```

The index computation for all array references of X is done in the form of i+b or j+b where b is a compile-time constant. Furthermore, all elements of the array X are defined exactly once in this loop.

The program in the above example consists of considerable number of array operations, a fact typical for a scientific numerical computation program. For example, many applications take the form of computing successive states of a physical system represented by physical quantities on an Euclidean grid in two or three dimensions, and the new values of each grid point may be computed independently. Thus, the degree of concurrency is often at least equal the number of the grid points (for a 100×100×100 case, the parallelism will be well over 10^6 !). Therefore, the efficient mapping of the massive parallelism of array computation into machine-level code structure has been a major consideration in the design of high-performance computer architecture as well as program transforming compilers.

1.2 Vector Processing and Vectorizing Compilers

Achieving massive speed-up of array operations has been a challenging task for designers of von Neumann parallel computers. The array processors, with ILLIAC IV as a pioneer [14,18], depend upon simultaneous lock-step operations on many elements of an array. These processors perform well only when data structures can be mapped onto a fixed machine structure imposed by the physical arrangement of the processors, e.g., linear arrays, two-dimensional arrays, etc.

Vector and pipelined processors [22,86] perform repetitive operations on elements of an array sequentially with substantial overlap through the hardware pipelined functional units. The architecture of such machines usually consists of pipelined function units, interleaved memory modules and fast vector registers. For such processors to be efficiently utilized, the machine programs must be organized such that the sequence of elements of the operand arrays needed to complete a vector operation are continuously accessed and processed by special pipelined function units or by many tightly coupled function units.

The architecture of the various vector processors usually supports a set of vector operations (instructions). For example, *vector add* is a typical vector operation described as:

$$\text{VADD: } C(I) = A(I) + B(I)$$

where A and B are vector operand, and C is the result vector, and I = 1 through n — the *length* of the vector. A vector operation performs the same scalar operation on successive elements of the vector. In most commercial vector processors, identical hardware pipelines must execute the same vector operation at the same time.

User programs for vector processors are written in conventional programming languages (e.g., Fortran, Pascal) which are sequential in nature. An intelligent compiler must be developed to detect and regenerate parallelism lost in the use of sequential

languages. The process to replace a block of sequential code by vector instructions is the so-called *vectorization*. For example, the following Fortran loop can be vectorized into the vector VADD instruction discussed above.

```
      DO 10 I = 1, N
10    C(I) = A(I) + B(I)
```

For vector processors the fundamental problem, given that the parallelism in the program is being successfully detected, is to schedule the machine code to overcome the von Neumann bottleneck. To achieve this goal, the compiler must vectorize complicated data accesses and restructure program sequences, subject to instruction precedence and machine hardware resource constraints. On one hand, an array is a block of storage cells physically allocated in memory. Transmission of an array from one part of the program to another occurs directly through physical allocation and moving blocks of data in the memory. On the other hand, the object programs are coded in von Neumann machine instruction set which depends on the sequential control mechanism and lacks clarity in terms of resource constraints when sharing is concerned. The inflexibility severely limits its ability to schedule different computations on different elements of an array. For example, a barrier to vectorization exists in the handling of conditional and branch statements, sequential dependencies, etc. It remains to be seen whether an overall program mapping scheme can match the detected parallelism in source programs with a sequentially controlled processor architecture.

Moreover, when there is substantial parallelism of operations on multiple arrays in different parts of the program, the problem of scheduling and synchronizing these operations on multiple pipelined vector processors becomes more difficult. In fact, it has been indicated that multi-pipeline scheduling is computationally hard, even for a restricted class of problems [55]. A feasible solution must include a suitable scheme for programming the communication of tremendous amounts of data traffic between processors and

memories when many pipelined instruction/data streams are processed. Until recently, the most successful vector machines were uni-processors such as the Cray-1 or Cyber-205. The current direction of vector processing is to allow a small number of vector processors (2,4 or 8) to form a parallel computer architecture, as in the Cray-X-MP-2 and Fujitsu VP-200 [80].

1.3 Data Flow Computers

The data flow model of computation which has been proposed as an approach for high-performance parallel computers represents a radical departure from von Neumann architecture [10,12,25,26,60]. In a data flow model, the computation is modeled by a data flow graph — a directed graph with nodes that represent actors and arcs that transmit tokens which carry values to be processed by these actors. Actors are activated for execution by the presence of tokens carrying operand values at their input arcs. In this computation model, the execution of a program is intrinsically data-driven, i.e., the order of execution between operators is determined only by data dependencies.

In recent years research has been conducted on data flow architecture that can directly execute programs encoded as data flow graphs [12,34,35]. A machine-level data flow program, regarded as a collection of instruction cells, is essentially a directed graph, with each node corresponding to an instruction and each arc specifying the destination of the result value of an instruction execution. Unlike von Neumann computers, data flow computers have no program counter or other form of centralized sequential control mechanism. The parallelism which can be exploited by an ideal data flow computer is limited only by the data dependencies in the data flow programs.

Two major approaches to the architecture of data flow computers are currently being pursued. They are the static data flow model [26] and the tagged token model [11,92]. In this thesis, we deal only with the static data flow model. In the static data flow model, an arc can hold no more than one token. The machine programs for a static data flow computer are loaded into the processor memory before execution and only one instance of

an instruction is active at a time. Once an actor has been executed, it cannot be executed again until the tokens carrying previous result values have been consumed by all successor actors.

The structure of a static data flow supercomputer proposed by the Computation Structures Group of MIT [35] is shown in Figure 1.1. The data flow programs are held in the memory local to each processing element (PE). When an instruction is enabled for execution by the arrival of its operand values, an operation packet is formed and sent to the function unit FU or array memory AM, depending on the type of operation it requires. The result value of the operation is sent to its successor instructions in the program graph. The organization of the processing units that handle enabled instructions and initialize

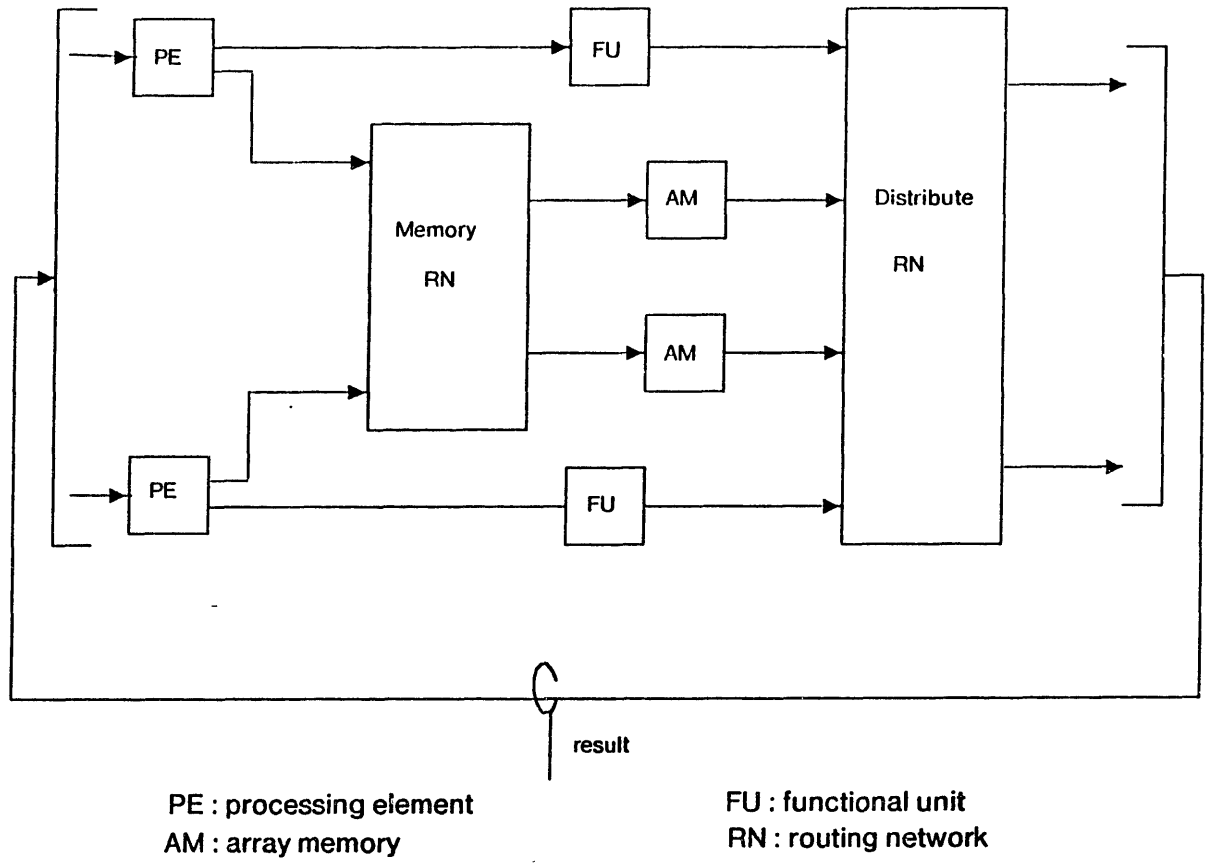


Figure 1.1. A static data flow architecture

their execution has been described in [27,28,92]. The role, analysis and structure of routing networks are described in [28,29,16,17]. Performance evaluation of the static data flow architecture for a few benchmark programs has been studied in [31,35,84].

1.4 Granularity and Functionality Issues

Despite its attractive features, data flow model of computation has raised considerable controversy among researchers. Not surprisingly, the criticisms of data flow architecture for high-performance numerical scientific computation have also been centered on array computation. The main skepticisms about the ability of data flow computers in handling array operations are due to their emphasis on fine-grain, operational-level concurrency [51,52].

Parallelism can be exploited at different levels of computation: task level, procedure/function level, or instruction level. Granularity issues have been an important consideration in parallel computer architecture design. The operational model of data flow graphs can exploit the parallelism explicitly at each atomic machine operation level, and the corresponding architecture is said to be based on the *fine-grain* data flow principle. For example, both static and tagged token data flow machines use the fine-grain data flow principle.[1]

The success of data flow architecture in scientific computation depends on the program mapping or transformation schemes which can organize the computation such that the massive parallelism and regularity in array operations can match the fine-grain parallelism efficiently supported by the architecture.

According to the functionality of data flow graphs, any actors (operations) must be side-effect free. Conceptually, this precludes the sharing of data structure such as arrays.

---

1. In contrast, some researchers advocated a coarse-grain data flow principle, where an atomic actor in the graph is a procedure [51].

An array append operation A[i:v] expressed in data flow languages such as Val [4] would mean the construction of a new array A' which is a fresh copy of A except that the i-th element is replaced by v [4]. One direct implementation is to copy the array A each time an append operation is executed. Clearly such a scheme is expensive. The I-structure concept represents one attempt to remedy the problem [11]. An I-structure is implemented as an array-like structure where a present bit is associated with every element of the structure in the physical memory. An attempt to access an empty location in the structure will be deferred until an update operation to the same location is performed. Although it allows the possibility of concurrency between simultaneous read and write operations of an I-structure, criticism has been made of the overhead of handling the deferred read operations. Some researchers have argued that the benefit of such fine granularity does not pay for its overhead [51].

One way to represent an array in a data flow graph is to allow a token to carry the array as a single value, and use append and select actors in the program graph to access and process array elements and to construct the result array. It appears that the fine-grain advantage for a graph actor operating on arrays is lost. If a program involves many random array access operations, the overhead of transmitting the array values may be high. Furthermore, if random update functions are involved, the storage management may become expensive [3]. Criticism of proposed tree-like array storage structures in data flow computers is also well-known [51].

1.5 A Pipelined Code Mapping Scheme

The massive fine-grain parallelism which can be exploited by data flow architecture presents challenges as well as opportunities for compiler construction for such parallel machines. The functionality of data flow graphs relieves the burden of solving low-level scheduling problems due to the von Neumann bottleneck which severely restrict the power of a conventional vectorizing compiler. As a result, it may provide the foundation on which

programs can be restructured and transformed to meet both global and local data flow requirements. This certainly adds a new dimension to compilation techniques for parallel machines to which this thesis is devoted.

1.5.1 Fine-Grain Parallelism and Pipelining of Data Flow Programs

Fine-grain parallelism exists in two forms in a data flow machine level program, as shown in Figure 1.2, which shows seven actors grouped into four stages. In Figure 1.2 (a),

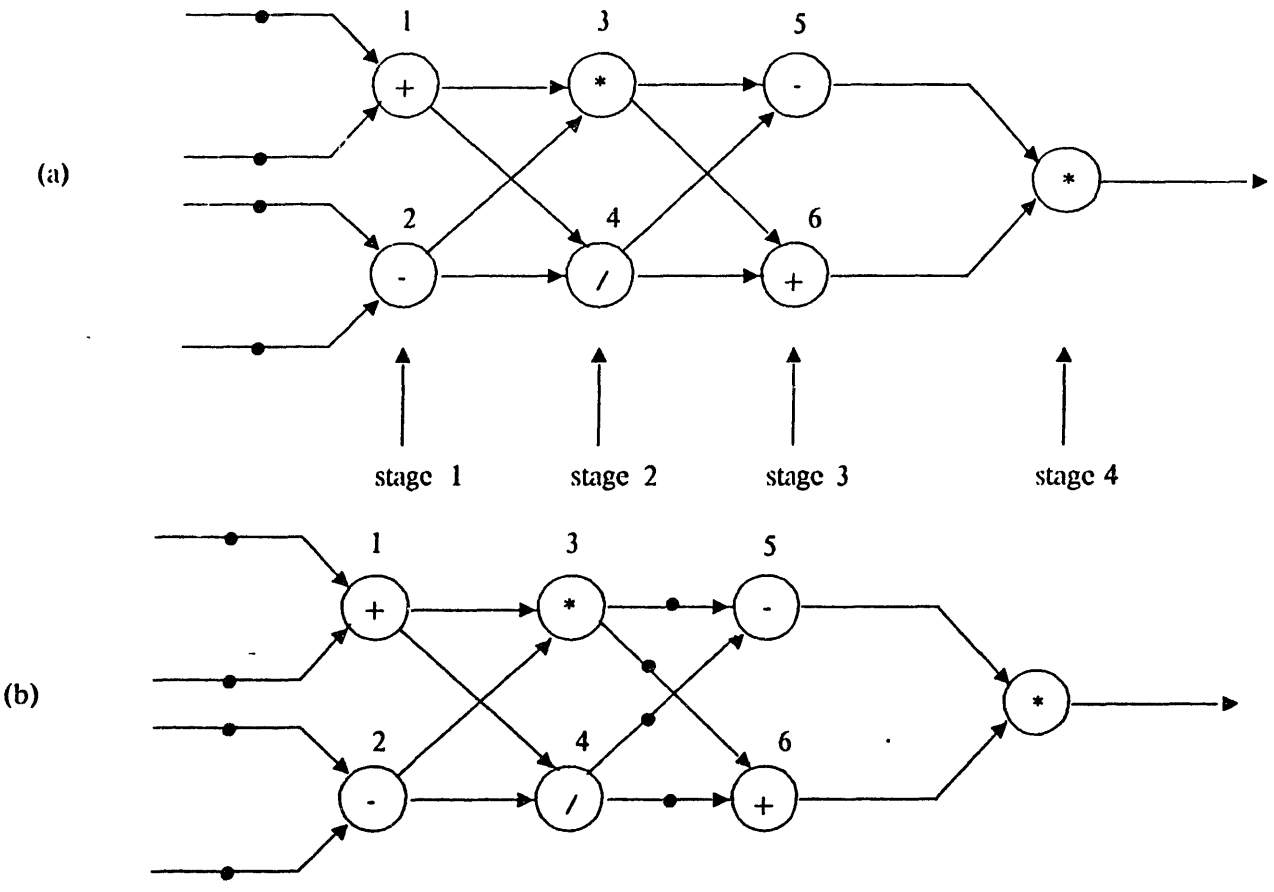


Figure 1.2. Pipelining of Data Flow Programs

actors 1 and 2 are enabled by the presence of tokens on their input arcs, and thus can be executed in parallel.[1] This is called *spatial* parallelism. Spatial parallelism also exists between actors 3 and 4, and between actors 5 and 6. The second form of parallelism is *pipelining*. In static data flow architecture, this means arranging the machine code such that successive computations can follow each other through one copy of the code. If we present a sequence of values at each input of the data flow graph, these values can flow through the program in a pipelined fashion. In the configuration of Figure 1.2 (b), two sets of tokens are pipelined through the graph, and the actors in stages 1 and 3 are enabled and can be executed concurrently. Thus, the two forms of parallelism are fully exploited in the graph.

The power of fine-grain parallelism in a data flow computer derives from machine-level programs that form large pipelines in which thousands of actors in hundreds of stages are executed concurrently. Each actor in the pipe is activated in a totally data-driven manner, and no explicit sequential control is needed. With data values continuously flowing through the pipe, sustained parallelism can be efficiently supported by the data flow architecture.

1.5.2 Data Flow Languages and Array Computations

The use of data flow languages [5] encourages an applicative style of programming which does not depend on the von Neumann style of machine program execution. An important feature of this style is that the flow of data values is directly specified by the program structure. The basic operations of the language, including operations on arrays, are simple functions that map operands to results. Data dependencies, even those involving arrays, should be apparent. This construction helps exploit concurrency in

---

1. A solid disk on an arc represents the presence of a token.

algorithms and simplifies the mapping of such algorithms into data flow machine programs. For the purpose of this paper, we choose to use Val [4] as the high level language for user programs.

Since large numerical computation programs involve many array operations, their efficient mapping is crucial in the design and implementation of high-level languages. In functional languages, the concept of an array value does not depend on storage locations in the memory. Array operations, such as the Val array append and selection operations are applicative — an array value can be created or accessed, but never modified. However, a functional semantics of array operations does not guarantee efficient implementation. For example, if a program invokes many append operations, the excessive copying may result in substantial overhead.

Array operations in large numerical computations usually take place in a regular and repetitive pattern, as shown by the example in Section 1.1. An array is usually constructed by a code block such that each element in the array is defined only once. As a result, array construction can be implemented in a way such that copying of the array is normally avoided. Another regularity is the way an array value is used in computation by other parts of a program. The selection operations of an array, clustered in a code block, often exhibit a simple indexing pattern such as in the form A[i+b], where i is the index value name and b is a compile-time constant. This regularity permits optimization in the transmission of array values between different parts of a data flow program. The goal of this thesis is to examine how array operations with such regularities can be efficiently mapped onto static data flow computers.

Since our major concern is how to utilize the regularity of array operations in the source program, we concentrate on two array creation constructs — the **forall** and **for-construct** expressions.

The **forall** construct allows the user to specify the construction of an array where similar independent computations are performed to determine each element of the array.

The following is an expression which defines a one-dimensional array X from an input array A.

```
X : array[real] :=
   forall    i in [0, m + 1]                    % range spec
   construct
       if i = 0 then A[i]
       elseif i = m + 1 then A[i]
       else
           (A[i-1] + A[i] + A[i + 1])/3
       endif
   endall
```

The **for-construct** expression, proposed as a special case of the Val **for-iter** construct, is used to specify construction of an array where certain forms of data dependencies exist between its elements. The following is a **for-construct** expression which constructs an array X based on a first-order linear recurrence, using array A and B as parameter arrays.

```
X := array[real] :=
   for  i from 0 to m + 1                       % range spec
       T : array[real] from array-empty
   construct
       if i = 1 then x0
       else A[i]*T[i-1] + B[i]
       endif
   endfor
```

Typically the body of a **forall** or **for-construct** expression is a conditional expression which partitions the index range of the result array into mutually exclusive and collectively exhaustive index subranges, each corresponding to an arm of a conditional expression. Such a conditional expression is called an *range-partitioning* conditional expression. In the

above **forall** example, there are three index subranges, i.e. [0,0], [m + 1,m + 1] and [1,m].

The two constructs just illustrated provide means to express array construction operations of the desired regularity without using explicit array append operations. Expressions based on these constructs are the major code blocks studied in this paper.

1.5.3 Pipelining of Array Operations

One objective of the machine code mapping scheme for static data flow computers is to generate code which can be executed in a pipelined fashion with high throughput. The pipeline must be kept busy — computation should be balanced and no branch in the pipe permitted to block the data flow. Furthermore, computation resources should be efficiently utilized. In particular, the usage of storage for arrays is important, because the user program usually contains vast amounts of array data to be processed..

In a data flow computation model, an array value can be regarded as a sequence of element values carried by tokens transmitted on a single data flow arc — as the array A represented in Figure 1.3 (a). In Figure 1.3 (b), the four input arcs are presented with four input arrays A, B, C, D, all are spread in time as in Figure 1.3 (a). Obviously, the sequences of input array values can be pipelined through the data flow graph.

We can observe that each actor in Figure 1.3 (b) is effectively performing a vector operation, e.g., actor 1 — vector addition, actor 2 — vector subtraction, etc, a total of seven vector operations. However, unlike the vector operations usually supported in conventional vector processors, there is no requirement that the activities of one such vector operation be continuously processed by one or a group of dedicated function units in the processor. The applicative nature of the data flow graph model allows flexible scheduling of the execution of enabled actors in the pipeline. In fact, an ideal data flow scheduler (with a sufficiently large data flow computer) will execute each actor as soon as its input data become available. As a result, the activities of the seven vector operations overlap each other, performing operations on different elements of different arrays

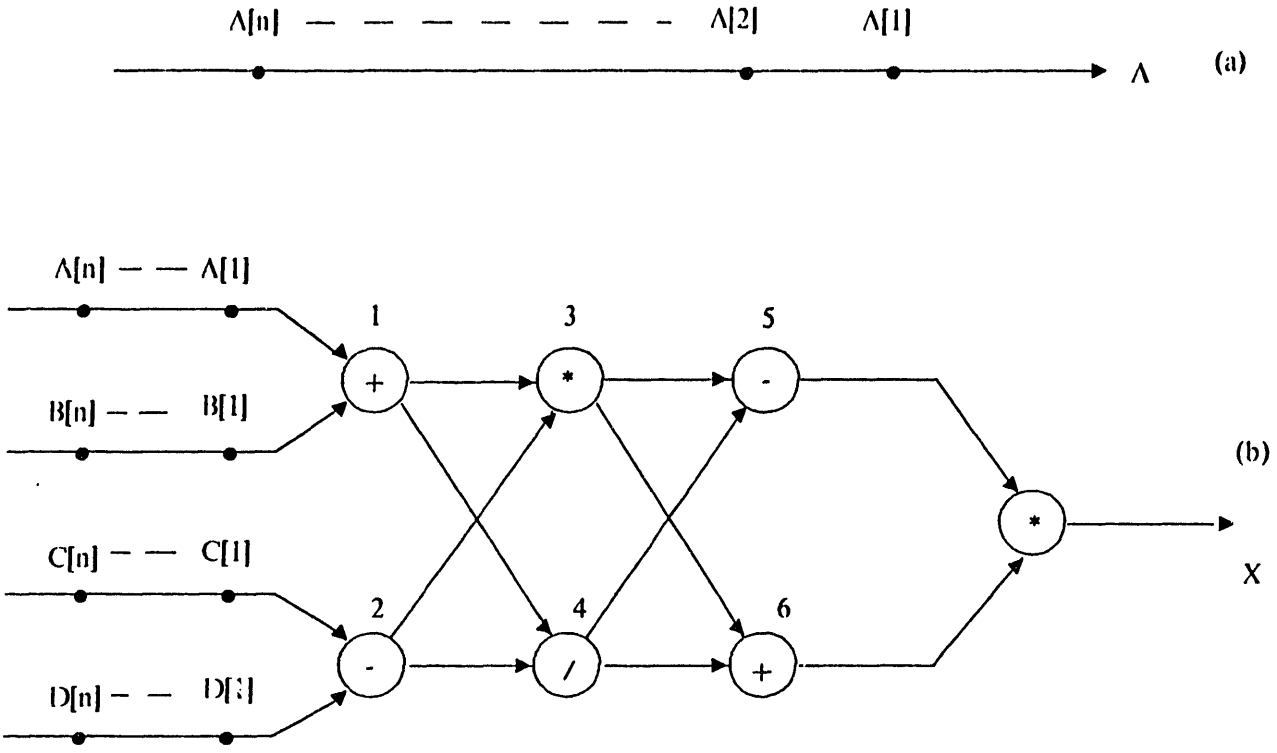


Figure 1.3. Pipelining of array operations

concurrently. Therefore, massive parallelism of vector operations can be effectively exploited by a data flow computer in a fine-grain manner: the scheduling of the physical function units and other resources for sustaining such vector operations is totally transparent to the user.

This pipelined principle of array operations can be further extended. The data flow graph in the above example corresponds to the code block in the user program which defines array X from array A, B, C, D. The core of each of the several benchmark programs for scientific computation we have studied usually consists of multiple code blocks, for example in the order of 10 - 100 code blocks. Each code block is defined by

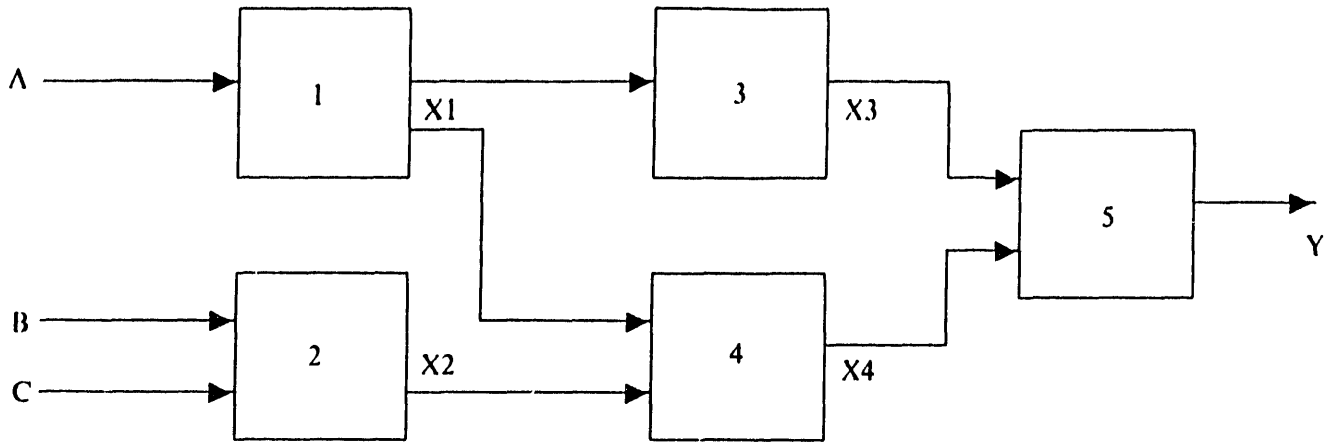


Figure 1.4. A group of code blocks

such a **forall** or **for-construct** expression [35,37,47,84].[1] A data flow graph corresponding to a program of five code blocks is illustrated in Figure 1.4. There are three input arrays A, B, C, and an output array Y. There are also internal arrays X1, X2, X3 and X4 defined by the code blocks. We are particularly interested in the case where each code block is defined by a **forall** or **for-construct** expression.

1.5.4 The Pragmatic Aspects of Compiler Construction

The issue in compiler construction for a static data flow supercomputer is to produce machine code structures that keep the processing resources usefully busy and correctly implement the computation specified in the source program. Using a functional language such as Val, the detection of parallelism is straightforward; the absence of side effects

---

1. This includes the collection of several benchmark programs provided by the six groups of scientists in the Workshop sponsored by NASA Research Institute of Advanced Computer Science [84].

allows avoidance of the complexity of such analysis for many conventional programming languages.

In contrast to conventional compilers, we are primarily concerned with both the overall and the local structure of the code. The performance of major code blocks and the effective communication between them are two key problems that a successful compiler for a data flow computer must solve.

The attention of this thesis is focused on a pipelined code mapping strategy to achieve the goal for high performance scientific computation. As will be outlined in the next section, the thesis work is mostly devoted to the algorithmic aspects of mapping blocks of code in the source program into pipelined data flow machine code structure, and the optimization techniques which can effectively implement the communication between the code blocks. The pipelined code mapping scheme developed in the thesis can serve as a basis on which a practical compiler can be built.

Of course, there are many other important issues in the compiler construction which are not covered in the thesis. However, the thesis will give a brief discussion on some of these issues in Chapter 11. The preliminary structure of a compiler is outlined and the pragmatic issues of implementation are addressed.

1.6 The Scope and Synopsis of the Thesis

1.6.1 The Scope of the Thesis

As a basis, the first part of the thesis describes a static data flow graph model as an operational model for concurrent computation, and presents the timing considerations for the graph execution on an ideal static data flow computer. Based on such an execution model, the notion of pipelining and its performance can be characterized. The thesis discusses the principles and balancing techniques used to transform certain data flow graphs into fully pipelined data flow graphs. In particular, the optimal balancing of an

acyclic data flow graph is formulated as a linear programming problem for which an optimal solution exists. As one major result, the thesis shows that the optimal balancing problem of acyclic data flow graphs can be reduced to a particular class of linear programming problem i.e. the network flow problems for which well known efficient algorithms exist. This result reverts a conjecture that such problem is computationally hard.

The second part, the kernel of the thesis, concentrates on the development of a pipelined code mapping scheme for static data flow computers. The key is the pipelined mapping of array operations in user programs. After the source language and object language are introduced and defined, the basic pipelined code mapping scheme is developed and formulated in an algorithmic fashion. The optimization of array operations is also presented in an algorithmic fashion. The major result in this part is to show that a class of program blocks (expressible in **forall** or **for-construct** expressions) can be effectively mapped into pipelined data flow graphs. The mapping scheme can handle the code blocks with conditional and nested structures frequently found in numerical computation programs. Our technique emphasizes both global and local optimization, and these two aspects are unified under the pipeline principle. The treatment of array operations is unique in the sense that information about overall program structure can be used to guide the code generation such that the massive parallelism of array operations can be exploited in a fine-grain manner by the data flow architecture.

Although our presentation is centered on the formulation of the pipelined code mapping scheme, it is also important that other related optimization techniques may be combined to improve the performance of the result data flow graphs. In the second part of this thesis, we include a short survey of several other optimization techniques which can be used for this purpose.

The third part of the thesis addresses issues which are important for extensions of this work. One important direction of further extension is the construction of a compiler based

on the pipelined code mapping scheme. The reader may find a discussion of the structure of the user programs (the program block graphs as shown in Figure 1.4), and their relations with pipelined mapping schemes for each code blocks in the first half of Chapter 11. In particular, we outline the structure of a potential compiler which can incorporate the basic principles developed in the thesis research. Much interesting work remains to be done in this area, and our limited discussion suggests possible topics for further research. Another aspect is the pragmatic impact of the pipelined code mapping scheme on the machine architecture design. This is the topic of the second part of Chapter 11.

1.6.2 The Synopsis of the Thesis

Chapter 2 describes the static data flow computation model which is the basis of static data flow architecture. It also discusses important aspects of pipelining data flow graphs, including basic concepts and performance considerations. The highest computation rate is achieved through the maximum pipelining of data flow graphs. Chapter 3 formulates the balancing of a data flow graph as a linear programming problem and discuss an algorithmic approach to balancing techniques.

Chapters 4 through 9 are devoted to the development of basic pipelined code mapping schemes. Chapter 4 specifies the representation of the source language. In particular, it introduces PIPVAL — a subset of Val used as the source language to describe the user programs to be mapped. The chapter introduces the major code blocks, i.e. array creation expressions built using **forall** and **for-construct** language constructs. The mapping of array operations organized in these code blocks are the focus of the thesis. Chapter 5 gives an overview of the basic code mapping scheme. It addresses the topic of array representations used in pipelined code mapping schemes. It also introduces a static data flow graph language — SDFGL as an object language for the code mapping scheme. Chapter 6 presents the code mapping scheme for all expressions in PIPVAL except expressions built from the two array creation constructs. Chapters 7 and 8 are devoted to

the pipelined code mapping scheme of **forall** expressions. This includes the generation of the data flow graphs as well as the optimization of array operations in them. Chapter 9 discusses the mapping scheme of **for-construct** expressions.

Chapter 10 is a survey of related optimization techniques which can be combined with the basic code mapping schemes. Chapter 11 discusses considerations of program structure and machine design to support the pipelined code mapping scheme. Important pragmatic issues for compiler construction are addressed. It also suggests topics for future research. The conclusions of the thesis are in Chapter 12.

2. The Static Data Flow Model

In this chapter, we describe the static data flow graph model as an operational model for concurrent computation. This model has evolved from a number of graph operational models for studying concurrent computation. Earlier models concentrated more on basic theoretical aspects such as decidability of properties of concurrent computations: deadlock, nondeterminacy, equivalence of program graphs, and expressive power for parallelism [7,59,85]. Later works were oriented toward operational models of practical programming languages designed for data flow computers [19,24,25,91]. The static data flow graph model that originated from this research has provided the power to express most language features found in high-level programming languages such as Val.

The goal of this thesis is to develop a pipelined program mapping scheme to efficiently exploit the degree of concurrency achievable in the static data flow model. In Section 2.1 we briefly present the static data flow graph model, outline the main features of an idealized static data flow computer as an implementation model, and introduce terminologies and notations used in discussing the model and in the rest of the thesis. A survey of other major data flow models can be found in [36]. In Section 2.2, we describe the basic concept of pipelining of static data flow graphs, the timing considerations in their execution, the concept of maximum pipelining and related performance issues, and finally the balancing of data flow graphs.

2.1 Static Data Flow Graph Model

2.1.1 The Basic Model

The basic execution model of a static data flow computer is the *static data flow graph*

model. As in most data flow models, a program module is represented by a directed graph.[1] Nodes in the graph are also called *actors.* Associated with each actor are an ordered set of *input arcs* and an ordered set of *output arcs.* The arcs specify paths over which data values can be transmitted.

The state of a computation is described by *configurations* and the *firing rules* governing the transition between configurations. Data values are denoted by placing tokens on the arcs. A configuration is an assignment of tokens in the graph. One configuration is advanced to another by *firing* of the actors. With the exception of a few *special actors* (i.e. the T-gate, F-gate, switch and merge actors to be studied later) for implementing conditional and iteration computations, the firing rules for static data flow model are quite simple:

Regular Firing Rule:

(1) an actor becomes enabled iff all its input arcs have one token and all output arcs are empty;
(2) an enabled actor may fire and, once fired, removes all tokens on its input arcs and places a token on each of its output arcs.

In Figure 2.1, we show a static data flow graph and a succession of several possible configurations for the following expression:

$$(a+b)\times(c-d)$$

---

1. A short summary of terminologies regarding the digraph and the model can be found at the end of this chapter.

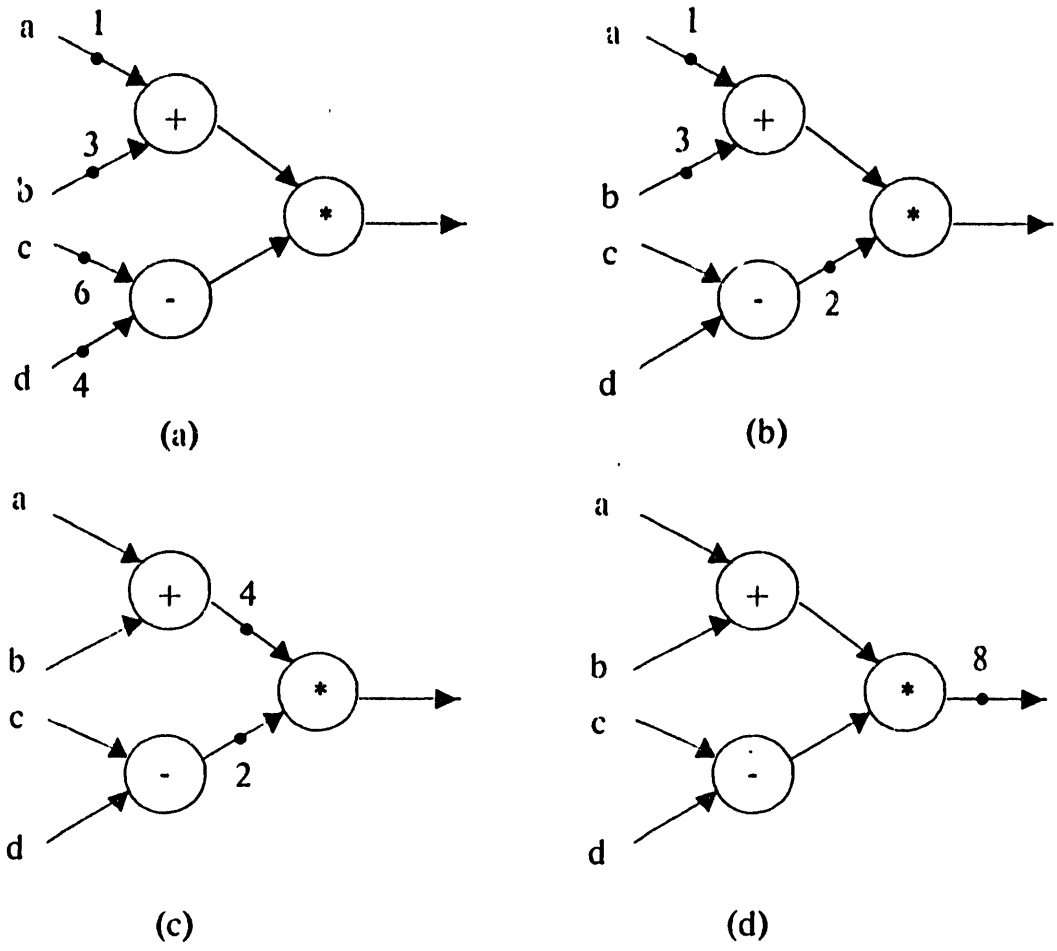


Figure 2.1. A Static Data Flow Graph

Here we adopt an earlier notation convention that a token on an arc is represented by the presence of a solid disk. Labels are used to denote values carried by the tokens, and can be omitted if irrelevant to our discussion. For simplicity, constant operands can be subsumed into the nodes.

The firing of an actor involves the computation characterized by the particular operation associated with the actor, and the result token has a new value defined by the set of values of the input tokens. We assume the set of operations is rich enough to express the

computations we are interested in, including arithmetic operations, boolean operations, relational operations, etc. An *identity* actor is a unary actor which, when fired, simply forwards its input token to each of its output arcs. As a notational convention, the function symbol of the operation to be performed by an actor is directly written inside the actor, except as otherwise noted.

In order to implement conditional and iteration expressions, we need T-gate, F-gate and merge actors which have special firing rules. We also include switch actors, although their function can be performed by using T-gate and F-gate actors. A T-gate (F-gate) actor has two input arcs: a data input arc and a control input arc which expects a token with a boolean value. The firing rules for a T-gate (F-gate) actor are:

Firing Rule for T-gate (F-gate) Actors

(1) A T-gate (F-gate) actor becomes enabled iff both data and control input arcs have a token, and all output arcs are empty;

(2) An enabled T-gate (F-gate) actor may fire; once fired, it removes the tokens from both input arcs. It forwards the data input token to each of its output arcs if the control input token has a true (false) value; otherwise the input data token is simply absorbed and no output token is generated.

The graph notation for T-gate and F-gate actors and their firing rules is presented in Figure 2.2. Note that we adopt the convention of representing control input arcs by open arrowheads.

A merge actor has three input arcs: two data input arcs and one control input arc. The two data inputs are labeled T and F respectively. A data input arc is *selected* by the

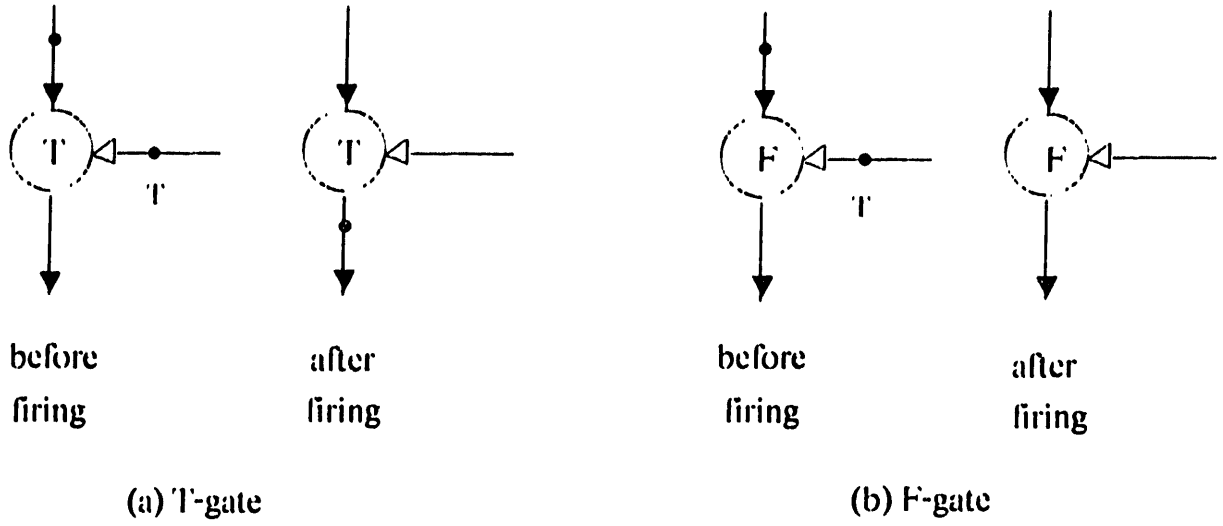


Figure 2.2. Firing Rules for T-gate and F-gate actors

presence of a token on the control input arc with a boolean value matching the corresponding label. Its firing rules are as follows:

Firing Rule for Merge Actors

(1) A merge actor becomes enabled if a token with a boolean value is presented to its control input arc, and a data token is presented to the selected input arc, and all output arcs are empty;

(2) An enabled merge actor may fire; once fired, it removes the tokens on its control input arc and the selected data input arc. A token carrying the selected input data value is placed on each of its output arcs.

The graph notation for merge actors and their firing rule is presented in Figure 2.3.

A switch actor has two input arcs: a data input arc and a control input arc which

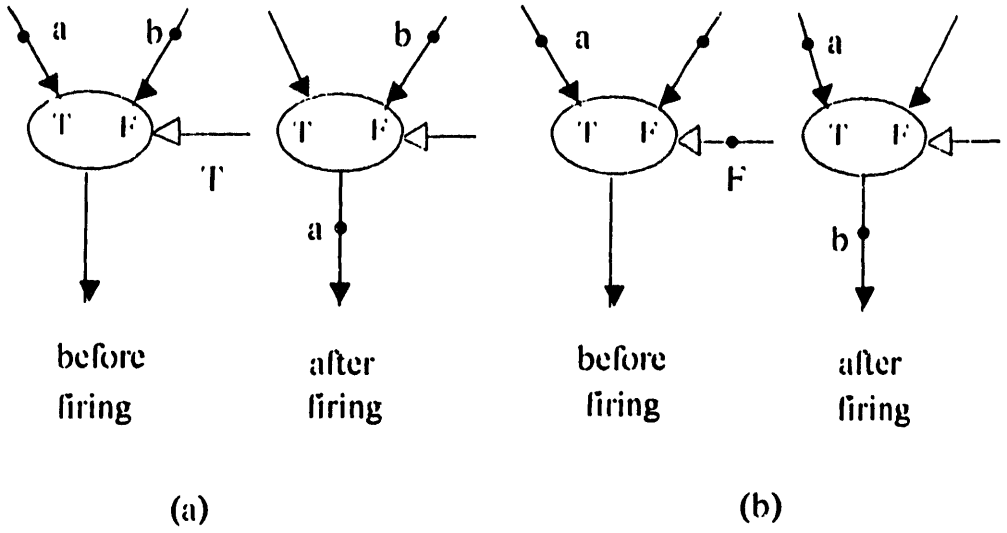


Figure 2.3. Firing Rule for Merge Actors

expects a token with a boolean value. It has two output arcs labeled *T* and *F* respectively. The firing rules for switch actors are as follows:

Firing Rule for Switch Actors

(1) A switch actor becomes enabled iff both data and control input arcs hold tokens, and all output arcs are empty;

(2) An enabled switch actor may fire; once fired, it removes the tokens from both input arcs. It forwards the data input token to the output arc labeled *T* if the control input token has a true value; otherwise the input data token is forwarded to the output arc labeled *F*.

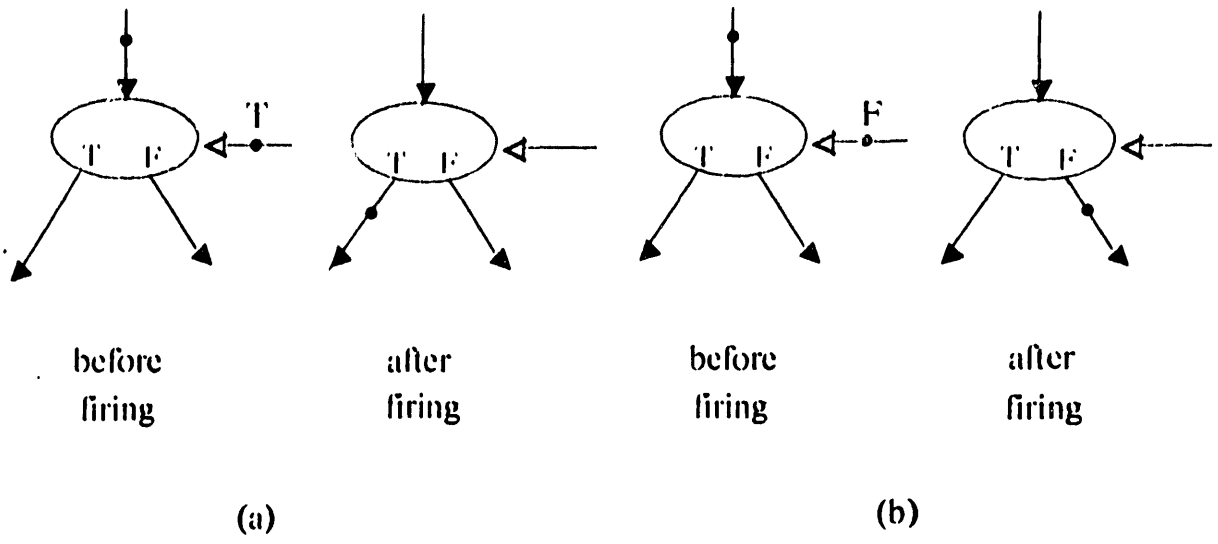


Figure 2.4. Firing Rule for Switch Actors

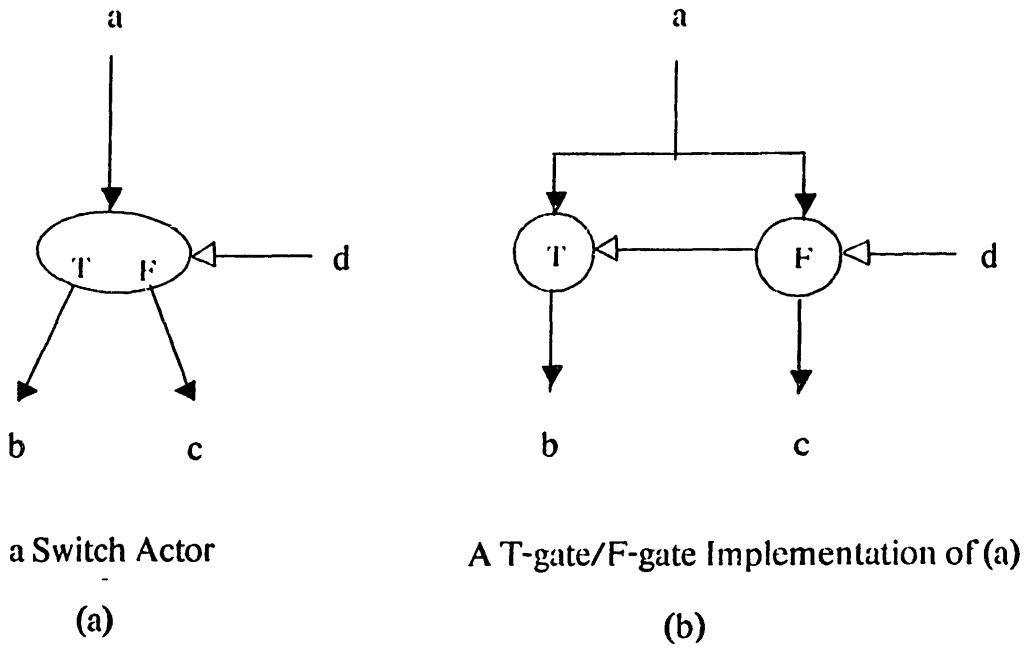


Figure 2.5. Implementation of Switch Actors Using T-gate and F-gate Actors

The graph notation for switch actors and their firing rules are presented in Figure 2.4. Using a pair of T-gate and F-gate actors, we can implement the role of classical switch actors (see Figure 2.5).

Using the special actors, a data flow graph for a conditional expression

if P(x) then f(x,y) else g(x,y) endif

is shown in Figure 2.6. As long as the computation of p, f and g does not diverge, exactly one token will eventually be generated at the output arc of the graph. Such a data flow graph is called a *conditional subgraph*. The switch actor in Figure 2.6 can be replaced by a

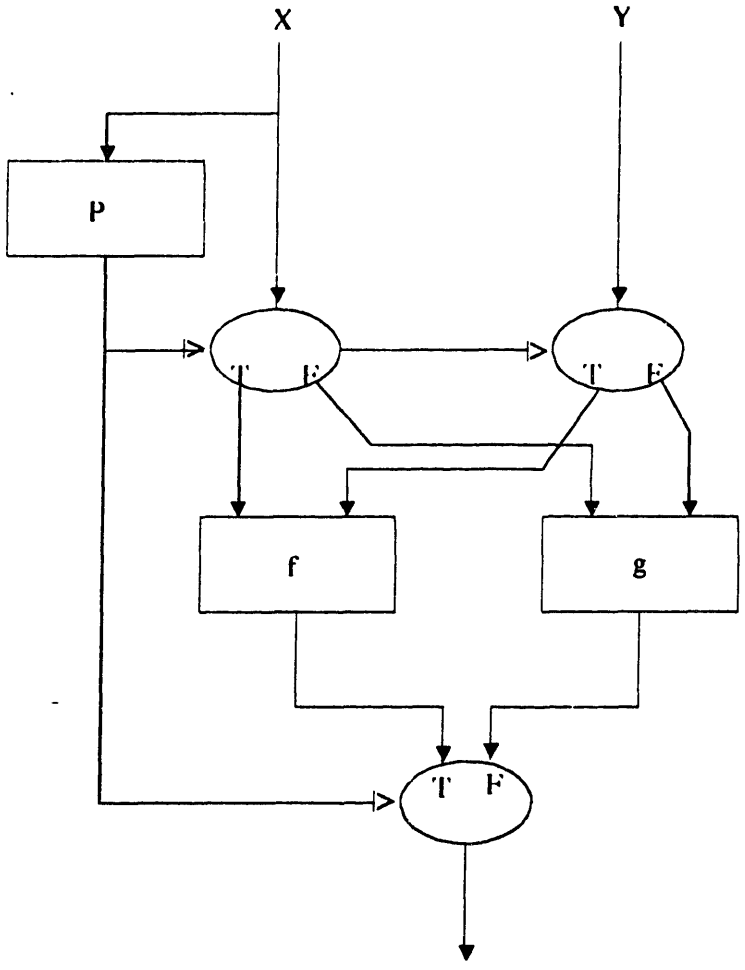


Figure 2.6. A conditional subgraph

pair of T-gate and F-gate actors according to Figure 2.5. Figure 2.7 shows the static data flow graph for the following iterative expression which computes the factorial of n.

```
for i := 0, f := 1
do
        if i > n then f
        else iter i := i+1, f := f*i enditer
        endif
endfor
```

Here the two merge actors initialize the loop value names at the first iteration and provide

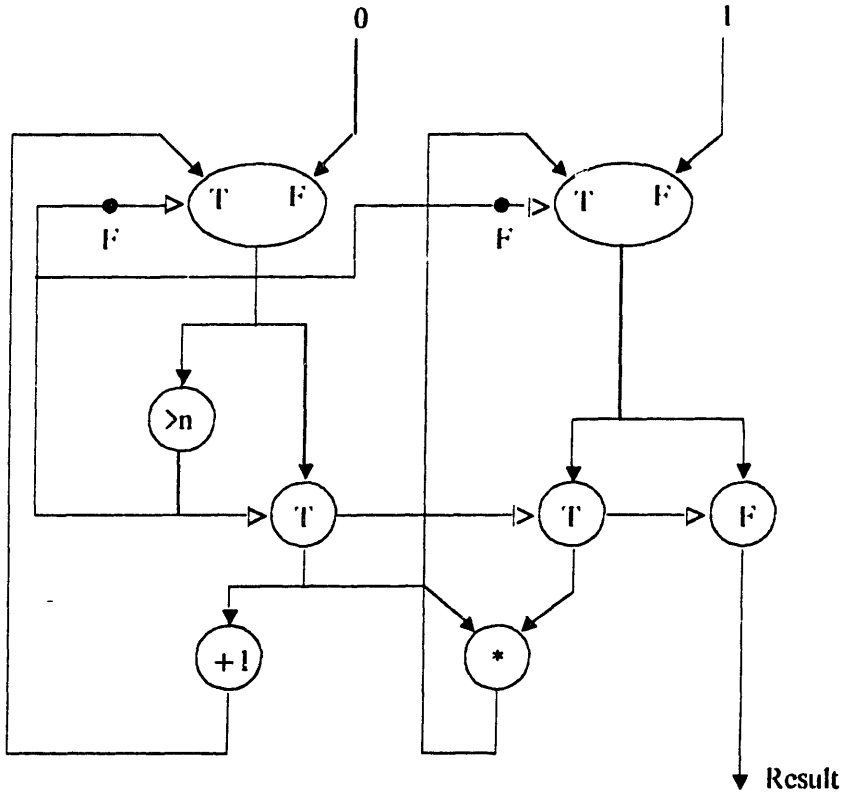


Figure 2.7. An iteration subgraph

the redefined values in successive iterations. The termination of the iteration is controlled by the test i > n in each iteration. When the test yields the value 'F', the iteration is continued with the redefined values as the inputs to the next iteration. Otherwise, the iteration will be terminated and a result token generated at the output arc of the bottom merge actor. Such a graph is called an *iteration subgraph*. For each set of input values, an iteration subgraph will generate exactly one set of result values, unless the computation diverges.

In this thesis, we only consider data flow graphs that are *well-behaved*, i.e., have exactly one set of result tokens generated at the output arc for each set of tokens presented at the input arcs [26]. In fact, the data flow graphs derived from expressions found in most user programs, including the conditional subgraphs and iteration subgraphs, are well-behaved [26].

2.1.2 Determinancy and Functionality of Static Data Flow Graphs

Recall that the state of computation of a data flow program is defined in terms of its configurations and that firing rules determine *execution sequences* corresponding to the change of states in the computation process. In general, a well-behaved data flow graph (with a certain initial configuration) may have many legal execution sequences. The determinate property of the static data flow graph model guarantees that it is necessary to examine only one execution sequence to derive the result of graph execution [82]. In terms of the results produced by the computation, all execution sequences represent the "same" computation. As we will see later, this determinate nature simplifies our study of pipelining for data flow graphs.

The determinate property of the static data flow model ensures that input/output behaviors are functional. There is no notion of locus of control, such as the notion of the program counter in a conventional computer. The execution of an actor does not have side-effects. All parallelism is faithfully represented in the static data flow graph

model—the only dependencies among actors are data dependencies. Therefore, the static data flow graph is an ideal model for mapping applicative or functional programming languages such as Val. It has been shown that a static data flow graph generated from a syntax-correct Val program is both determinate and *live*, i.e., a complete set of inputs will eventually produce a unique set of outputs [79].

2.1.3 Static Data Flow Computers

In the static data flow graph model, only one token can occupy an arc. Therefore, an actor can not become enabled unless all its output arcs are empty. Such a requirement can be implemented by a data/acknowledgment mechanism in the instruction set design of static data flow computers [35]. Actors in a static data flow graph correspond to instructions in the machine-level data flow program. Each instruction (actor) has two types of output arcs: *data arcs* (also called *result arcs*) and *acknowledgment arcs* (also called *signal arcs*). Each arc in the classic data flow graph now becomes a pair of arcs — a data arc and an acknowledgment arc. Figure 2.8 shows the data/acknowledgment arcs for the example

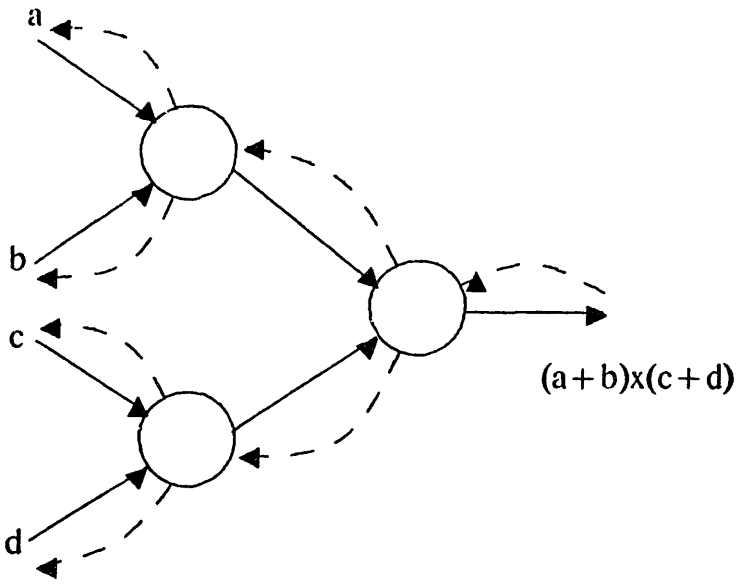


Figure 2.8. Acknowledgement Arcs

in Figure 2.1 with the acknowledgment arcs drawn as dashed lines.[1] The condition for firing an instruction in the static data flow machine now demands that a *signal token* be placed on each of the acknowledgment arcs, indicating that the corresponding data arcs are empty, i.e., the tokens placed on the corresponding data arcs from the previous firing are already consumed. Furthermore an actor, when fired, must signal its predecessor actors that the input data tokens have been consumed by placing a signal token on the appropriate acknowledgment arcs.

Data flow computer systems based on the static data flow model have been studied at MIT and elsewhere. The organization of the processing units which handle enabled instructions and initialize their execution has been described in [27,28,35]. The role, analysis and structure of routing networks are described in [17,28,29]. The structure of data flow processors and the interconnection networks for such machines are described in several publications [17,32]. The architecture of a practical static data flow supercomputer and its applications has been proposed in [28,31,35].

For the purpose of this thesis we adopt a simple, abstract model of a static data flow computer. This *idealized static data flow computer* can execute all enabled actors independently. Such a computer, being an idealization of true static data flow computers, can fully exploit potential parallelism in applicative programs. It has been used as machine model to study the implementation issues of applicative languages in [6]. Later in this chapter, we introduce assumptions regarding the timing behavior of the idealized static data flow computer.

---

1. Here we adopt the notation described in [35].

2.1.4 Terminologies and Notations

Since a static data flow graph is represented by a directed graph, or *digraph*, we will use certain terminologies and notations from graph theory. This section serves as a brief summary.

Let $G = (V,E)$ be a digraph where V is a set of *nodes* and E is a set of edges. Unless otherwise stated, both V and E are assumed to be finite. In this case we say that G is finite. Members of V and E are also called *vertices* and *arcs* respectively. If G represents a data flow graph, members of V and E can also be called *actors* and *links*. These terms will be used interchangeably throughout the thesis.

Each edge $e \subseteq E$ is associated with an ordered pair of vertices (u,v). We sometimes write this as $e = (u,v)$, or $u \rightarrow_e v$, or simply as $e_{u,v}$. When $u \rightarrow_e v$, we say e is *directed from u to v*, and u,v are called the *start node* (*tail*) and *end node* (*head*) respectively. Furthermore, we call e an *input edge* of node v and an *output edge* of node u, and we also say v is adjacent to u.

The *indegree* of a node is the number of its input edges, and the *outdegree* is that of its output edges. A node is *multi-input* (or *multi-output*), if its indegree (or outdegree) is greater than one. The set of input edges of a node is called its *input edge list* and the set of output edges of a node is its *output edge list*. A graph is called a *one-in-one-out* graph if both its indegree and outdegree are equal to one.

A *path* p is a sequence of edges $(e_1, e_2 \ldots e_k)$ such that the end node of e_i is the start node of e_{i+1} $(1 \leq i < k)$; the start node of e_1 is called the *initial node* of p and the end node of e_k the *terminal* node of p. The *length* of the path is k. Also, a node v is *reachable* from u if there is a path p from u to v, denoted by $u \mapsto_p v$. A path is *simple* if all edges and all nodes on the path, except possibly the initial and terminal nodes, are distinct. A path is a *chain* if the indegree and outdegree of each node equals one except the initial node and the terminal node.

A graph in which a number, say $w_{i,j}$, is associated with every edge (i,j) in the graph is

called a *weighted graph* and the number $w_{i,j}$ is called the *weight* of the edge. The *cost* of a path is the sum of the weights of the edges in the path.

A *cycle* is a simple path with a length of at least one which begins and ends at the same node. A digraph is *acyclic* if it does not have cycles.

Throughout this thesis, we use $|V|$ and $|E|$, respectively, to denote the number of nodes and the number of edges in the graph $G = (V,E)$.

2.2 Pipelining of Data Flow Programs

The concept of pipelined computation as a major technique to achieve concurrency has been used in diverse ways in computer architecture and organization. It certainly can be used in various levels of the design for a data flow computer as well. In this thesis, we do not attempt to address the broad spectrum of problems regarding pipelining. Instead, we concentrate on the pipelining of static data flow graphs. Such pipelining is a very effective way to organize parallel computation on a static data flow machine. In this section, we introduce the basic concepts for such pipelining and establish important criteria for its performance.

In Section 2.2.1 we first illustrate the basic concepts of pipelining in the static data flow model through some examples. In Section 2.2.2, we discuss timing considerations during program execution of the static data flow graph model. The important notion of maximum pipelining is introduced in Section 2.2.3. In Section 2.2.4, we introduce the notion of a balanced data flow graph.

2.2.1 Basic Concepts of Pipelining

Pipelining is a well-known approach in the design of conventional computers to exploit parallelism. The general approach of pipelining is to split a function into basic operations and allocate separate hardware to each basic operation. These range from components of arithmetic operations to the producer-consumer computation by CPU and

I/O processors. In a real computer system a basic operation at one level of pipelining may itself be pipelined at another level.

In this thesis, we are interested in the pipelining of data flow graphs as a model of machine-level programs for a static data flow computer.[1] For our purposes, a basic operation of the pipeline is a data flow actor in the graph. Successive operands are pipelined through actors of the graph. The goal is to structure the data flow graph in such a way that many actors in various parts of the graph may be executed concurrently. This provides an effective way to exploit parallelism in user programs.

The example illustrated in Figure 2.9 (a) is a three-stage pipeline made of four actors. When tokens arrive at *a* and *b*, actor 1 at stage 1 fires and sends results to actors 2 and 3. Once actors 2 and 3 at stage 2 fire and acknowledge receipt of their operands, actor 4 at stage 3 may fire, and actor 1 may fire again on new data, as indicated by the two sets of tokens in Figure 2.9 (b). Thus data tokens may flow continuously through the three-stage pipe. Each stage is kept busy processing units of data, one after another, as they flow through successive stages of the pipeline. Thus the computation rate of a pipeline is not dependent on the number of stages, but is determined by the processing rate of one stage.

In a typical scientific program, the machine code may be organized as a huge pipeline, perhaps hundreds of actors long and wide. If successive values of long vectors can be pipelined through the pipeline, there may be many thousands of actors in hundreds of stages in concurrent operation. The potential parallelism of such "two-dimensional" concurrent execution in a properly structured data flow program is enormous.

---

1. From now on, the terms data flow program and data flow graph are used interchangeably in our discussion when no confusion may occur.

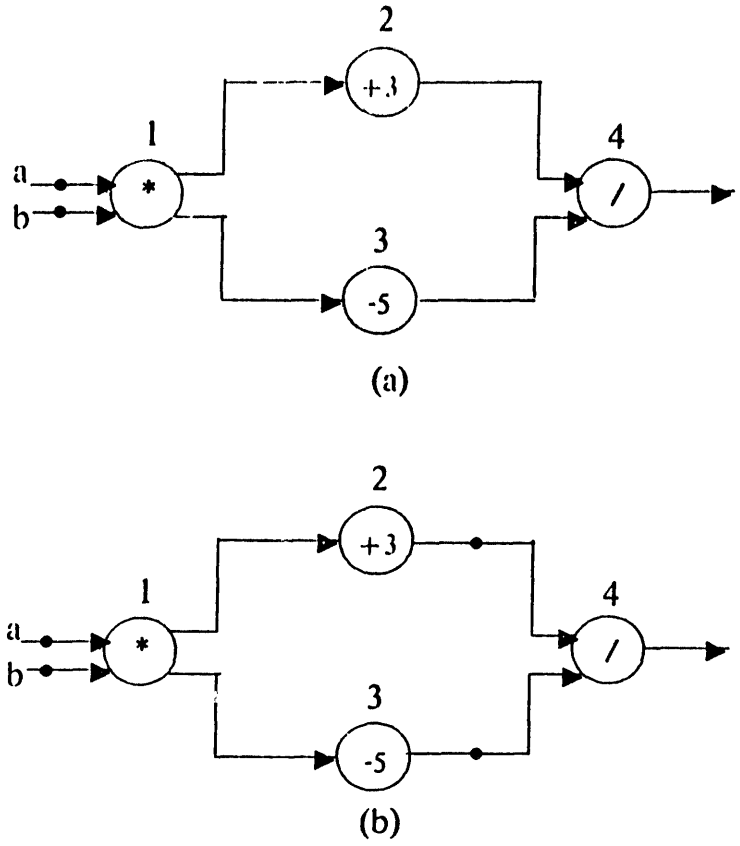


Figure 2.9. An Example of Pipelining

2.2.2 Timing Considerations

The nature of program execution on a data flow computer is asynchronous — there is no centralized control mechanism to schedule the firing of the instructions. To study performance, however, it is convenient to associate timing parameters with the static data flow graph model. In this section, we introduce two major assumptions about the idealized static data flow machine introduced earlier.

Let us consider the time needed to fire an actor in the graph. The firing of an actor includes the time needed to perform the operation specified by the actor on the operand tokens presented on its input arcs; to generate the result tokens and place them on its

output arcs; and to signal the emptiness of the input arcs such that it is ready to accept a new set of input tokens. Recall that an idealized static data flow computer can fire all enabled actors independently. The following is our first assumption.

> **Assumption 1** (A-2.1). The firing of any enabled actor can be completed within a constant time τ, where τ is a parameter of the idealized data flow machine.

Assumption (A-2.1) limits execution time for any actor in a graph. The time interval τ is an important performance measure of the machine. In a real machine, the firing time may not be a constant for actors of different types. It may even vary for the actors of same type as the processing load on the machine changes. For our purposes however, we ignore such factors and assume τ is a constant, called the *basic cycle time* (or for short, *cycle time*). We use a clock with cycle time τ as the timing reference for computations by data flow programs. We also assume, without loss of generality, that the firing of any enabled actor may only happen at the beginning of a clock cycle. This assumption gives the execution of data flow programs a somewhat "synchronous" behavior, which facilitates the study of their pipelined execution.

In a data flow program, many actors may become enabled at the same time. An important performance criteria of the machine is how long the firing of an enabled actor may be delayed. Let $t_0, t_2 ... t_i ...$ be the starting points of a sequence of machine cycle with $t_i - t_{i-1} = \tau$ for all $i > 0$. According to assumption (A-2.1), any enabled actor that starts firing at t_{i-1} will complete the execution before t_i. As a result of the firing, some other actors may become enabled during the time from t_{i-1} to t_i. We assume that the machine will not fire those newly enabled actors until all the actors currently being fired finish their execution; as soon as that happens, the newly enabled actors should be fired without further delay. Therefore, we have the following assumption.

Assumption 2 (A-2.2). All actors enabled during the interval t_{i-1} to t_i will start to fire at t_i for all i>0.

An immediate consequence of (A-2.2) is that an enabled actor X cannot start to fire before the firing of any actor enabled earlier than X. However, this does not exclude the possibility that they may start to fire at the same time. This guarantees that the machine is fair in the sense that no actor can be fired twice in a row unless all actors enabled in between start to fire.[1]

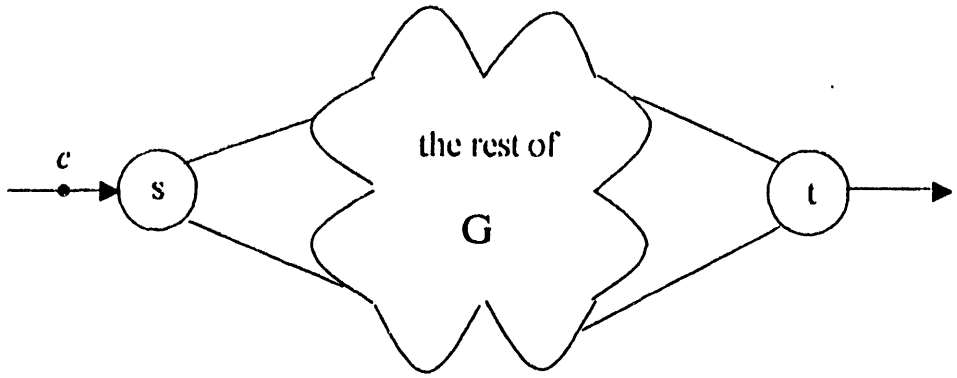
Both assumptions (A-2.1) and (A-2.2) require that the machine architecture have enough parallelism to process multiple enabled actors in a graph simultaneously. The two assumptions are related to each other in an interesting way. In a real machine with finite parallelism, it is reasonable to expect some delay between the time an actor is enabled and the time it is actually fired. On the other hand, we certainly would not want the machine to repeatedly fire some fraction of the actors without firing the enabled actors in the rest of the program. Assumption (A-2.2) suggests that every actor may experience some delay but that the maximum delay is bounded by a basic cycle time of the machine. By delaying certain enabled actors, machine resources may be devoted to the actors presently being fired, thus reasonably limiting the cycle time τ as specified in (A-2.1).

2.2.3 Throughput of Pipelining Data Flow Programs

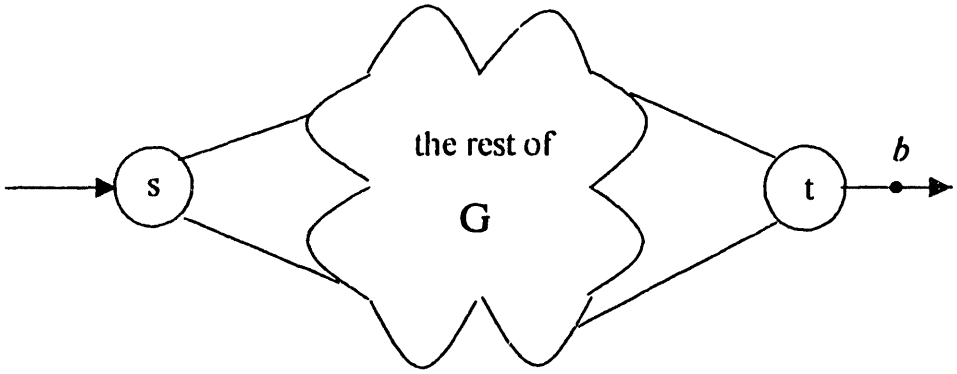
Based on the timing of the machine, we can characterize the performance for pipelined execution of data flow programs. Let us consider the data flow graph shown in Figure 2.10. We assume that the input node s of the graph G is driven by an input *source* which can generate a stream of input tokens as fast as the pipeline can use them. Furthermore, the output node t of the graph G is connected to a *sink* which can absorb

---

1. A practical implementation of fair-firing mechanisms is discussed by the author in [45].



(a) before the run with respect to c



(b) after the run

Figure 2.10. One run of a data flow graph

tokens as fast as the pipeline can generate them. Such a source (or sink) is called a *perfect source* (*sink*). If all inputs and outputs are connected to perfect sources and sinks, the pipeline is said to be under a *maximum loading* condition. Note that the perfect sources (sinks) may themselves be implemented by data flow graphs.

Figure 2.10 (a) shows an initial configuration where an input token c is presented at the input arc of G. Assume the computation starts at t_0. After a finite number of cycle times we should expect a result token b coming out from G as in the configuration shown

in Figure 2.10 (b). The sequence of firing of the actors in G caused by the input token c is called one *run* of G with respect to c.

In Chapters 2 and 3 (except Section 3.6), we restrict our attention to a very simple class of data flow graphs. This can simplify the discussion of the fundamental issues and techniques for achieving maximum pipelining. Two main features of this class of graphs are (1) the graphs are acyclic; and (2) the graphs contain no special actors, such as T-gate,F-gate, switch or merge actors. The first assumption excludes data dependencies between different runs. The second assumption provides a simple environment to facilitate the development of our framework. It follows directly that any single run of such a graph will cause the firing of each actor in the graph exactly once. This implies that all runs of the same static pipeline have the same pattern of actor usage regardless of the value carried by the token which activates a particular run. We should note, however, that the second assumption is not essential to our results and we will extend the result to more general cases with special operators arranged in conditional subgraphs (see the end of Chapter 3). In the rest of this chapter and Chapter 3, unless otherwise stated, the term data flow graphs refers to graphs from this simplified class.

Now let us consider the case when a sequence of input tokens c_1,c_2,c_3... arrives at the input of G. At time t_0, the first run with respect to c_1 is activated. The second run with respect to c_2 can start as soon as the result tokens on each output arc of I (I is the set of input nodes) generated during the first run are consumed by their corresponding successor actors. The same is true for the successive runs with respectively to c_3,c_4...., etc., and a sequence of result tokens is produced at the output arc of t. The concurrent execution of several runs of a data flow graph such as G is called the *pipelining* of G, and G itself is sometimes called a *pipeline.*

The performance of a pipeline is measured by the *activation rate* of successive runs, i.e., the rate at which input tokens can be consumed. Another key parameter in determining performance is *latency*, or the number of cycle times separating two

consecutive activations of the pipeline. Obviously, two cycle times are the minimum latency.[1] If a pipeline can run with an activation rate of $1/2\tau$, the execution of the graph is *maximally (fully) pipelined*, or simply, the graph is maximally (fully) pipelined. The simplest example which can run in a maximally pipelined fashion is a chain of actors each obeying the regular firing rule. The performance of pipelining a data flow graph is often characterized by its *throughput*, i.e., the rate at which output tokens are generated when driven by a sequence of input tokens. Obviously, the *maximally pipelined throughput* for any graph is also $1/2\tau$.

Real pipelines are usually more complex. Some actors in the graph may not obey regular firing rules. The presence of special actors may result in very different patterns of actor usage, for given runs with different input tokens. There may be cycles in the graph which imply dependencies between different runs, and thus may place constraints on the activation rate of the pipeline. These complications will be addressed in the later chapters.

2.3 Balancing of Data Flow Graphs

In general, a data flow graph may not be maximally pipelined, as illustrated by the example in Figure 2.11. Figure 2.11 (a) - (d) presents configurations during the first four cycle times of the computation, and it becomes apparent that the activation rate of this pipeline can not be higher than $1/4\tau$. Since the types of operations associated with actors do not affect the throughput of the data flow graph under our assumptions, we can omit them from the graph. Instead, we use an X inside the actor to indicate that it is enabled in the configuration shown.

A key notion closely related to the study of maximum pipelining is introduced in the following definition.

---

1. One cycle time for an actor to fire, and one cycle time for the predecessor and successor actors to fire and provide necessary inputs or signals [35].

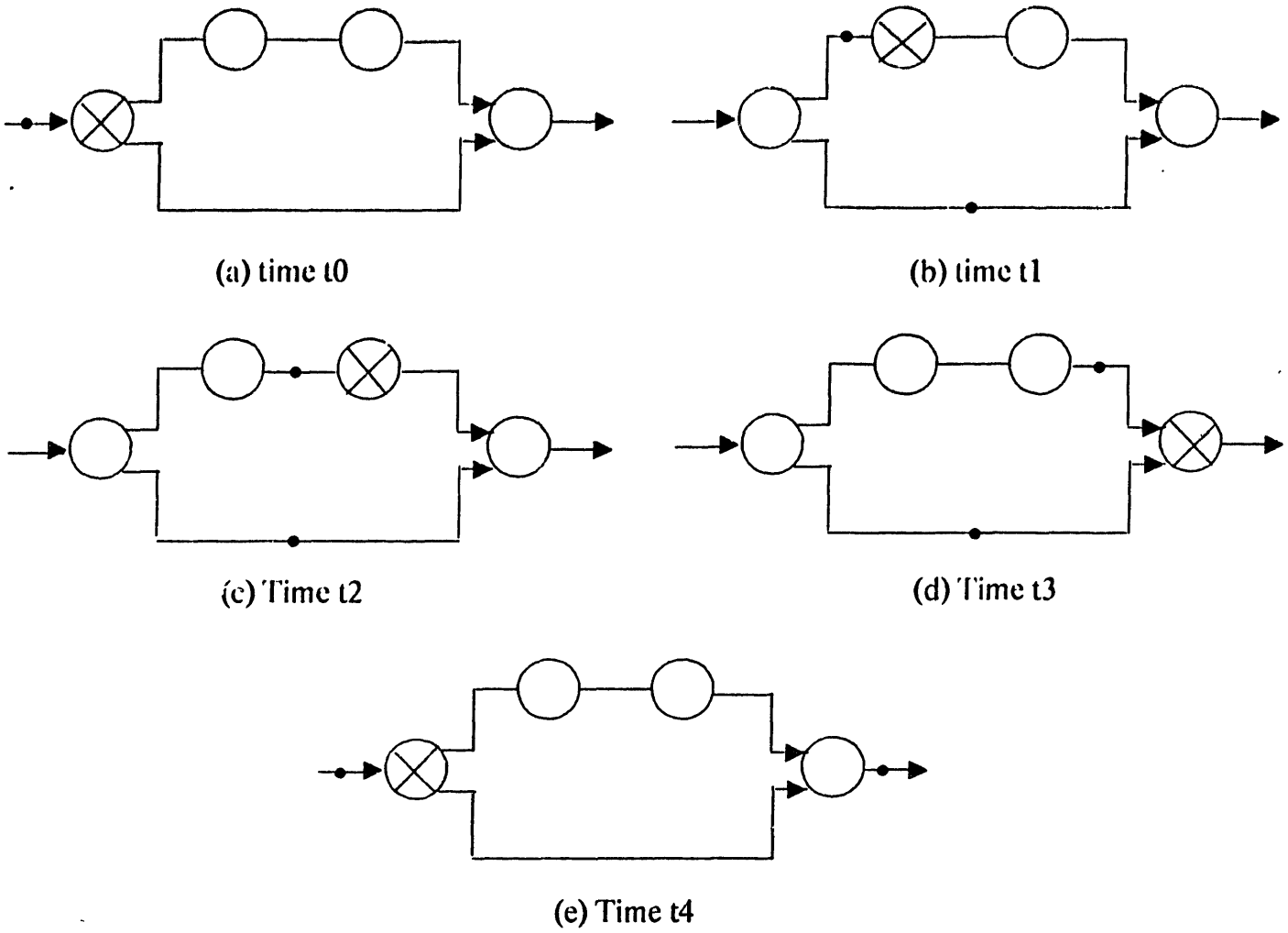


Figure 2.11. A data flow graph with Maximum Throughput of $1/4\tau$

Definition Let G be a one-in-one-out static pipeline. Let *s* be the input actor and *v* be an arbitrary actor of G other that *s*. If the lengths of any two distinct paths from *s* to *v* are equal, then G is called a *balanced* graph.

Every path from an input node to an output node through a balanced graph must contain the exact same number of actors. An apparent consequence is that a balanced graph can run in a maximally pipelined fashion [79,43]. Figure 2.12 shows a balanced

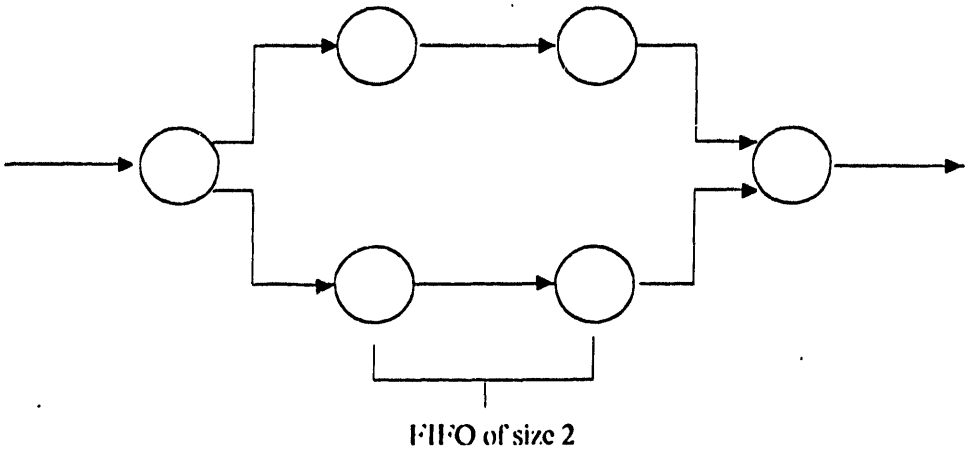


Figure 2.12. A balanced data flow graph

graph for Figure 2.11 where a FIFO made of two identity actors is introduced on the short path. It is easy to see that the graph can be maximally pipelined. Let us state this result as a theorem.

Theorem 2.1 A balanced data flow graph is maximally pipelined.

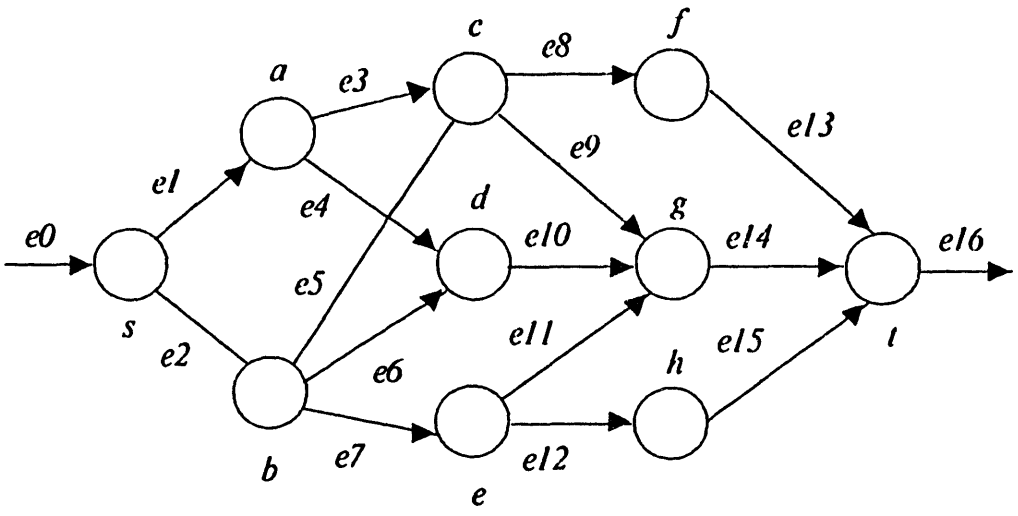
Before we prove Theorem 2.1, let us note some important facts about a balanced graph. Let $G = \{V,E\}$ be a balanced one-in-one-out graph with input actor s and output actor t. Since G is balanced, any path p between s and a node $v \subseteq V$ has the same length (say j, where j is an integer and $j \geq 0$.). We can uniquely label the actors by an integer function L: $V \rightarrow Z$ such that $L(v) = j$, where j is the path length from s to v. Now V can be partitioned into mutually exclusive and collectively exhaustive sets of nodes, or *stages* of nodes $V_0, V_1 \ldots V_m$ such that $V_j = \{v \mid \text{where } L(v) = j\}$. Note $L(s)=0$, $L(t)=m$, where we assume m is the length of each path from s to t. Obviously we have:

(1) $V_0 = \{s\}$;

(2) if node $u \subseteq V_j$ and there is an edge $e = (u,v)$ in E, then $v \subseteq V_{j+1}$;

(3) $V_m = \{t\}$.

We can also partition E into mutually exclusive and collectively exhaustive sets of edges (also called stages of edges) $E_1 \ldots E_m$ such that $E_j = \{(u, v) \mid u \subseteq V_{j-1}, v \subseteq V_j\}$. For convenience, we include the input edge to s (e_s) and the output edge from t (e_t) in the sets of edges by introducing E_0 and E_{m+1}, where $E_0 = \{e_s\}$, $E_{m+1} = \{e_t\}$.



$E0 = \{e0\}$ $V0 = \{s\}$

$E1 = \{e1,e2\}$ $V1 = \{a,b\}$

$E2 = \{e3,e4,e5,e6,e7\}$ $V2 = \{c,d,e\}$

$E3 = \{e8,e9,e10,e11,e12\}$ $V3 = \{f,g,h\}$

$E4 = \{e13,e14,e15\}$ $V4 = \{t\}$

$E5 = \{e16\}$

Figure 2.13. Stage Partitioning of a Balanced Pipeline

An example of the stage partitioning is illustrated in Figure 2.13 which is a pipeline with 9 nodes partitioned into 5 stages.

Based on such partitioning, the proof of Theorem 2.1 is straightforward.

Proof of Theorem 2.1

Assume that, at time t_0 (i.e. the beginning of the first cycle time), a token c_0 is presented at the input arc of s, e_s. Assume also that all arcs of G are initially empty and the computation starts at t_0. From the machine timing assumptions (A-2.1) and (A-2.2), we immediately have the following observations: at time t_i ($0 \leq i \leq m$) one token is presented at each arc in E_i and all other arcs are empty. Hence during cycle time τ_i all actors in stage i are enabled and fired, and no actors in other stages are enabled. The above observation can also be phrased as that a run can be advanced at its maximum speed (one stage of nodes per cycle time) under the condition that it is no blocked. The initial emptiness of the pipe is certainly a sufficient condition.

Recall that the the input of G is connected to a perfect source. Since the input arc e_s is empty at time t_1, a second token c_1 can be presented to the input at t_2. Obviously, since the first run (initiated by c_0) is advanced in its maximum rate, it will not block the second run at all. Thus, the second run can also be advanced in its maximum rate. Similarly, a new token c_2 can be presented to the input arc at t_4, etc. Therefore, the graph G can run in a maximally pipelined fashion. □.

As a remark, we note that the proof is entirely based on the partition presented earlier. This is the point where the fact that the graph must be balanced plays a key role.

2.4 Pragmatic Issues in Machine Model and Balancing

Let us briefly comment on the relation between the machine model and the data flow graph balancing considerations.

As we outlined in this section, we use an ideal static data flow machine as our machine model. The timing behavior of program execution on such a machine model is characterized by the timing assumptions (A-2.1) and (A-2.2). Under such conditions, the machine supports optimal performance of a balanced data flow graph running in a maximally pipelined fashion. The rest of the thesis will assume such a machine model is being used.

What will be the effect if, in a real machine, we consider the variation of the execution time for different types of instructions, or even the same type of instructions due to different communication delay? Our timing assumptions may still be valid if we allow the cycle time τ of (A-2.1) be considered as a bound on the firing time of all instructions. The bound should be chosen such that it can absorb not only the time difference in firing different types of instructions but also the time variations due to the machine computation and communication load.

What effect may occur if the machine has only limited parallelism? An immediate consequence is that some enabled actors may experience some delay for their firing, because the machine does not have enough computational resources. However, the assumption (A-2.2) can certainly tolerate such variations because, as long as the machine supports a "fair" firing mechanism, it is reasonable to add an average delay to the cycle time τ.

With the execution time variation of each instruction, it is sometime helpful to consider the effect of balancing from a slightly different perspective. When a static data flow graph is balanced and executed in a maximally pipelined fashion, the density of enabled instructions also achieves its maximum. In other words, balancing is a way to maximize the quantity of parallel activities in program execution. If the machine has

sufficient power of parallel processing, this also means that the pipelined execution of a balanced graph may maximally exploit the parallelism in the program.

In reality, a machine may only have limited parallelism. The introduction of FIFO actors in the graph will certainly increase the total number of executable actors. Therefore, efficient implementation of FIFOs becomes an important factor in achieving desired performance of a data flow program. A discussion of machine implementation of FIFOs is included in Chapter 11.

3. Algorithmic Aspects of Pipeline Balancing

Pipelining of data flow programs is a very attractive way to organize computations on static data flow computers to effectively exploit parallelism in programs. In order to achieve maximally pipelined throughput, a data flow graph must be balanced. The basic technique is to transform an unbalanced data flow graph into a balanced graph by introducing FIFO buffers on certain arcs. For the purpose of this discussion, a FIFO of size k is equivalent to a chain of k *identity* actors.[1] The procedure to perform such transformations is called *balancing* the data flow graph.

An example of balancing is illustrated in Figure 3.1. Figure 3.1 (a) shows an unbalanced graph with eight nodes. Figure 3.1 (b) is the result graph after balancing the graph in Figure 3.1 (a). Two FIFO buffers, with size two and three respectively, are introduced on arcs (f,t) and (e,t) as shown in Figure 3.1 (b). Each buffer is denoted by a box with a number denoting the size of the buffer. In this case, the total size of buffering introduced for balancing is five.

In general, there may be more than one balanced version of a data flow graph. For example, Figure 3.1 (c) presents another balanced graph for Figure 3.1 (a). However, the total amount of buffering in Figure 3.1 (c) is three, a considerable savings compared with what is needed in Figure 3.1 (b). Since the minimum amount of buffering needed to balance the original pipeline is three, Figure 3.1 (c) is an optimal solution. In general, we have the following definition.

Definition Let G' be a balanced graph for G. If G' uses the least amount of buffering among all balanced graphs of G, then G' is called an *optimal balanced graph* of G. A balancing procedure to transform a data flow graph into an optimally balanced data

---

1. There are other ways to implement FIFO buffers [35]. We will defer a discussion of these different implementations until Chapter 11.

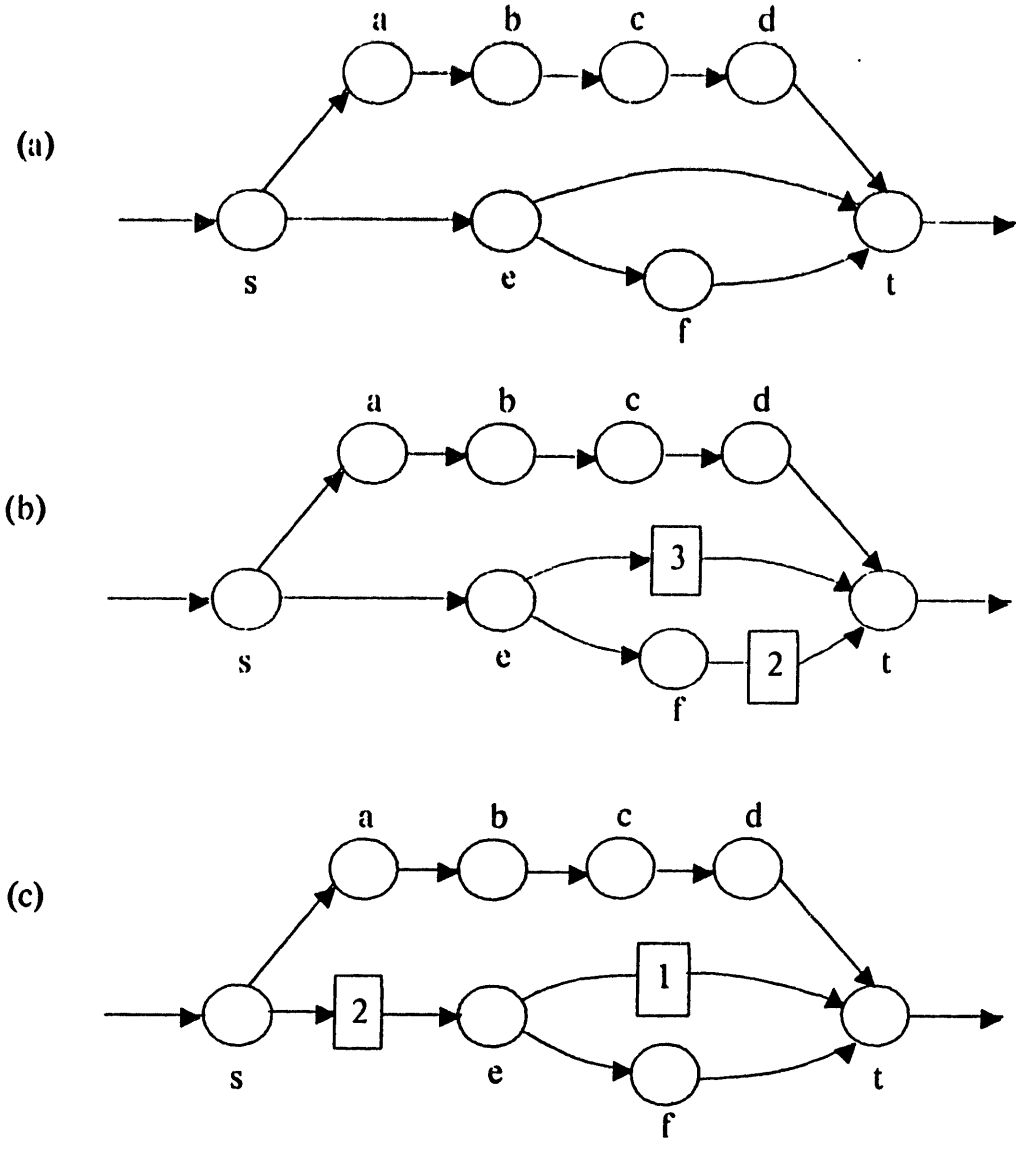


Figure 3.1. An Example of Balancing

flow graph is called *optimal balancing procedure.*

In organizing data flow programs for maximum pipelining, an efficient algorithmic procedure for optimal balancing is important. Earlier work to find such an algorithm can be found in [43,79]. In applications where a data flow program may consist of hundreds of

instructions, any feasible solution to optimization must rely upon computer programs. A conjecture was made in [43] that such optimization is not computationally tractable, and hence makes impractical the construction of a compiler which can perform automatic optimal balancing on data flow programs. Fortunately, this conjecture is not true. In this chapter, we show how techniques from linear programming can be applied to solve optimal balancing problems.

In Section 3.1, we introduce the concept of weighted data flow graph which facilitates our later discussion. In Section 3.2, we briefly review the previous related work in balancing techniques using graph-theoretic terms, identify problems in the approach, and suggest a new solution. By doing so, we not only simplify the formulation of our major results, but also get a better insight of the balancing problems which is important for the discussion in the succeeding sections. In Section 3.3 and 3.4 we present a different formulation of balancing and optimizing problems from our previous work. As an important result, we show that these problems are equivalent to a class of linear programming problems. This class can be reduced to a class of known network flow programming problems which have practical algorithmic solutions. Hence, the construction of an automatic program to perform such optimization is computationally tractable. In Section 3.5, we discuss the extension of the balancing techniques to a broader class of data flow graphs.

3.1 *Weighted data flow graphs*

In the discussion of balancing problems using graph-theoretic terms, it is often convenient to use what is known as a *weighted data flow graph*. A weighted data flow graph $G = (V,E)$ is a weighted directed graph where each node in V represents an actor, and each arc $e = (u,v)$ in E, weighted by $w_{u,v}$, denotes that a chain of length $w_{u,v}$ exists between node u and v (see the definition of a chain in Chapter 2). Obviously, all weights must be positive integers.

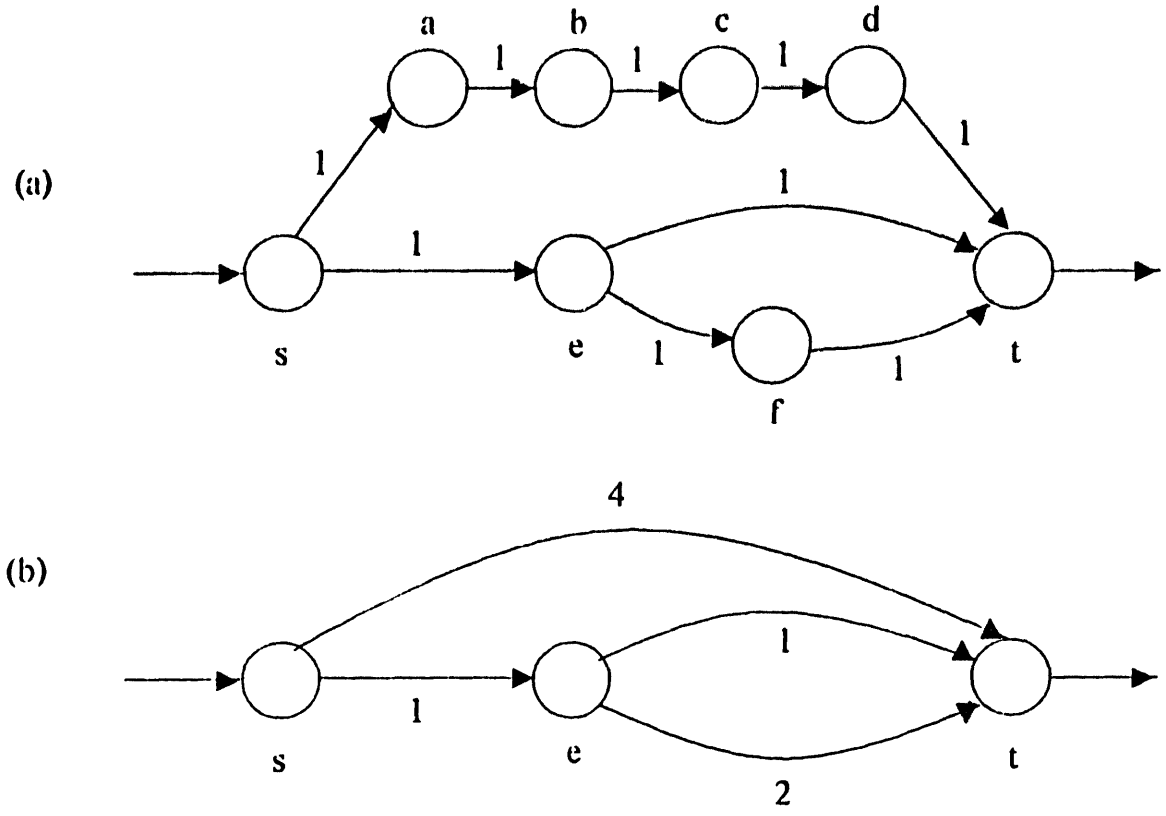


Figure 3.2. A weighted data flow graph

For example, Figure 3.2 shows two weighted graphs, where a weight k is written next to each arc to denote a chain of length k. We note that all arcs in Figure 3.2 have weight 1. We also allow weighted arcs with a weight greater than 1. For example, a chain of m nodes can be replaced by an arc which has a weight equal to the sum of the weights of those arcs. Figure 3.2 (b) shows a weighted graph which is equivalent to (a), with each chain of nodes replaced by an arc with proper weight. The notion of balancing and optimizing can be extended naturally to a weighted graph.

3.2 Related Work on Balancing Techniques

3.2.1 Balancing Techniques

Balancing a data flow graph requires determining the set of arcs where FIFO buffers should be introduced and computing the size of each buffer. Let $G = (V,E)$ be a one-in-one-out weighted data flow graph with an input node s and an output node t. Let v be an arbitrary node. The balancing techniques presented in [43] and other related work [79] are essentially based on the following observation:

> **Observation 3.1.** Since the cost of the longest path from s to any node v cannot be reduced further, one should introduce buffers on all other paths from s to v to make their costs equal to that of the longest path.

An important step in these balancing algorithms is to identify the longest path from s to v for each v in V. Mathematically, this is equivalent to computing a *max-cost function* $L : V \rightarrow Z$ (Z is the set of non-negative integers), where $L(v)$ is the cost of longest path from s to v for any $v \subseteq V$. Once this step is done, we may take a shortcut to determine the location and size of each FIFO buffer to be introduced. The key to the shortcut is to not alter the cost of longest path from s to any node v in V. Or, to state it mathematically, we observe the following invariant:

> *Max-cost Invariant*: Let G' be a balanced graph of G as a result of applying some balancing procedure. Let L and L' be the max-cost functions in G and G' respectively. Then for any node v, $L(v) = L(v')$ should hold.

The decision to keep such an invariant results in a very simple balancing algorithm.

In fact, both the locations and sizes of the FIFO buffers can be determined immediately from the max-cost function of the pipeline. An algorithm for balancing a one-in-one-out data flow graph is presented below:

Algorithm 3.1. A Balancing algorithm for One-in-one-out Data Flow Graphs.

Input: An one-in-one-out weighted data flow graph $G = (V,E)$ with input node s and output node t

Output: A balanced graph G'.

Steps:

Step 1: Compute the max-cost function L for G

Step 2: For each arc $e=(u,v)$ in E construct a buffer of size $L(v) - L(u)$ and insert the buffer on e.

Step 3: Return the result graph.

Step 1 is equivalent to the problem of finding the longest paths from a source node to all other nodes in an acyclic directed graph. This can be accomplished efficiently using some known graph-theoretic algorithms. For example, the Dijakstra algorithm for finding the shortest paths from a source node to all other nodes [8,40] can easily be modified to compute the above single source longest path problem. In the following we briefly outline such a solution.

Since the graph is acyclic, we can proceed by first performing a *topological sort* [65] of the node in G. We determine a labeling of the nodes in V with integers 1,2,...n (where $n = |V|$) such that if there is an arc (i,j), then $i < j$. The graph in Figure 3.3 is topologically sorted. Note that s and t are labeled by 1 and n respectively. Recall from graph theory that a digraph can be topologically sorted if it is an acyclic graph. Therefore, this step of topological sorting can detect if the graph to be balanced is indeed acyclic. Since the balancing algorithm to be presented only applies to acyclic graphs, the benefit of including

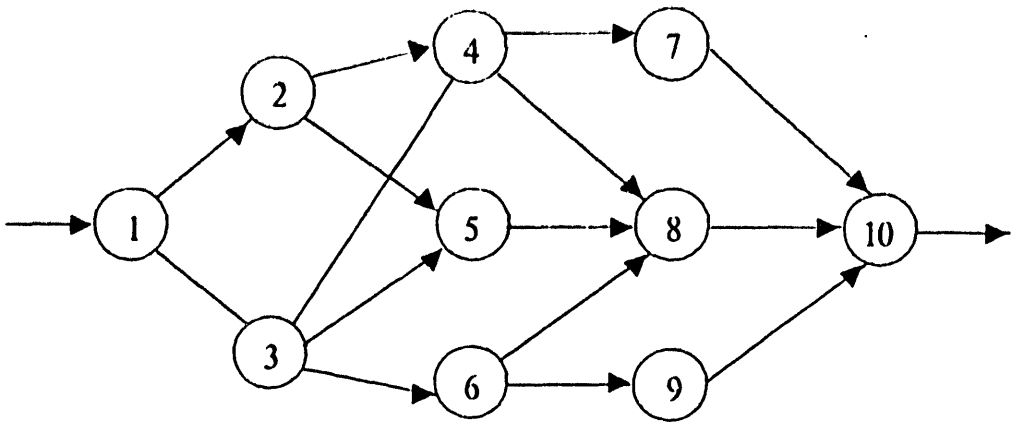


Figure 3.3. A Toplogically Sorted Graph

such a step is obvious.

Once the nodes in G are topologically sorted, the construction of an algorithm to compute the max-cost function L becomes straightforward. Clearly, $L(s) = 0$. Node 2 can only be reached from node 1, and therefore

$$L(2) = L(1) + w_{1,2}.$$

Node 3 can only possibly reached from node 1 and 2, hence

$$L(3) = \max\{L(1)+w_{1,3}, L(2)+w_{2,3}\}$$

Similarly, the general expression to compute $L(k)$ can be written as

$$L(k) = \max\{L(i)+w_{i,k}\} \text{ for all } i < k$$

Based on the above observation, construction of an algorithm for computing the max-cost function is straightforward. If we use the adjacency list data structure, the time complexity of such an algorithm is $O(|V||E|)$. For the graphs we are interested in, the indegree of any node are bounded by a small constant. thus we can use $O(|V|)$ to represent $O(|E|)$. Thus, the time complexity of computing the max-cost function is $O(|V|^2)$.

Now let us study the time complexity of Algorithm 3.1. The time for Step 1 is $O(|V|^2)$ as in the above analysis. Step 2 is executed for each arc in E. Therefore, the time

complexity of the entire balancing algorithm is $O(|V|^2)$.

3.2.2 Relation between Balancing and Optimization

One philosophy regarding optimization is to separate the optimal balancing procedure into two phases: (1) perform balancing using an algorithm similar to that of Algorithm 3.1; (2) rearrange the buffering to achieve an optimal buffer configuration. However, the solution to the second part of the problem has not been successful, as was indicated by the conjecture mentioned at the beginning of this chapter.

Let us use two examples to illustrate the problems encountered in applying such an optimal balancing strategy. The first example is a weighted one-in-one-out graph as shown in Figure 3.4 (a). The first step is to apply Algorithm 3.1 to derive a balanced graph as shown in Figure 3.4 (b). Two buffers of size 10 and 15 are introduced on arc (3,5) and (4,5), so the total buffering is 25. The label written inside each node is the number derived from the topological sorting. We can easily observe that the graph in Figure 3.4 (b) is not optimally balanced. Figure 3.4 (c) shows an optimally balanced graph for G which uses only a buffering of 15. Through optimizing, the amount of buffering is reduced by 40%.

Another optimizing procedure is to perform a transformation which propagates some buffering back through nodes which have a larger outdegree than indegree [43]. Such a transformation will usually produce a graph that uses less buffering. For example, in Figure 3.4, we can propagate back a buffering of size 10 through node 2, reaching the solution in Figure 3.4 (c).

However, it is very difficult to predict the effect of propagating some buffers back through a node which has a smaller indegree than outdegree. Let us look at the example in Figure 3.5 (a) which is similar to Figure 3.4 (a) except that it has one more arc from node 3 to 4. Applying Algorithm 3.1, we derive the balanced graph in Figure 3.5 (b) with 3 buffers introduced on arcs (2,4), (3,5) and (4,5) respectively. At this point it becomes tricky to use the scheme outlined above to perform optimizing. For example, node 4 has an indegree of

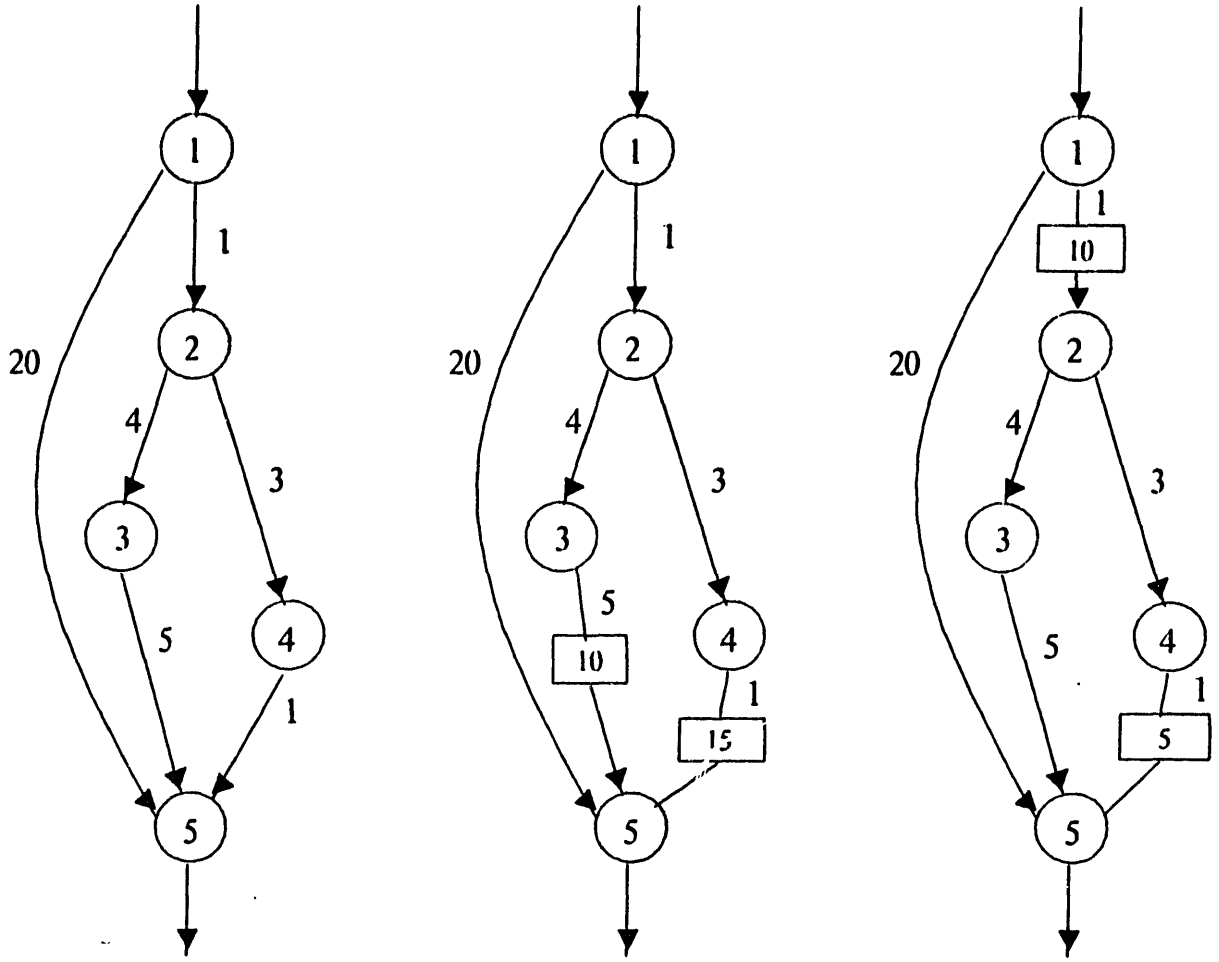


Figure 3.4. An Example of Applying Algorithm 4.1

1 and an outdegree of 2. It is difficult to envision any benefit in propagating buffers from its output arcs back to its input arcs. Assume that we have propagated a buffer of size 10 back through node 4 and reached the graph in Figure 3.5 (c). It appears that the graph in Figure 3.5 (c) is less desirable then that in Figure 3.5 (b), since the total buffering has increased by 10. However, from Figure 3.5 (c) we can propagate a buffer of size 10 on arcs

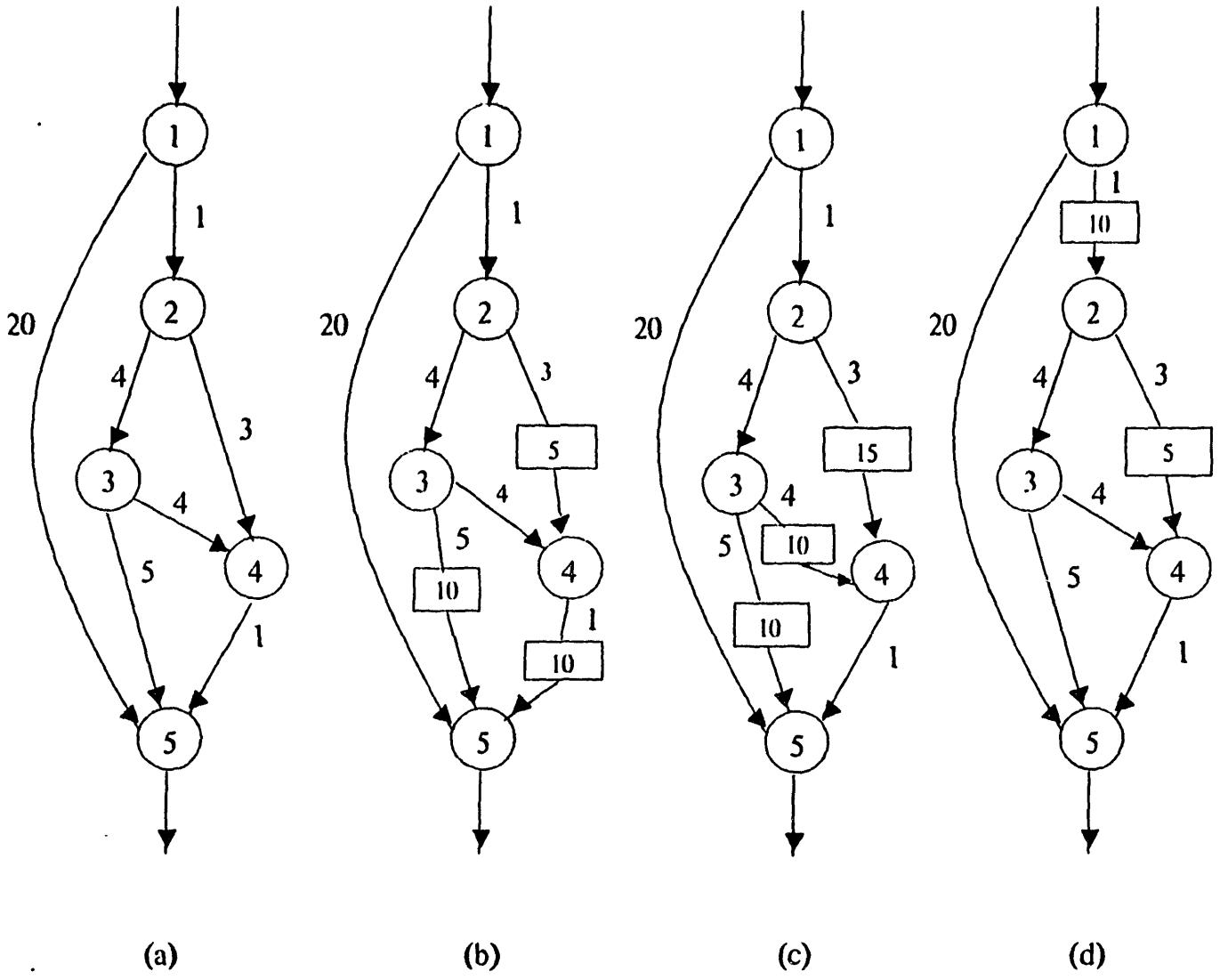


Figure 3.5. Problems in Classical Balancing Approach

(3,5) and (3,4) back through node 3, which can be further propagated back through node 2 together with a buffer of the same size from arc (2,4). This leads us to the configuration in Figure 3.5 (d). We observe that Figure 3.5 (d) is an optimally balanced graph with a total buffering of 15.

The above example indicates that optimal balancing is a global optimization process;

hence it is difficult to determine the effect of moving buffers around based only on local information about a particular node. In order to find a better solution, let us examine the decisions made in the Max-cost Invariant based on Observation 3.1. Step 2 in Algorithm 3.1 is a direct consequence of such a decision. Since the cost of the longest path from s to any node can not be changed, buffers are always introduced on the input arcs of multi-input nodes, i.e., on the last arcs of any paths which need them. Although this decision makes the balancing algorithm simple, it is somewhat arbitrary in terms of optimization. For the example in Figure 3.5, we must violate this decision in order to achieve optimization. This raises the question: is the max-cost invariant essential to the balancing process? In the rest of this chapter, we will answer this question.

3.3 A Linear Programming Formulation of the Optimization Problem

The examples presented in the last section clearly show the weakness of the optimal balancing approach proposed in [43]. In this section, we take a very different approach to attack the problem. Instead of separating the optimal balancing strictly into two phases, we view the entire analysis as one combinatorial optimization problem. We do not rely upon ad hoc decisions such as the Max-cost Invariant requirement. Instead our decision process is based on the set of constraints imposed by the structure of the graph itself. Both the locations and sizes of buffers are computed under the same theoretical frame work. In fact, under the new scheme we can transform an optimization problem into a particular class of combinatorial optimization problem known to have efficient solutions.

Let us first introduce a mathematical formulation of an optimal balancing problem. Let $G = \{V,E\}$ be a weighted data flow graph for which we have computed the max-cost function L, using the algorithm outlined in the last section. The nodes in V are numbered (according to the topological sort) by integers 1,2...n, where $n = |V|$ with s numbered by 1. We assume that G is balanced by some process; hence the cost of any path from s to a node v in V has a unique value. In mathematical terms, we can define a labeling function f for a

balanced graph which associates each node in V with an integer value, i.e., f: $V \rightarrow Z$, where Z is the set of integers. As a convention, we use u_i to denote $f(i)$, i.e., $u_i = f(i)$ for all nodes i where $1 \leq i \leq n$. We can interpret u_i-u_1 as the delay of the firing of node i with respect to node 1 (i.e., node s) for a particular run.

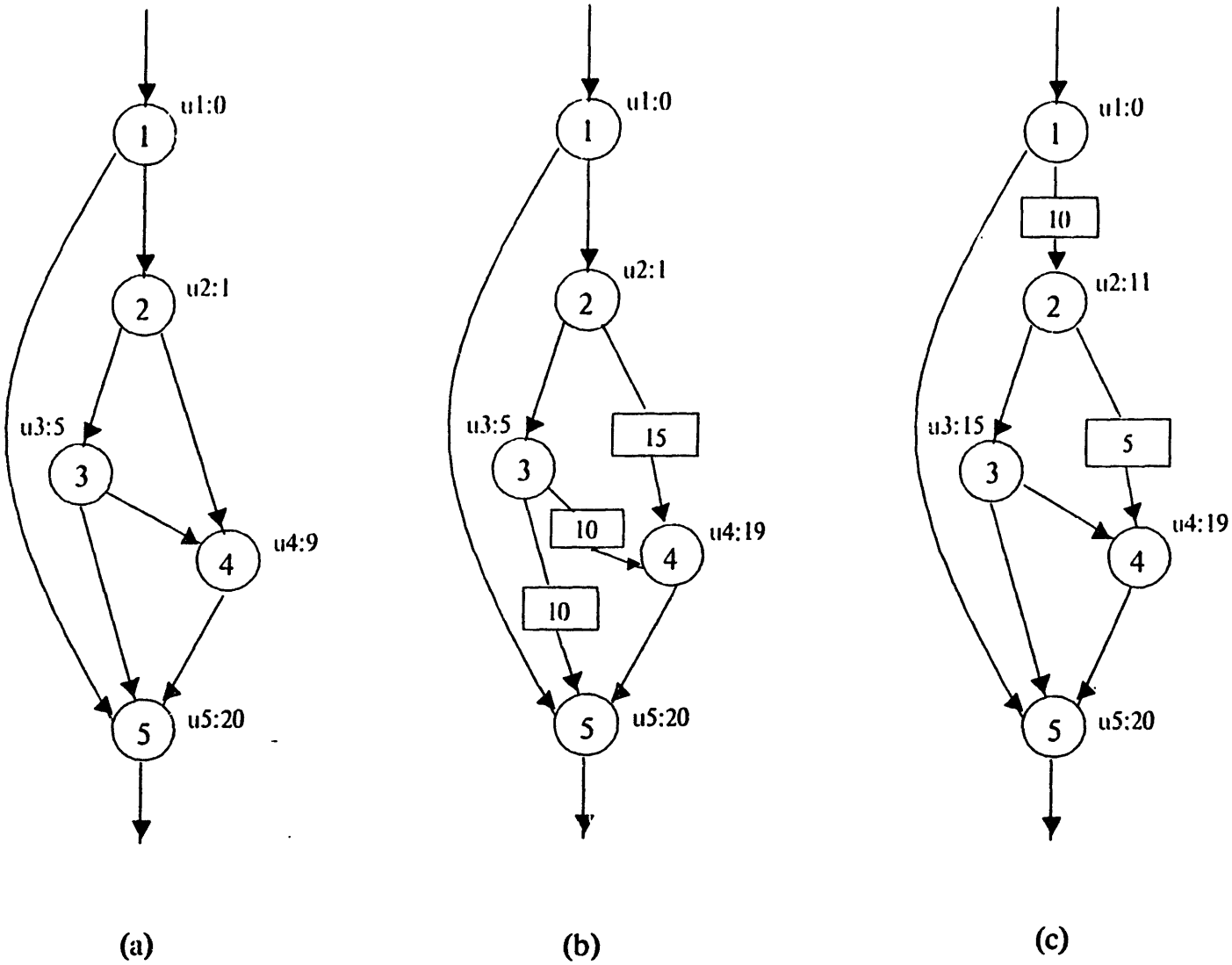


Figure 3.6. Delay Changes due to Buffer Moving

i	1	2	3	4	5
L(i)	0	1	5	9	20
$\overline{L}$(i)	20	9	5	1	0
$\overline{L}$(5) - L(i)	0	11	15	19	20

Figure 3.7. The Earliest and Latest Firing Times

Different balanced graphs may correspond to different labeling functions. For example, the Max-cost Invariant implies a labeling function such that u_i-u_1 = $L(i)$ for all nodes i. In other words, it requires that each node i in the graph be fired at the earliest possible time (called the *earliest firing time*) determined by $L(i)$. However, since the graph is one-in-one-out only the total delay from s to t is important. To ensure that such delay does not increase, none of the nodes on the longest path from s to t can have a time delay longer than the minimum delay. However, for nodes not on this path, the time delay may be allowed to slip to a certain extent. This gives us some freedom to adjust the buffer configuration in a graph for optimization purposes. Increasing or decreasing buffers on an arc can cause corresponding changes in time delay in the firing of some nodes on the graph. Figure 3.6 shows such delay changes due to buffer adjustment of Figure 3.4. Figure 3.6 (a) shows the values of u_i for the balanced pipeline in the result graph of Algorithm 3.1. Propagating a buffer of size 10 back through node 4 is equivalent to changing (by 10) the delay of firing time for node 4. Thus we have a new labeling function as illustrated in Figure 3.6 (b), where u_4 has changed from 9 to 19. After further moving the buffers back through nodes 3 and 2, u_3 and u_2 have been changed from 5 and 1 to 15 and 11

respectively. The result graph is shown in Figure 3.6 (c).

We should note that the adjustment of the labeling function cannot proceed without constraints. For example, in Figure 3.6 the time delay of node 4 can not exceed 19; otherwise the total delay from the input node 1 to the output node 5 will exceed 20. In fact, the maximum time delay of any node i is related to the cost of longest path from *i* to *t*. To compute such cost, let us reverse the direction of every edge in *G*, and name the result graph $\overline{G}$. $\overline{G}$ is a one-in-one-out graph with node *n* as its input node and node 1 as its output node. The nodes in $\overline{G}$ are still topologically sorted, but the order is reversed, *n*, *n*-1...2,1. Let $\overline{L}$ denote the max-cost function in $\overline{G}$, i.e., $\overline{L}$ (*i*) denotes the cost of the longest path from node *i* to node *n* in $\overline{G}$. The *latest firing time* for node *i* can be expressed by $L(n)-\overline{L}(i)$. Figure 3.7 shows the values of both $L(i)$ and $\overline{L}$ (*i*) for each node *i*, together with its latest firing time.

We have just seen that u_i is constrained by both $L(i)$ and $\overline{L}$. We can directly relate the constraints for u_i to the weights of each arc in the original graph. Recall that, in any balanced graph, the size of buffer $b_{i,j}$ to be introduced on arc (i,j) is $(u_j - u_i) - w_{i,j}$, where $w_{i,j}$ is the weight of arc (i,j) in the original graph. Since all buffers should have nonnegative size, we have $b_{i,j} \geq 0$ for all arcs (i,j). Therefore, the following set of linear inequalities should hold:

$$u_j - u_i \geq w_{i,j}. \qquad \text{for all arcs } (i,j)$$

We also add another constraint so that no extra delay from input to output is introduced:

$$u_n - u_1 = w_{s,t} \qquad \text{where } w_{s,t} = L(t)$$

Let us denote the total amount of buffering in *G* by *B* which can be computed by the summation of the sizes of the buffers introduced on each individual arc. Thus we have:

$$B = \Sigma(u_j - u_i - w_{i,j})$$

$$= \sum_i u_i(\text{indegree}(i)-\text{outdegree}(i)) + \sum w_{i,j} \qquad (3.1)$$

Since $\sum w_{i,j}$ is a constant, minimizing B is equivalent to minimizing $\sum u_i(\text{indegree}(i)-\text{outdegree}(i))$, or to maximizing $\sum u_i(\text{outdegree}(i)-\text{indegree}(i))$. We also note that (outdegree(i)-indegree(i)) is a constant for each node i, determined solely by the structure of G. Hence (3.1) is a linear combination of u_i. Thus, optimal balancing of the graph G can have the following linear programming formulation:

LP1 Linear Program for Optimal Balancing of a Data Flow Graph

Maximize $\sum_i u_i(\text{outdegree}(i)-\text{indegree}(i))$.

Subject to

$u_i - u_j \le -w_{i,j}$ for all $(i,j) \subseteq E$ (1)
$u_n - u_1 = w_{s,t}$ (2)
u_i unrestricted (3)

In matrix notation, the above linear program can be expressed as

LP2 Linear Program **LP1** in Matrix Notation

Maximum **ub**

Subject to

$\mathbf{uA} \le \mathbf{c}$
u unrestricted

Here **ub** is the *objective function*, **b** is the *objective vector* of size n where the i-th component b_i = outdegree(i)-indegree(i); the *constraint matrix* A is the *incident matrix* of the original graph; and **c** is the *constraint vector* with its elements corresponding to the right

hand side in (1) and (2) of LP1.[1]

The linear programming formulation explicitly specifies all assumptions and constraints of the model, thus give a clear mathematical insight of the balancing problem. The optimal balancing of a graph is expressed and studied under one theoretical framework. There are many well-known solution techniques for linear programming problems. In the next section, we discuss efficient solutions to LP1 and LP2.

3.4 Solution of the Optimal Balancing Problems

3.4.1 An Example

Let us first study an example. We formulate the optimal balancing problem of the graph G in Figure 3.6 as follows:

Example 3.1

Maximize $2u_1 + u_2 + u_3 - u_4 - 3u_5$

Subject to

$$\begin{aligned} u_1 - u_2 &\le -1 \\ u_1 - u_5 &\le -20 \\ u_2 - u_4 &\le -3 \\ u_2 - u_3 &\le -4 \\ u_3 - u_4 &\le -4 \\ u_4 - u_5 &\le -1 \\ u_3 - u_5 &\le -5 \\ u_5 - u_1 &\le 20 \end{aligned}$$

---

1. Without loss of generality, constraint (2) can be replaced by $u_n - u_1 \le w_{s,t}$ for the sake of simplicity. We can think of this as adding an extra arc from node n back to node 1 with weight $w_{s,t}$, treating the auxiliary arc the same as other arcs in the graph.

u_1,u_2,u_3,u_4,u_5 unrestricted.

In matrix notation (LP2), the above linear programming model can be expressed as in

Maximize ub

$$uA \leq c$$

where

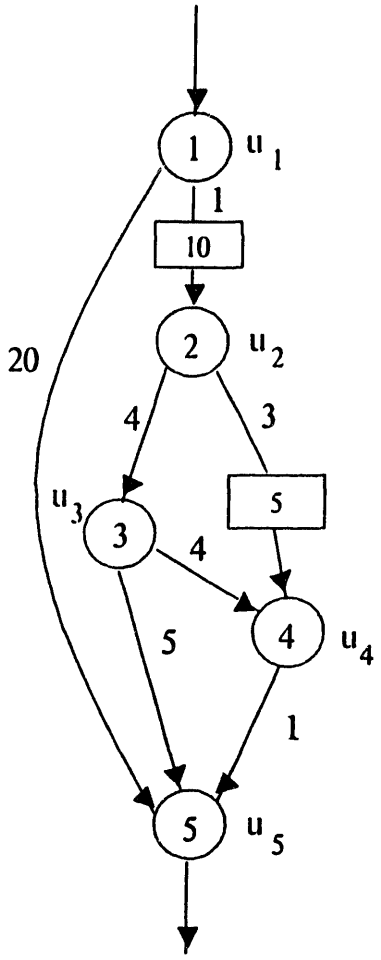
$u = (u1,u2,u3,u4,u5)$

$b = (2,1,1,-1,-3)$

$$A = \begin{pmatrix} 1 & 1 & & & & & -1 \\ -1 & & 1 & 1 & & & \\ & & & -1 & 1 & & 1 \\ & & -1 & & -1 & 1 & \\ & -1 & & & & -1 & -1 & 1 \end{pmatrix}$$

$c = (-1,-20,-3,-4,-4,-1,-5,20)$

u1, u2, u3, u4, u5 unrestricted



total buffer: 15

optimal solution:

u1 = 0, u2 = 11, u3 = 15, u4 = 19, u5 = 20

Figure 3.8. A Matrix Notation for the Example 3.1

Figure 3.8 (a).

An optimal solution to the problem is:

$$u_1 = 0$$
$$u_2 = 11$$
$$u_3 = 15$$
$$u_4 = 19$$
$$u_5 = 20$$

The optimal value of the object function is 53. Recall that our goal is to optimize the total amount of buffering B, where $B = \Sigma(u_j - u_i - \Sigma w_{i,j})$. Note that $\Sigma w_{i,j} = 38$, hence $B = 15$. The amount of buffering needed for each arc can be computed easily from the above solution and the result graph is shown in Figure 3.8 (b).

As we expected, all the variables in the above solution have integer values. We note that the constraint matrix A is *totally unimodular*, meaning that every subdeterminant of A is either $+1,-1$ or 0.[1] A linear programming problem with a totally unimodular coefficient matrix yields an optimal solution in integers for any objective vector and any integer constraint vector on the right-hand side of the constraints [71]. This guarantees the integrality of the optimal solutions for balancing problems.[2]

3.4.2 Solution Techniques

Thus, we can apply any solution technique for linear programming problems to optimal balancing problems to get optimal integer solutions. The most well-known technique for solving general linear programming problems is the *simplex method* [23].

---

1. Observe that the coefficient matrix A in LP2 corresponds to the incident matrix of the graph under consideration. Hence its unimodularity is straightforward.
2. Here we have applied without proof some important results from linear programming theory, such as the theory of solution integrality and its relation to unimodularity of constraint matrices. These results are well-known and discussed elsewhere, for example in [71].

There are a number of combinatorial optimization techniques based on simplex methods [71]. Other solution techniques are available, notably the polynomial Ellipsoid algorithm [58].

The structure of the balancing problems, however, is closely associated with a special class of linear programming problems, i.e., *network flow programming* problems [42]. In fact, the dual of the optimal balancing problem LP2 can be formulated as follows:

LP3 Linear Program Dual of LP2

Maximum $\mathbf{cx}$

Subject to

$$\mathbf{Ax} = \mathbf{b}$$
$$\mathbf{x} \geq 0$$

We immediately recognize that LP3 can be reduced to a class of well-known network flow programming problem — *min-cost flow problems* which have an efficient solution [71,57].

3.5 Extensions to the Results

3.5.1 Graphs with Multiple Input and Output Nodes

It is intuitively clear that any balancing technique for one-in-one-out graphs should also work for graphs with multiple inputs and outputs. Using the linear programming formulation, such an extension is straightforward.

Let us consider a graph $G = \{V,E\}$ with multiple input and output nodes. Assume G is under maximum loading conditions, i.e., all inputs and outputs are connected to perfect sources and sinks respectively. Let the set of input and output nodes be I and O respectively, and without loss of generality, we assume that each node in I has indegree one and each node in O has outdegree one. Let us introduce a dummy input node s such that all input arcs directed to each node in I emanate from s, and let us name this set of arcs E_s. Similarly, we introduce a dummy output node t such that all output arcs emanating from nodes in O are directed to t, and the corresponding set of arcs is called E_t. Let us study an augmented graph $G' = \{V',E'\}$ for G, where $V' = V \cup \{s,t\}$, $E' = E \cup E_s \cup E_t$. Assume that arcs in E_s and E_t have unit weights. It is easy to see that G' is an one-in-one-out graph Therefore, the optimal balancing technique developed in the last section can be applied directly to G, with the following extensions.

First extension is in the delay constraints between the input nodes and the output nodes, which are termed the *interface delay constraints.* For a one-in-one-out graph, the only interface delay constraint is that the total delay from the input node to the output node be unchanged by the optimal balancing procedure. Accordingly, this is called the *critical constraint.* (Recall that the last inequality in LP1 and LP2 reflects this constraint.) For a graph with multiple input and output nodes, there may be a set of interface delay constraints, one constraint for each pair of nodes in the set $I \times O$. Hence the total number of interface constraints is bounded by $|I||O|$. In practice, it may be that only certain interface delays are critical, i.e., the cost of the longest paths between corresponding pairs

of input and output nodes cannot be increased (by some "boundary conditions"). As long as the interface delay constraints are given, it is straightforward to add the set of corresponding linear inequalities to the constraint matrix and give a complete formulation of the linear program.

To establish the set of critical interface constraints, we need to compute the cost of the longest paths between each pair of nodes in $I \times O$. This can be solved based on the algorithm for finding the longest paths between all pairs of nodes in a graph. Since the graphs we are interested in are all acyclic, the polynomial time Floyd-Warshall algorithm for finding the shortest paths between each pair of nodes can easily be modified to compute

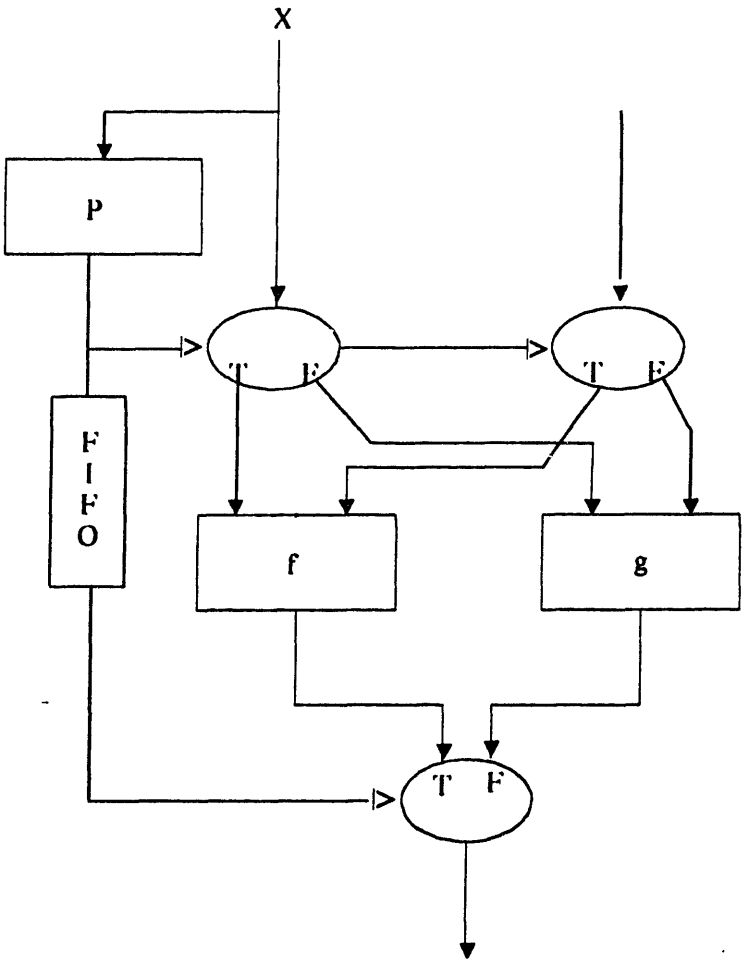


Figure 3.9. Balancing of conditional subgraph

the longest paths [8]. We simply replace the weight on each arc with its negative value and the computation of the shortest-path algorithm can be conducted on the transformed graph. Remember, however, that this is equivalent to finding longest paths, rather than shortest paths, in the original graph.

Assume that G' is balanced by an optimal balancing algorithm. Hence, all nodes can be executed with the maximum rate, including all nodes in I and O. The function of dummy nodes s and t can be removed and their function can be performed by a set of perfect sources and sinks (under the maximum loading condition). Therefore, the original graph G is also maximally pipelined.

3.5.2 Conditional Subgraphs

Another straightforward extension is to conditional subgraphs. Let use examine a conditional subgraph as shown in Figure 3.10 which computes the expression **if** p(x) **then** f(x) **else** g(x) **endif**. Assume the subgraphs p,f, and g are acyclic. As before, in order to balance the graph, each path from the input to the output should contain exact same number of actors. However, a token may take one of the two different paths, i.e. f or g, depending on the value it carries. Since the value is not known a priori, a strategy one may take is to always consider the worst case. As indicated in the graph, a FIFO is introduced in the arc between the output of P to the control input of merge actor to balance the graph, where the FIFO is chosen that it will balance the arc with the longer one of the two alternative data paths [32,43]. With such a strategy to extend our earlier work, we count each special actor in a conditional subgraph as an ordinary actor in formulating the linear program. Then the solution of the linear program will be a balanced graph. Obviously, no matter which path a token may travel in the balanced conditional subgraph, they all guarantee to have the same path length. Therefore, the graph can be maximally pipelined.

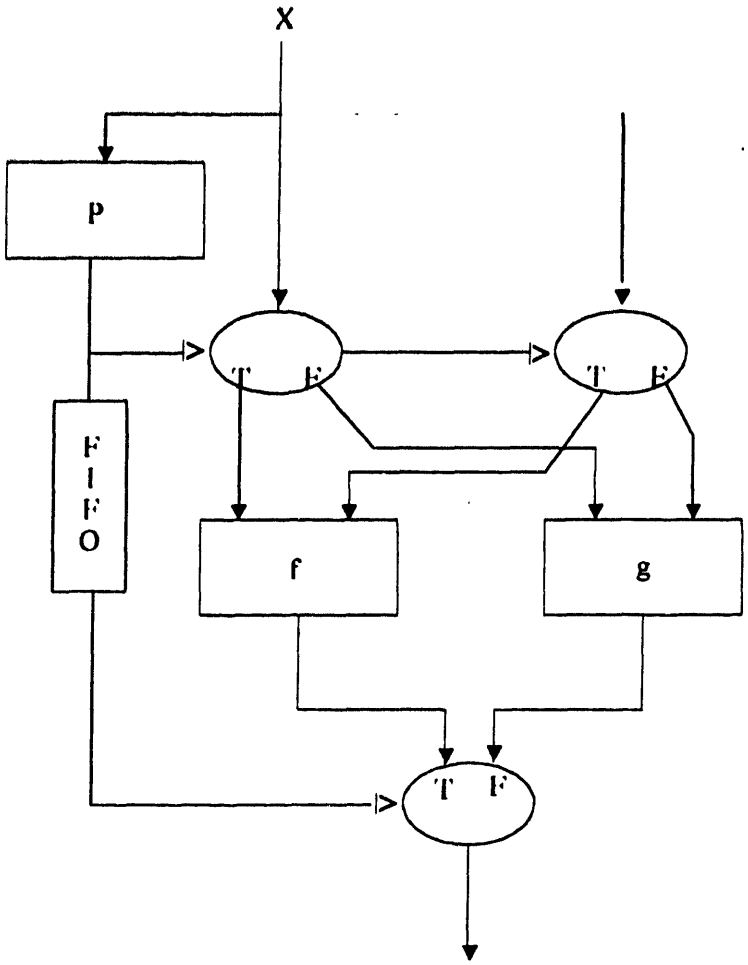


Figure 3.10. Balancing of conditional subgraph

4. The Structure and Notation for Source Programs

In this chapter, we describe briefly the structure of the source programs to be handled by the pipelined code mapping scheme developed in this thesis. An important feature of such programs is the regularity of array operations as outlined in Chapter 1. Such regularity is frequently found in scientific and numerical applications, and provides a good opportunity for a suitable computer architecture to efficiently exploit the parallelism in the programs.

The major portion of such a program usually consists of a collection of program blocks. Each block defines a new array from one or more input arrays. Figure 1.4 in Chapter 1 illustrates an example of a program which consists of five code blocks. All array operations are organized in these program blocks and take place in regular and repetitive patterns.

The communication between two program blocks in terms of an array can be viewed as a producer-consumer pair. One block, the producer, generates the elements of the array, while the other block, the consumer, uses them to produce results which may become the elements of another array. The mapping strategy for such programs on a data flow machine is particularly attractive when the machine code of the producer and the consumer can be executed concurrently, in a pipelined fashion. In an ideal case, the data flow machine program for both blocks can run in a maximally pipelined fashion and the communication between them can be implemented directly as a simple arc in the data flow graphs for carrying the element values of the array, avoiding the use of memory for the entire array. Thus, the corresponding array operations can be removed from the machine code and the overhead of memory operations effectively avoided. Often a certain delay is needed to balance the computation between various code blocks, and it may be necessary to implement the communication link through a FIFO. Even so there will still be less overhead because implementation of FIFO is expected to be more efficient in the target data flow machine than ordinary memory operations.

Sometimes, the data flow machine programs for both blocks cannot run concurrently. For example, the order of the array values generated by the producer block may be different from that needed by the consumer block. In such a situation, although the producer and consumer blocks may each run in a pipelined fashion, their execution may not be able to overlap, and the communication link should be implemented through some form of storage.

Our objective is to design a suitable code mapping scheme for such programs. For the purpose of clarity and simplicity, we introduce PIPVAL — a small subset of Val with slight extensions -- as the source language. It is particularly suitable to represent the program blocks to be handled by the mapping scheme. In the Sections 4.1 and 4.2 we outline PIPVAL and its main language features in terms of array operations. In Section 4.3, we describe several types of code blocks that are of most interest to us.

4.1 The PIPVAL Language

4.1.1 An overview

In this section we briefly introduce the language PIPVAL, which is basically a subset of the programming language Val [4] with slight extensions. As a subset, the language inherits most of the Val syntax notation and semantic conventions. Most important, PIPVAL is an applicative (or value-oriented) language. As in Val, each basic syntactic unit, called an expression, corresponds to a function whose evaluation produces a set of values. The language is free of *side-effects*, i.e. the evaluation of an expression does not interfere with the evaluation of other expressions.

A major feature of PIPVAL is the way in which array operation constructs are provided. In choosing the set of array operation constructs for the language subset, we hope both to simplify the task of expressing the class of Val programs where array manipulation has strong regularity and to facilitate the formulation of the basic code

mapping scheme for such programs. Some extension is made in this direction — mainly the introduction of the **for-construct** expression.

Now we discuss the basic syntax of PIPVAL. Again, our major concern is not to provide powerful language features to allow flexible programming (for that purpose, readers should refer to the full Val language). The language is intended to provide a reasonably simple source language to express a class of Val programs to be handled by the pipelined code mapping schemes.

The syntax of the language is given in Figure 4.1. Here, we eliminate certain syntactic sugaring of Val to keep the syntax simple. Type information is not explicitly included in the syntax. However, we assume the values expressed in a PIPVAL program have the correct types, which are a subset of types defined in the full language Val. That is, PIPVAL provides values of ordinary scalar data types (such as integers, reals, booleans and characters) and a structured data type, i.e., array. We also assume that the PIPVAL programs handled by our mapping schemes are type correct in the sense described in [4].

The basic syntactic unit of PIPVAL is an expression. There are four major major types of expressions: primitive expressions, conditional expressions, **let-in** expressions, **for-iter** expressions, **forall** expressions and **for-construct** expressions. An expression can have arity greater than one, as in the form of exp,exp.

In the rest of this section we briefly outline the first five types of expressions and related terminologies. They need little explanation since they are equivalent to those in Val, or any similar constructs found in other applicative languages. Readers who are familiar with Val may wish to proceed to the next section where array selection operations and array construction expressions (mainly **forall** and **for-construct** expressions) are described.

A PIPVAL primitive expression is either a constant, a *value name* (a term inherited from Val), or an expression constructed by the primitive operators in the form

$$\mathbf{op}_1\ \mathbf{exp}$$

```
exp ::=     primitive-exp
          | exp,exp
          | let-in-exp
          | conditional-exp
          | forall-exp
          | for-construct-exp
          | for-iter-exp

primitive-exp ::= const | id | op exp | exp op exp | id[exp]

let-in-exp ::= let idlist = exp in exp endlet

conditional-exp ::=
          if exp then exp
          {elseif exp then exp}
          else exp
          endif

for-iter-exp ::= for idlist = exp do iterbody endfor

iterbody ::= exp | iter idlist = exp enditer

forall-exp ::=
          forall id in [exp]
          construct exp
          endall

for-construct-exp ::=
          for
               id from exp to exp
               idlist from exp
          construct exp
          endfor

idlist ::= id {,id}
```

Figure 4.1. The Syntax of PIPVAL

or

$$exp_1\ op_2\ exp_2$$

where exp, exp_1, exp_2 are expressions of scalar data types, and op_1, op_2 belong to the set of unary and binary operations respectively. These operators are defined on the appropriate data types and include the useful scalar operations found in Val. The set of primitive expressions also includes an expression A[i] that denotes array selection operation. It is discussed in the next section.

A **let-in** expression is of the form

let $x_1, x_2,...x_k = E_1, E_2,...E_k$ **in** E **endlet**

A **let-in** expression is used to introduce new value names such as x_1 to x_k and define their values by expression E_1 through E_k respectively. The body E is evaluated in the scope making use of the values defined for $x_1...x_k$. Note that each value name may be defined only once. We can also use an alternative notation for **let-in** expression, as shown below:

```
let
    x1 = E1,
    x2 = E2,
      :
      .
    xk = Ek
in
    E
endlet
```

Such syntactic sugaring is useful when . e definition expressions $E_1...E_k$ are complicated. It can also be applied to other expressions introduced below where a definition idlist = exp is allowed.

The PIPVAL nesting rule of scopes is essentially the same as Val. The scope of each value name introduced in a **let** block is the scope of E less any inner constructs that reintroduce the same names. As in Val, a value name in E, other than $x_1...x_n$, is called a *free value name* of the **let-in** expression, unless it is in the definition part of any inner **let-in** expression. The only difference is that in PIPVAL, value names $x_1...x_k$ cannot be used in $E_1...E_k$.[1]

A conditional expression is of the form

if B_0 **then** E_0
elseif B_1 **then** E_1
.
.
elseif B_{k-1} **then** E_{k-1}
else E_k
endif

The expressions B_0, B_1..., B_{k-1} following the keywords **if** and **elseifs** are test expressions which must have arity one and be of type boolean. The expressions following **then** and **else** are called *arms* and should *conform* to each other, i.e., they must have the same arity and type. When an expression has more than two arms, it is called a *multi-armed* conditional expression. The above example is an expression with k arms.

The value of a conditional expression will be the value of one of its arms, depending on the values of the test expressions. Let B_i be the first test expression in the sequence $B_0...B_{k-1}$ that evaluates to **true**. The corresponding arm E_i is said to be *selected.* Otherwise, if the values of all B_is (i = 1...k-1) are FALSE, the last arm E_k is selected. The value of the

---

1. In Val, the **let-in** allows a sequence of definitions. Our simplification, however, does not limit the expressive power of the language by noting that a sequence of definition in Val can also be replaced by a nested **let-in** expression. It helps to simplify the presentation of the mapping rule for **let-in** expression.

selected arm is the result of the expression.

Finally, let us describe the PIPVAL **for-iter** expression briefly by the example shown below.

```
for
    f = 1,                                  % initialization
    i = n
do
    if i = 0 then f                         % iterbody
    else
        iter
            f = f*i,
            i = i-1
        enditer
    endif
endfor
```

This expression computes the factorial of n. Two value names f,i — called *loop names* — are introduced and defined in the initialization part. The iterbody part is evaluated using the current definition of the loop names, and the result is either to terminate the iteration, with the value f returned, or to iterate again with the new definition of the loop names. In this thesis, we are particularly interested in two iteration constructs for constructing arrays. These are introduced in the next section.

The set of expressions just described and any expressions constructed from these constructs are called *simple expressions.* The set of simple expressions which do not contain any iterative expression as subexpression is called the set of *simple primitive expressions.* Simple (primitive) expressions are important building blocks for the more complex expressions to be discussed later.

4.2 Array Operation Constructs

An important aspect of PIPVAL is the way in which array operations are expressed. Remember that the purpose of PIPVAL is to represent the main feature of a class of VAL programs in which array operations are organized in a regular pattern. The set of array operation constructs provided in PIPVAL should effectively meet this goal.

A PIPVAL array is similar to a Val array. Arrays and their operations are applicative. An array is nothing but a value. An array value consists of : (1) a range (LO, HI) where LO, HI are integers and LO $\leq$ HI + 1; (2) a sequence of HI - LO + 1 elements of the same type. We should note the distinction between the concept of an applicative array and the concept of an array in conventional languages. In conventional languages an array is a place in store in which values may be stored in sequence.

Array indices should be considered as a mechanism to provide value names for the array elements. An element of an array can be accessed by an array selection operation construct which has the form A[E]. Here A is an array to be accessed and E is an expression which specifies an index value (say i) within the index range of A. Then the evaluation of the expression A[E] returns the value of the i-th element of A.

As in Val, the syntax provides abbreviated forms of the selection operations for multi-dimensional arrays. Multi-dimensional arrays are regarded as arrays of arrays. Hence, if A is a two-dimensional array, a straightforward way to write an expression to select an element is A[i][j]. PIPVAL allows the use of A[i,j], as a form with syntactic sugaring added to A[i][j]. We will discuss multi-dimensional arrays in more detail in later chapters.

The means of expressing array creation operations distinguishes PIPVAL from Val. Like most functional languages, Val allows arrays to be generated by append operations [4]. Since arrays are treated as if they were values, an array append operation such as A[i:v] in Val conceptually means the creation of a new array which is identical to A except that the i-th element is replaced by v. The excessive overhead due to the copying of the entire

array makes the append operation very inefficient in terms of its implementation [2,3]. In designing PIPVAL, we have concentrated on a class of programs where the unrestricted use of append operations is disallowed. In fact, the append operation is not even included in the language subset. Instead, suitable array creation constructs are provided which allow arrays to be generated in a regular fashion.

As outlined at the beginning of this chapter, the programs to be handled by our mapping schemes are organized as a collection of code blocks where each block is essentially one of the PIPVAL array construction expressions. Such an expression consists of either a **forall** or a **for-construct** expression. Although both constructs can be considered as special cases of the **for-iter** construct, the unique features of **forall** and **for-construct** make them particularly useful in expressing programs in which arrays are constructed and accessed in a regular fashion.

4.2.1 Forall Expressions

A **forall** expression can be used to express the construction of an array where all elements of the array can be computed independently. The following is an example of a **forall** expression which defines a one-dimensional array X in terms of array A.

```
X =
        forall i in [LO,HI]
        construct
                if i = LO then A[LO]
                elseif i = HI then A[HI]
                else
                (A[i-1]+A[i]+A[i+1])/3
                endif
        endall
```

The **forall** construct may introduce one index value, such as i in the above example,

and define its range. The body expression of the **forall** construct is evaluated for each index value in the range, and an array is constructed with the index range so defined. Since the result array value can be considered as constructed from an empty array, and the element value for each index is only computed once, we do not need to use any explicit append operations in the expression.

The main feature of a **forall** expression is that the array elements can be evaluated in parallel because there are no data dependencies between them. Typically the body of a **forall** expression is a conditional expression which partitions the index range of the result array into mutually exclusive and collectively exhaustive index subranges, each corresponding to an arm of the conditional expression. Such a conditional expression is called an *range-partitioning* conditional expression. In the above example, there are three index subranges, i.e. [LO,LO],[HI,HI] and [LO+1, HI-1].

```
X =
    forall i in [0,m+1]
    construct
        if i = 0 then A[i]
        elseif i = m+1 then A[i]
        else
            forall j in [0,n+1]
            construct
                if j = 0 then A[i,j]
                if j = n+1 then A[i,j]
                else
                    (A[i,j-1] + A[i,j+1]
                     + A[i-1,j] + A[i+1,j])/4
                endif
            endall
        endif
    endall
```

Figure 4.2. An example of a two-level forall expression

The **forall** expression in the above example constructs a one-dimensional array X, where each element is computed by a simple expression which computes a scalar value. We call such an expression an *one-level* **forall** expression. The **forall** constructs can also be nested to compute a multi-dimensional array, thus forming a *multi-level* **forall** expression. A *k-level* **forall** expression (k>1) constructs a k-dimensional array, where its elements are constructed by either (k-1)-level **forall** expressions, or simple expressions.

For example, Figure 4.2 shows a two-level **forall** expression which constructs a two-dimensional array X.[1] The result array A can be viewed as a one-dimensional array constructed by the outer **forall** expression. This expression, called a *level-1* **forall** expression, has an index range of [0, m+1]. The elements of this one-dimensional array are also one-dimensional arrays with an index range of [0, n+1]. The majority of these arrays are constructed by the inner one-level **forall** expression which is called a level-2 **forall** expression in this case. The two expressions that compute the two boundary arrays (i.e. i=0, i=m+1 respectively) are not **forall** expressions. But they are simple expressions, i.e., expressions made of array selection operations.

In this way, the nesting levels of a **forall** expression are paired naturally with the dimensions of the result array being constructed. This construction defines multi-dimensional arrays as arrays of arrays.[2] The notion of nesting levels will also be used in the nested **for-construct** expressions and other nested expressions introduced later.

---

1. This **forall** expression is the core of the Possion solver program known as the (two-dimensional) *model problem* [41].

2. Other different views of multi-dimensional arrays and their impacts on the representation and implementation are discussed in later chapters.

4.2.2 For-construct Expressions

When data dependencies exist between array elements, some form of iterative construct is usually needed to express the corresponding array creation operation. In Val the **for-iter** construct and the array append operations are used to perform this function. In PIPVAL, we introduce the **for-construct** expression to express array creation operations with certain regularity which allows one to avoid the use of the append operation.

A typical Val **for-iter** expression with such regularity is shown in Figure 4.3.[1] The initialization part defines an internal array name and an index value name, corresponding to T and i in our example. The internal array T is initialized to an empty array denoted by a constant **array-empty**.[2] The evaluation of the expression is conducted iteratively,

```
X =
        for
                i = 0,
                x = B[1],
                T = array_empty
        do
                if i > n then T
                else
                        iter
                                T = T[I:x],
                                i = i+1,
                                x = A[i]*T[i-1] + B[i]
                        enditer
                endif
        endfor
```

Figure 4.3. An Example of a Val for-iter construct for creating an array

---

1. This **for-iter** expression specifies a first-order linear recurrence.
2. PIPVAL inherits this constant array from Val and reader is referred to [4] for a discussion of its meaning.

controlled by the simple test expression i > n. If i ≤ n, the iteration is continued and T and i are redefined as specified in the **iter** arm of the body. When the test expression returns false, (i.e. i > n), the evaluation of the **for-iter** expression is completed by returning the array T as the result array X.

One important feature of the above **for-iter** expression is that the array is built from **array_empty** by a series of append operations, one for each index i in a specific range (i.e. [1...n] in our example). Such an expression is characterized by having a loop name (e.g., i in the above example) be a counter, and having the iteration termination predicate be a simple range limit test of that counter (e.g., i > n in the example). The importance of such **for-iter** array construction expressions is also observed in [4]. The iteration is advanced by incrementing (or decrementing) the counter by one. In this thesis, we are interested mostly in the case where the number of iterations is known in advance through the evaluation of the index limit expression (e.g., n) or at least is known before the iterations are started. This means that we are mostly interested in the mapping of arrays which have compile-time computable bounds.

In PIPVAL, the **for-construct** expression is introduced to express the feature of such regular array construction operations. The following shows the **for-construct** expression which is equivalent to the **for-iter** expression in Figure 4.3.

```
X =
        for     i from 1 to n
                T from array_empty
        construct
                if i = 1 then B[i]
                else
                        A[i]*T[i-1] + B[i]
                endif
        endfor
```

Between **for** and **construct**, the index value name and the internal name (T) for the

result array are introduced and specified. The mechanism **from-to** specifies not only the index range but also the order of the indices to be generated. The introduction of the internal array name T is important because it may be used extensively inside the body expression — the expression after **construct**. For the purpose of this thesis, the constant **array_empty** is always assumed to be the initial value of the internal array. As with the **forall** construct, **for-construct** provides a mean to express array creation operations without using the append construct.

The body of a typical **for-construct** expression also has a range-partitioning conditional expression as its top-level structure. In the above example, it partitions the index range into two subranges: [1:1], [2,n]. The **for-construct** expressions can be nested to construct a multi-dimensional array. Following the same rule for constructing a multi-level **forall** expression, we can construct a *multi-level* **for-construct** expression. Similarly, the nesting levels of a nested **for-construct** expression correspond to the dimensions of the array it generates.

4.3 Code Blocks with Simple Nested Structure

Code blocks in a program often have nested structures. In this thesis, we are particularly interested in the following three classes of code blocks which are frequently found in the computation intensive part of source programs.

4.3.1 Class-1: Primitive **forall** expressions

A one-level **forall** expression is *primitive* if its element is computed by simple primitive expressions. A k-level (k>1) **forall** expression is primitive if its elements are constructed either by (k-1) level- primitive **forall** expressions, or simple primitive expressions.

For example, the code of the model problem in Figure 4.2 is a two-level primitive **forall** expression. Its body is a range-partitioning conditional expression which partitions

the array elements into two boundary rows for i = 0, i = m+1 and the internal rows correspond to index range [1,m]. The boundary rows are specified by simple expressions, while the internal rows are specified by a one-level primitive **forall** expression.

4.3.2 Class-2: Primitive **for-construct** blocks

A one-level **for-construct** expression is *primitive* if its element is computed by simple primitive expressions A k-level (k>1) **for-construct** expression is primitive if its elements are constructed either by (k-1) level primitive **for-construct** expressions, or simple primitive expressions.

In Figure 4.4, we show a two-level primitive **for-construct** expression which takes an input array U and constructs a two-dimensional array UT. The index range of i is divided

```
UT =
        for   i from 0 to m+1
              T1 from array_empty
        construct
              if i = 0 then U[i]
              elseif i = m+1 then U[i]
              else
                    for   j from 0 to n+1
                          T2 from array_empty
                    construct
                          if j = 0 then U[i,j]
                          else j = n+1 then U[i,j]
                          else
                                (U[i+1,j] + U[i,j+1]
                                 + T1[i-1,j] + T2[i,j+1])*1/4
                          endif
                    endfor
              endif
        endfor
```

Figure 4.4. A two-level primitive for-construct expression

into three subranges: the two boundaries and the subrange [1,m]. A level-2 **for-construct** expression computes the elements of UT (one-dimensional arrays) as the internal rows. Note how the arrays T1 and T2 are used in the body.

4.3.3 Class 3: Multilevel expression with innermost level primitive **forall** or **for-construct** expressions

The **forall** and **for-construct** constructs can be nested in an arbitrary fashion to form a *multi-level mixed* expression to compute a multi-dimensional array. For example, a two-level expression may consist of a **forall** construct to form its level-1 expression, but may contain **for-construct** expressions as its level-2 expressions as shown in Figure 4.5. Another situation is shown in Figure 4.6, where the the level-1 expression of the two-level expression consists of a **for-construct** expression and its body contains a **forall** expression.

```
X =
    forall i in [0,m+1]
    construct
            if i = 0 then B[i]
            elseif i = m+1 then B[i]
            else
                    for     j from 0 to n+1
                            T from array_empty
                    construct
                            if j = 0 then B[i,j]
                            else j = n+1 then B[i,j]
                            else
                             A[i,j]*T[j-1] + B[i,j]
                            endif
                    endfor
            endif
    endall
```

Figure 4.5. A two-level mixed code block -- example 1

```
X =
        for   i from 0 to m + 1
              T from array_empty
        construct
              if i = 0 then B[i]
              else
                    forall j in [0, n + 1]
                    construct
                          A[i,j]*T[i-1,j] + B[i,j]
                    forall
              endif
        endfor
```

Figure 4.6. A two-level mixed code block -- example 2

It may be more complicated for situations with many nesting levels. Recall that for a nested loop in a conventional language such as Fortran, the dominant factor for the overall performance of the implementation is the mapping of the innermost loop. We anticipate that the same will be true in mapping a nested expression on data flow computers. Hence, the nested mixed expressions to be studied in this thesis are partitioned according to the structure of their innermost expression. In particular, we are interested in the situation where the innermost level expressions consists of (1) a primitive **forall** expression; or (2) a primitive **for-construct** expression.

A slight extension of the above cases occurs when the innermost level expression allows its body to contain a **for-iter** expression which computes scalar values.

5. An Overview of the Basic Pipelined Code Mapping Schemes

The rest of this thesis will investigate pipelined code mapping schemes that can match the regularity of array operations in the PIPVAL representation of the source program with the power of the target data flow machines to exploit parallelism of data flow graphs.

The basic pipelined code mapping scheme (Chapters 6-9) concentrates on the analysis and handling of the class-1 and class-2 PIPVAL code blocks (hence also the core of class-3 code blocks) outlined in the last chapter. It is also the basis upon which a number of other related transformation techniques can be used (chapters 10). In this chapter we give a brief general outline.

The basic code mapping scheme is essentially a two-step process. The first step consists of the application of a set of basic mapping rules which can translate the code blocks into pipelined data flow graphs. These graphs are described in a static data flow graph language (SDFGL) to be introduced in the last section of this chapter. In this step, conceptual arrays in the source program — i.e. the input and output arrays as seen by each code block — remain unchanged, but the array operations are translated into corresponding data flow actors in the result graph. The links between code blocks are now represented by data flow arcs carrying tokens with array values.

The second step consists of the application of a set of optimization procedures which can remove the array actors from the result graphs of step 1 and replace them with ordinary graph actors. Thus, the links between code blocks become ordinary arcs of a data flow graph. The result graph for a pair of producer and consumer code blocks may be executed concurrently, both in a pipelined fashion, without involving array operations.

In presenting the basic mapping schemes, our efforts are devoted both to the development of the mapping algorithms for the code blocks, and the formulation of the set of conditions under which they can be applied. These conditions are derived by analyzing the structure of each type of the code block, especially the pattern of the array operations

involved in its computation. Therefore, these conditions are certainly important in terms of mapping each individual code block. Moreover, the information provided by the collection of these conditions becomes very valuable for some global analysis necessary in making critical mapping strategy decisions (see Chapter 11).

5.1 Data flow representation of arrays

To develop the mapping scheme for both step 1 and step 2, it is important to choose appropriate representations for the arrays in a data flow graph. Let us use a one-dimensional array A of integers as an example, where A has 1,m as its low and high

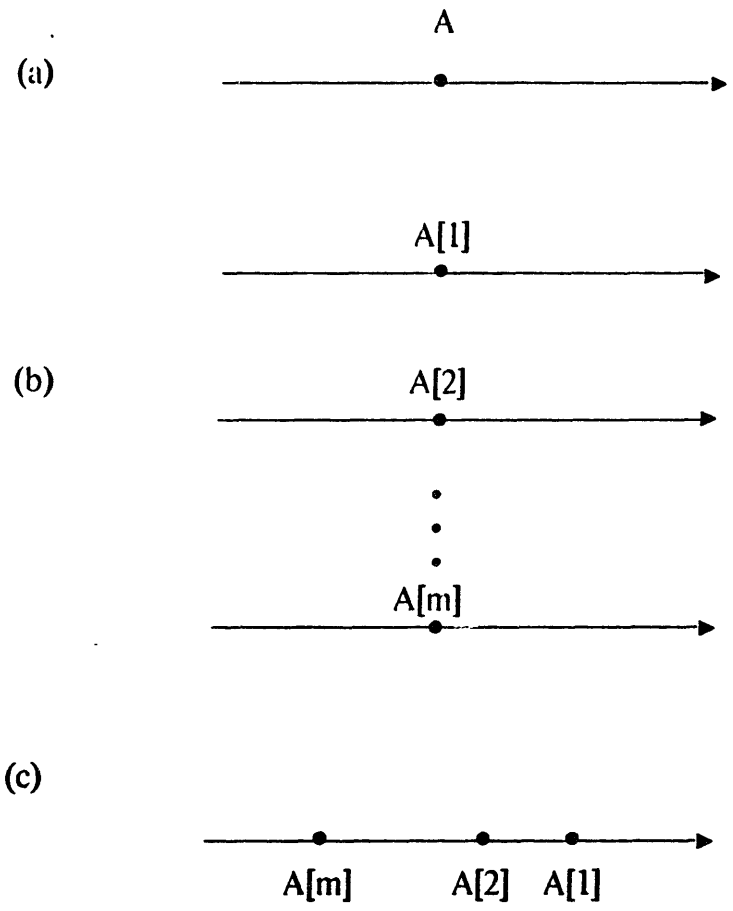


Figure 5.1. Data flow representations of a one-dimensional array

index limits respectively. Figure 5.1 shows the possible data flow representations of A. In Figure 5.1 (a), array A has an *unflattened* representation, i.e. array A is represented as an array value carried by one token on a data flow arc. An array can also take *flattened* representations as described below. In Figure 5.1 (b), array A is represented by a set of element values conveyed at some moment by tokens on a certain group of data flow arcs, one for each index value. In Figure 5.1 (c), array A is represented as a sequence of element values carried by tokens on a single arc at successive moments.

The unflattened representation in Figure 5.1 (a) is used in developing the basic mapping rules for array operations because it is conceptually close to the model of arrays in the source language. Consequently, the mapping rules can be presented in a general and simple fashion. As will be shown later, this representation is particularly helpful in formulating mapping rules for multi-dimensional arrays recursively from those of one-dimensional arrays. In this thesis, we do not study the detail of the format of an array token (e.g., array descriptor values, array memory addressing convention, etc.) but merely assume that it carries all the information needed for the corresponding graph actors to perform the necessary operations. A brief discussion on the efficient implementation of such array operations in the target machine is included in Chapter 11.

In contrast to Figure 5.1 (a), the two flattened representations in Figure 5.1 (b) and (c) directly represent values of the array elements. Thus, they both provide a basis for eliminating the overhead caused by manipulation of array values. Accordingly, graph actors for array operations can be replaced with ordinary actors. The difference between Figure 5.1 (b) and (c) reveals the basic space/time tradeoff in structuring machine code for efficient operations on a data flow computer. The pipelined code mapping scheme in this thesis uses all these representations in different aspects of the translation process.

In a flattened representation as in Figure 5.1 (c), the order of the element values in the sequence is an important part of the representation. For a one-dimensional array, there are two sensible orders of the representation as described in Figure 5.2 (a) and (b)

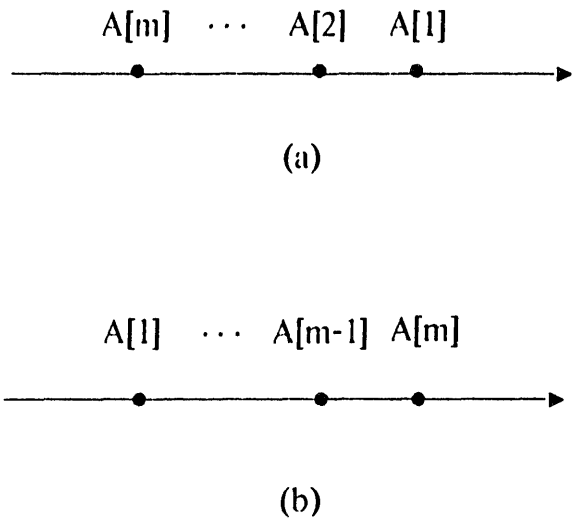


Figure 5.2. Two major orders for a flattened representation of a one-dimensional array

which are called the *major normal order* and *major reverse order* respectively.

So far our discussion has centered on one-dimensional arrays. The same principle can also be used for representing multi-dimensional arrays, although complexities arise when we elaborate the concepts of an array value and the order of its flattened representation. We leave the detail of such extension until Chapter 8, where the mapping scheme for multi-dimensional arrays is discussed in detail.

5.2 Basic Mapping Schemes

The basic mapping rules are presented as a set of recursive algorithms. These mapping rules together define a mapping M which translates PIPVAL expressions in each syntactic category into data flow graphs. In the next section, a static data flow graph language SDFGL is introduced which will be used to specify object data flow graphs generated by such mapping rules and the conditions under which they can be applied. In the result graph, there may exist explicit array actors, which are supported by array

operations in target machines (A brief discussion of their implementation is in Chapter 11).

As in most earlier work on the translation techniques for Val programs [6,79], our mapping rules are based on the framework of Brock's translation algorithms [19,20]. Therefore the data flow graphs generated by our mapping scheme are a correct semantic representation of the source program according to the semantic model developed in [19,20]. It is beyond the scope of this thesis to describe Brock's algorithms and his formal semantic model, and interested readers are referred to the above references. Instead, in the presentation of our basic mapping rules, we frequently indicate how it is related to Brock's algorithms.

The data flow graphs generated by the basic mapping rules may contain array actors. The direct architectural support of the actors may be expensive in data flow computers, especially when array descriptor values need to be manipulated [2,78]. For example, assume an array A is generated by code block C1 and is used by code block C2. In the corresponding data flow graphs, C1 may have array actors to "pack" the element values into array A, and C2 may have array actors to "unpack" the array so that its elements can be used. Storing an array in some form of RAM memory provides both the buffering between C1 and C2, and the mechanism to support random access so that the orders of "packing" and "unpacking" do not matter. However, if the two orders match each other, we do not need to pack or unpack the elements through memory. The array actors can be directly implemented through ordinary data flow actors, and the links between the two code blocks become regular data flow arcs, perhaps attached with certain FIFO buffers. The goal of the optimization procedures is to perform such transformation.

Although the optimization procedures are applied directly to data flow graphs, certain parameters used in the process are related to the attributes of the original code block. Therefore, optimization procedures are presented for different situations of the code block structure, and the conditions for each situation are outlined. These will contribute to the construction of the set of useful attributes associated with code blocks

useful for mapping strategy decisions. They together also characterize the set of PIPVAL code blocks which can be effectively optimized.

5.3 SDFGL — A Static Data Flow Graph Language

5.3.1 The Basic Language

We now introduce a static data flow graph language (SDFGL) as a textual description language for static data flow graphs. SDFGL is not a complete data flow programming language. It contains only those features which provide a convenient tool for the specification of the result graph generated by the basic mapping rules. Using SDFGL, a graph for a PIPVAL expression can be constructed from the graph of its subexpressions by recursive application of the mapping rules. This language is based on the graph assembly language in [19].

A SDFGL graph has two sets of labeled ports: *input ports* and *output ports* used for input/output connections. Internally, it contains a set of actors and links. A node in a SDFGL graph denotes an actor in the corresponding data flow graph. It also has two sets of ports, i.e., input ports and output ports. As a convention, the input and output ports of an actor are usually labeled by consecutive non-negative integers, unless otherwise specified. SDFGL provides the two functions IN, OUT to get the two sets of ports. For example, for an addition actor with 2 input ports 1,2, and an output port 1, we have IN (+) = {1,2}, OUT (+) = {1}. The cardinality of a set L of ports is denoted by #(L). For example, #(IN(+)) = 2, #(OUT(+)) = 1. Similarly, IN and OUT can also be applied to graphs for the same purpose. Providing graphs and actors with the same mechanism for their input/output specification facilitates the recursive graph definition.

The set of links is used to interconnect ports of actors or graphs. A link can be considered as a copy actor (or an identity actor), which has one input port and multiple output ports. Since there is no need to distinguish the ports, each link can be conveniently

denoted by a unique label.

As a syntactic convention, a graph is described by the following four basic parts:

input ports: <input-ports>
output ports: <output-ports>
links: <links>
components: <components>

In the above representation, <input-ports> and <output-ports> are the the sets of graph input and output ports; <links> is the set of all links of the graph; <components> is the set of all actors or named subgraphs (explained shortly) in the graph, as well as assignments specifying the direct connections between the input and output ports of the graph.

Each actor is specified by its operator name OP (e.g. +,-,*,/, etc.) followed by two sets:

OP **Inputs** : <input-assignments>
outputs : <output-assignments>

Each member of the set <input-assignments> specifies the assignment of an input port a of the actor, written as:

$\alpha \rightarrow a$

Similarly, each member of the set <output-assignments> specifies the assignment of an output port a written as:

$a \rightarrow \alpha$

Here, the arrow $\rightarrow$ always points in the direction of data flow, and α denotes a graph

input/output port or a link. Interconnections between actor ports are indicated by being assigned to the same link.

A subgraph can be named by an operation label and two lists for its input and output ports. Thus, it can be specified in the same way as an actor.

5.3.2 An Example of a SDFGL graph representation

In Figure 5.3 (b), a SDFGL graph of the data flow graph in Figure 5.3 (a) is given. The corresponding SDFGL description, shown in Figure 5.4, is self-contained. The input port of the graph, labeled *trigger*, is assigned to each constant actor in the graph. For

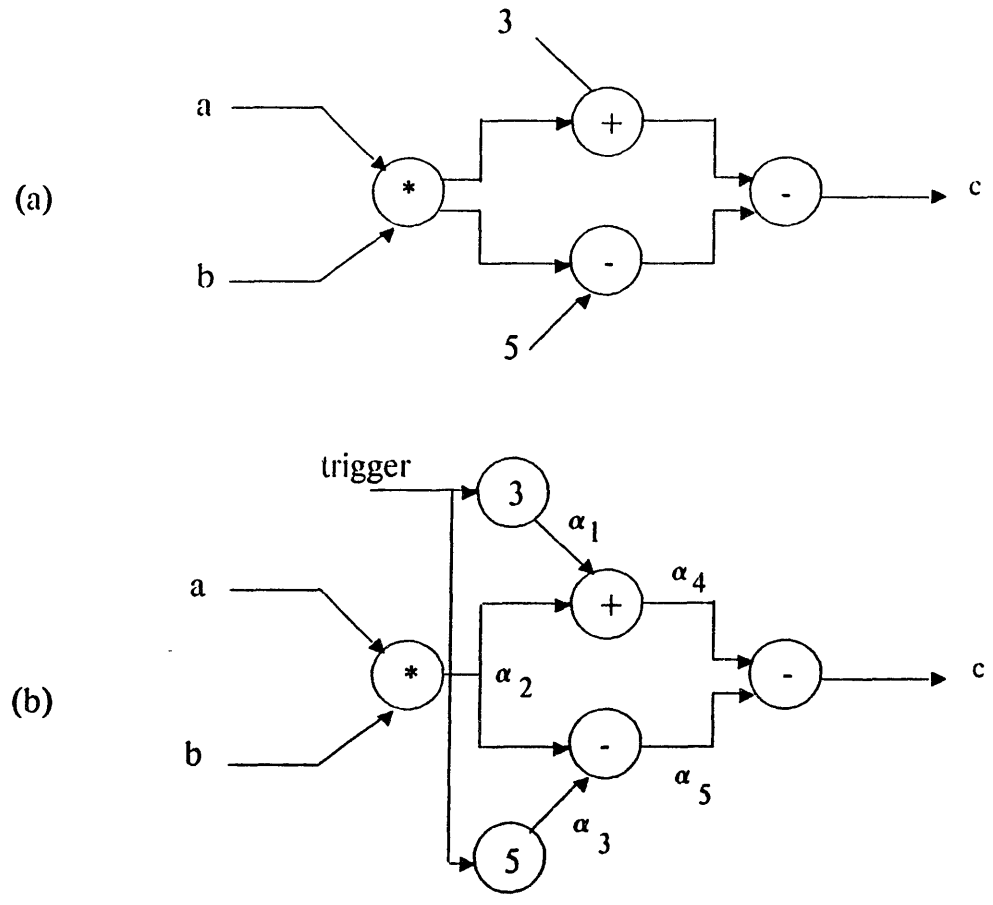


Figure 5.3. An example of SDFGL -- part 1

input ports: trigger, a, b

output ports: c

links: $i \in \{1...5\} \alpha_i$

components:

3 inputs: trigger → trigger
outputs: 1 → α_1

* inputs: a → 1, b → 2
outputs: 1 → α_2

5 inputs: trigger → trigger
outputs: 1 → α_3

\+ inputs: α_1 → 1, α_2 → 2
outputs: 1 → α_4

\- inputs: α_2 → 1, α_3 → 2
outputs: 1 → α_5

/ inputs: α_4 → 1, α_5 → 2
outputs: 1 → c

Figure 5.4. An example of SDFGL -- part 2

simplicity, the constant is usually written directly in the actor as the OP part of a constant actor.

5.3.3 Extension of the basic SDFGL

1. Definitions, Conditions and Remarks

In this thesis, we use an extended version of SDFGL. We add three more components to a SDFGL graph to get

def:	<definitions>
conditions	<conditions>
remarks	<remarks>
input ports:	<input-ports>
output ports:	<output-ports>
links:	<links>
components:	<components>

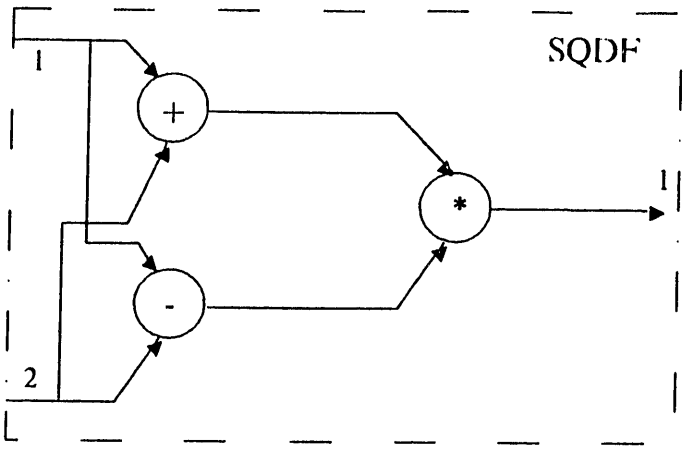
The <definitions> part is used to introduce a set of temporary names for a list of ports, a subgraph, etc. This component is used to simplify the graph presentation. The <conditions> part is used to specify the list of conditions or restrictions under which the graph construction is appropriate. The <conditions> part in the graph can also be used to formulate the set of attributes from which mapping strategy can be determined. The <remarks> part is reserved for comments.

Finally, any or all of <definition>, <condition> and <remark> parts need not be present in a SDFGL graph.

2. Named SDFGL Subgraphs

For convenience, a SDFGL graph may be given a name, known as a *named subgraph*, to be used as a component to construct other SDFGL graphs. For example, the SDFGL graph in Figure 5.5 (a) computes the difference of the squares of its two inputs. We can turn it into a named subgraph SQDF and use it elsewhere as shown in Figure 5.5 (b), where two copies of SQDF are used.

(a)



(b)

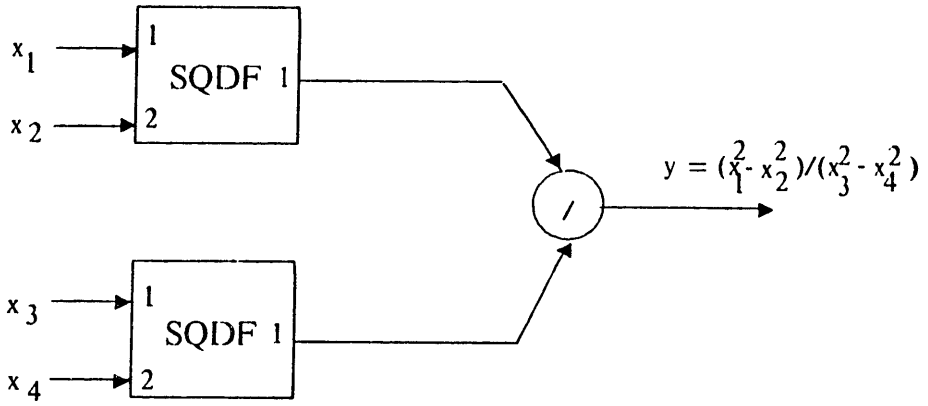


Figure 5.5. An example of using named subgraphs in SDFGL

3. Range Constructor

It is often necessary to construct a graph over a range of items or sets of input/output port labels. For this purpose we use

$(a \in \mathcal{A})$*item*

to specify a set which, for every $b \in \mathcal{A}$ ($\mathcal{A}$ is a set), contains an occurrence of an *item* with a

replaced by b. Here an *item* may be a port label, an assignment, etc. For example

$$(a \in \{x,y,z\})\ \alpha \rightarrow a$$

will generate a set

$$\alpha \rightarrow x,\ \ \alpha \rightarrow y,\ \alpha \rightarrow z$$

where the *item* is an assignment.

6. Mapping Rules for Expressions without Array Creation Constructs

The translation of a PIPVAL expression without array creation constructs into SDFGL graphs is quite straightforward. The basic mapping rules for such expressions are developed directly from Brock's translation algorithms [19]. These algorithms consist of two functions which respectively map ordinary Val expressions and iteration bodies into their graph representations. The translation algorithm for **for-iter** expressions is based on a combination of these functions.

In this chapter, we first study the rules for PIPVAL expressions without iterations, which are then used in the mapping rules of primitive **forall** and **for-construct** expressions. Sections 6.1-6.5 present translation rules for simple primitive expressions on a case by case basis. Section 6.6 addresses the issue of pipelining for the result graphs generated for such expressions. A brief outline of the mapping of the **for-iter** expression is given in Section 6.7

The two major array construction expressions — the PIPVAL **forall** and **for-construct** expressions — are considered as special **for-iter** expressions (also called *loops* in [6]). The structure of the two types of expressions makes it possible to present their mapping rules in a simpler fashion than that for general iterative expressions. In Section 6.7, we briefly outline the basic mapping algorithm for **for-iter** expressions. Our goal is to provide a basis for introducing the specialized mapping rules for the two array construction expressions in the rest of this thesis.

6.1 Mapping Rules — M⟦id⟧, M⟦const⟧, M⟦op exp⟧, M⟦exp op exp⟧

The mapping rule for M⟦id⟧ is very simple, as shown in Figure 6.1 (1a). The result SDFGL graph is shown in Figure 6.1 (1b). The graph has a single input port labeled id, and a single output port labeled 1. In the <components> part, there is only one assignment by which the input port is directly connected (assigned) to the output port.

The mapping rule for M⟦const⟧ is also simple, as shown in Figure 6.1 (2a). The result

M [[id]]

inputs port: id
outputs port: 1
components:
id → 1

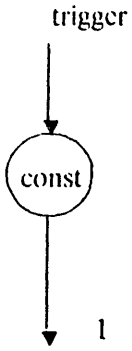
(1a)

(1b)

M [[const]]

input ports: trigger
output ports: 1
components:
const: inputs: trigger→trigger
outputs: 1 → 1

(2a)

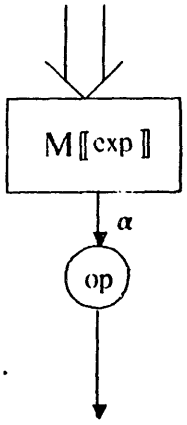


(2b)

M [[op exp]]

remarks (#OUT(exp) = #IN(op) = 1)
input ports: IN (M [[exp]])
output ports: OUT(op)
links : α
components:
M [[exp]] : inputs: a ∈ IN(M [[exp]]) a → a
outputs: 1 → α
op : inputs α → 1
outputs: 1 → 1

(3a)



(3b)

Figure 6.1. Mapping Rules M[[id]], M[[const]], M[[op exp]]

SDFGL graph is shown in Figure 6.1 (2b). It has a single input port labeled trigger and an output port labeled 1. It contains only one component — a constant actor with Const as its operator name.

The mapping rule of M⟦op exp⟧ is shown in Figure 6.1 (3a). The result SDFGL graph is shown in Figure 6.1 (3b). The graph M⟦op exp⟧ is constructed by connecting the output port of M⟦exp⟧ to the input port of the unary actor op. A requirement stated in the <remark> part is that the graph M⟦exp⟧ may only have one output port which provides an operand for the unary actor op.

The mapping rule of M⟦exp_1 op exp_2⟧ is shown in Figure 6.2 (a), where op is a binary operator. The result SDFGL graph is shown in Figure 6.2 (b). The graph can be constructed by connecting the two output ports from M⟦exp_1⟧ and M⟦exp_2⟧ to the two input ports of actor op respectively. The set of input ports of the result graph is the union of the input ports of exp1 and exp2, and these are assigned to the subgraph for the two subexpressions. The output port of the op actor is assigned to the output port of the result graph. A requirement stated in the <remark> part is that M⟦exp_1⟧ and M⟦exp_2⟧ must both have exactly one single output port, and op must be a binary actor.

6.2 The Mapping Rule for exp,exp

The mapping rule of exp,exp is shown in Figure 6.3 (a). The result SDFGL graph is shown in Figure 6.3 (b). The graph of an expression with higher arity such as M⟦exp_1, exp_2⟧ is constructed from the two subgraphs of M⟦exp_1⟧ and M ⟦exp_2⟧ in a straightforward way. The input ports of either subgraphs are connected to the graph input ports with the same label respectively. The output ports of M⟦exp_1⟧, ranging from 1 to #OUT(M⟦exp_1⟧), are assigned to the output ports of the graph with the same label, respectively. The output ports of M⟦exp_2⟧, ranging from 1 to #OUT(M⟦exp_2⟧), are also assigned to the corresponding graph output ports. However, in order to distinguish the two sets of output labels, the labels of the output ports for M⟦exp_2⟧ are shifted by #OUT(M⟦exp_1⟧), i.e., they

$M[\![exp_1\ op\ exp_2]\!]$

def G1 = $M[\![exp_1]\!]$
G2 = $M[\![exp_2]\!]$

remarks: #(IN(op)) = 2, #OUT(G1) = #OUT(G2) = 1

input ports: IN(G1) ∪ IN(G2)

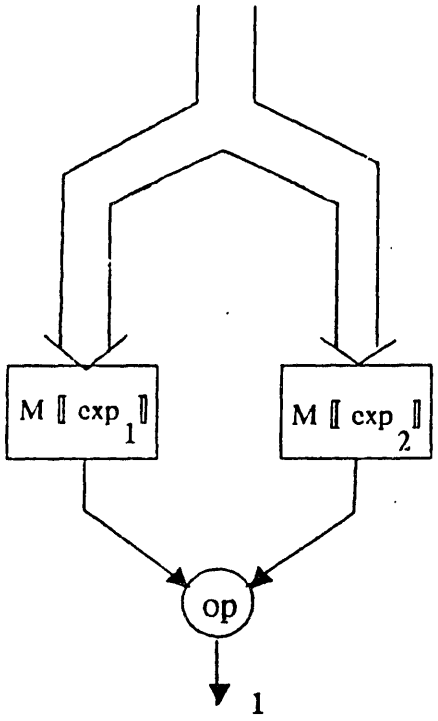
output ports: 1

links: α_1, α_2

components:

$M[\![exp_1]\!]$ inputs: $(a \in IN(G1))a \rightarrow a$
outputs: $(i \in OUT(G1))i \rightarrow \alpha_1$
$M[\![exp_2]\!]$ inputs: $(a \in IN(G2))a \rightarrow a$
outputs: $(i \in OUT(G2))i \rightarrow \alpha_2$
op inputs: $\alpha_1 \rightarrow 1, \alpha_2 \rightarrow 2$
outputs: $1 \rightarrow 1$

(a)



(b)

Figure 6.2. The Mapping Rule for $M[\![exp_1\ op\ exp_2]\!]$

now range from #OUT($M[\![exp_1]\!]$) + 1 to #OUT($M[\![exp_1]\!]$) + #OUT($M[\![exp_2]\!]$).

6.3 The Mapping Rule for Let-in Expressions

The mapping rule for let $id_1, id_2...id_k = exp_1...exp_k$ **in** exp **endlet** is shown in Figure 6.4. The definition part in a **let-in** expression is used to introduce and define value names $id_1, id_2,...id_k$. Hence the free value names of the entire **let-in** expression are the free value

$M[\![exp_1, exp_2]\!]$

def $G1 = M[\![exp_1]\!]$
$G2 = M[\![exp_2]\!]$
$n = \#(OUT(G1)) + \#(OUT(G2))$

input ports: $IN(G1) \cup IN(G2)$

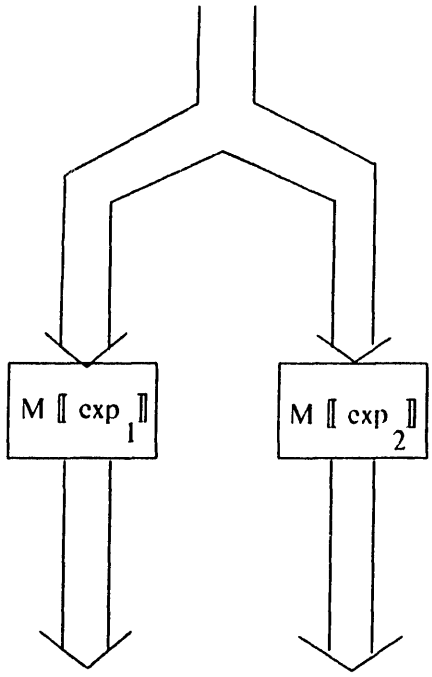
output ports: $(i \in \{1...n\})i$

components:

$M[\![exp_1]\!]$ **inputs:** $(a \in IN(G1))a \rightarrow a$
outputs: $(i \in OUT(G1))i \rightarrow i$

$M[\![exp_2]\!]$ **inputs:** $(a \in IN(G2))a \rightarrow a$
outputs: $(i \in OUT(G2))i \rightarrow i + \#(OUT(G1))$

(a)



(b)

Figure 6.3. The mapping rule for exp,exp

names of exp_1 through exp_k plus the free value names in exp less $id_1...id_k$. The result SDFGL graph is shown in Figure 6.5.

$[\![$let $id_1,....,id_n = exp_1,....,exp_n$ in exp endlet$]\!]$

def: $G_1 = M[\![exp_1]\!]...G_n = M[\![exp_n]\!]$
$G = M[\![exp]\!]$
$I1 = IN(G_1) \cup ... \cup IN(G_n)$
$I2 = IN(G)$

remarks: $\#(OUT(G_1)) = ... = \#(OUT(G_n)) = 1$

input ports: $(a \in I1 \cup I2)a$

output ports: $(i \in OUT(G))i$

links: $(i \in \{1...n\})\alpha_i$

components:

$(i \in \{1...n\})G_i$ inputs: $(a \in IN(G_i))a \rightarrow a$
outputs: $1 \rightarrow \alpha_i$

G inputs: $(i \in \{1...n\})\alpha_i \rightarrow id_i$, $(a \in (I2-\{id_1...id_n\}))a \rightarrow a$
outputs: $(i \in OUT(G))i \rightarrow i$

Figure 6.4. The Mapping Rule for a Let-in Expression

6.4 The Mapping Rule for Conditional Expressions

6.4.1 A Simple Conditional Expression

The mapping rule for a simple conditional expression **if** exp_1 **then** exp_2 **else** exp_3 **endif** is shown in Figure 6.6. The result graph is shown in Figure 6.7. It is constructed by the appropriate interconnection of the three subgraphs $M[\![exp_1]\!]$, $M[\![exp_2]\!]$, $M[\![exp_3]\!]$ listed in the component part.

The evaluation of the boolean-valued expression exp_1 will control which arm (exp_2 or

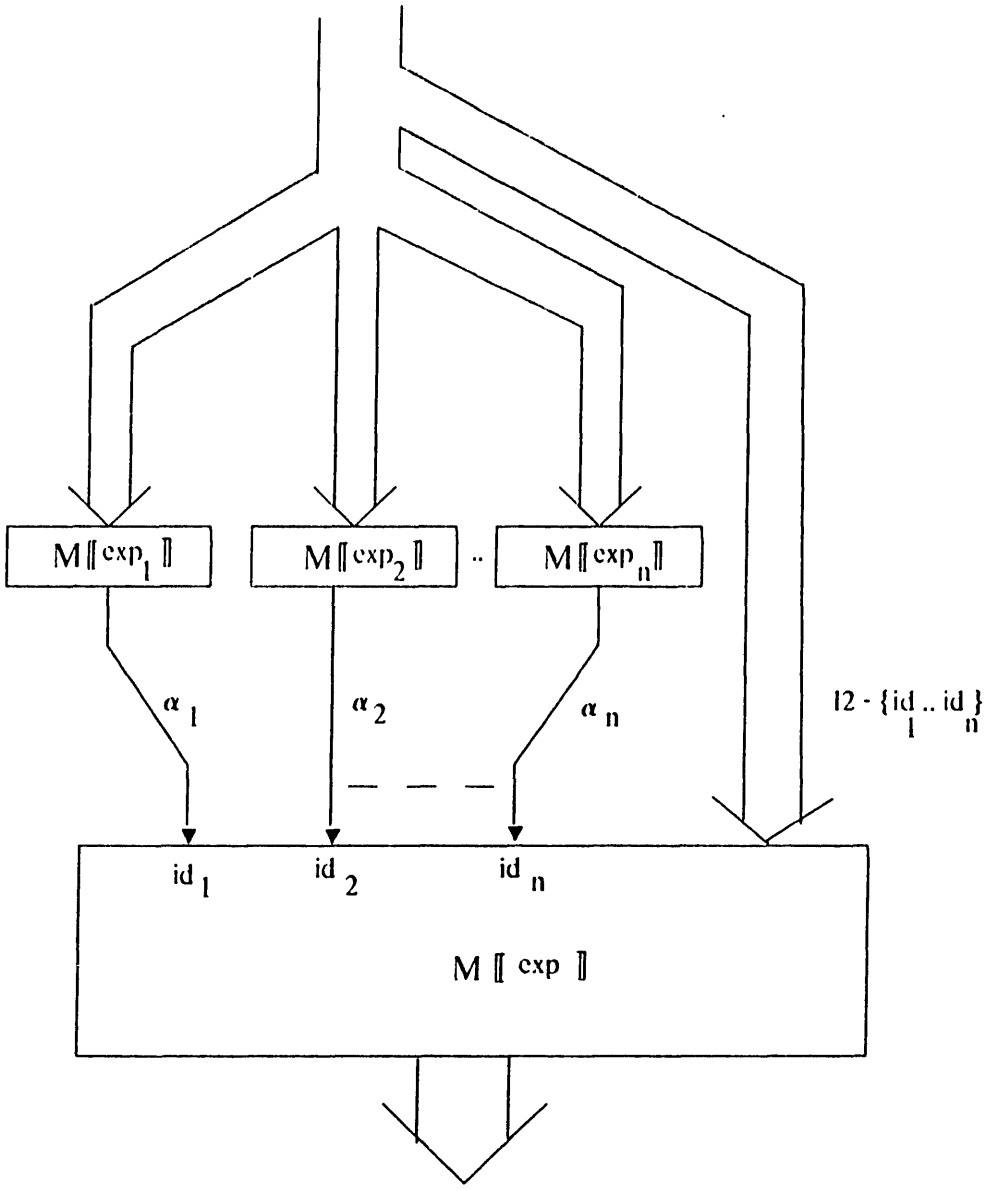


Figure 6.5. The SDFGL graph for mapping a Let-in Expression

exp_3) of the conditional expression will be evaluated. This is implemented by introducing a pair of T-gate actors and F-gate actors for each input to exp_2 and exp_3; these actors are controlled by the output of M[[exp_1]]. Furthermore, the output ports of the two arms should be combined for assignment to the output ports of the graph. This is implemented by a set of M-gate actors, one for each arm.

$M[\![\text{if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \text{endif}]\!]$

def $G_i = M[\![exp_i]\!]$ $\quad i = 1..3$

remarks: $\#OUT(G_1) = 1$, $\#OUT(G_2) = \#OUT(G_3) = n$

input ports: $IN(G_1) \cup IN(G_2) \cup IN(G_3)$

output ports: 1...n

links: $\alpha, (a \in IN(G_2))\beta_a, (a \in IN(G_3))\gamma_a, (a \in OUT(G_2))\delta_a, (a \in OUT(G_3))\lambda_a$

components:

$M[\![exp_1]\!]$ inputs: $(a \in IN(G_1))a \rightarrow a$
outputs: $1 \rightarrow \alpha$
$(a \in IN(G_2))$T-gate inputs: $\alpha \rightarrow 1, a \rightarrow 2$
outputs: $1 \rightarrow \beta_a$
$(a \in IN(G_3))$F-gate inputs: $\alpha \rightarrow 1, a \rightarrow 2$
outputs: $1 \rightarrow \gamma_a$
$M[\![exp_2]\!]$ inputs: $(a \in IN(G_2))\beta_a \rightarrow a$
outputs: $(i \in OUT(G_2))i \rightarrow \delta_i$
$M[\![exp_3]\!]$ inputs: $(a \in IN(G_3))\gamma_a \rightarrow a$
outputs: $(i \in OUT(G_3))i \rightarrow \lambda_i$
$(i \in \{1...n\})$M-gate inputs: $\alpha \rightarrow 1, \delta_i \rightarrow 2, \lambda_i \rightarrow 3$
outputs: $1 \rightarrow i$

Figure 6.6. The mapping rule for simple conditional expressions

6.4.2 The Mapping of Conditional Expressions with Multiple Arms

A conditional expression with multiple arms is equivalent to a properly nested simple conditional expression as illustrated by the examples in Figure 6.8 (a) and (b). Therefore, the mapping rule illustrated in Figure 6.6 can be recursively applied to the nested version. For example, Figure 6.9 is the result SDFGL graph of the 4-arm conditional expression in

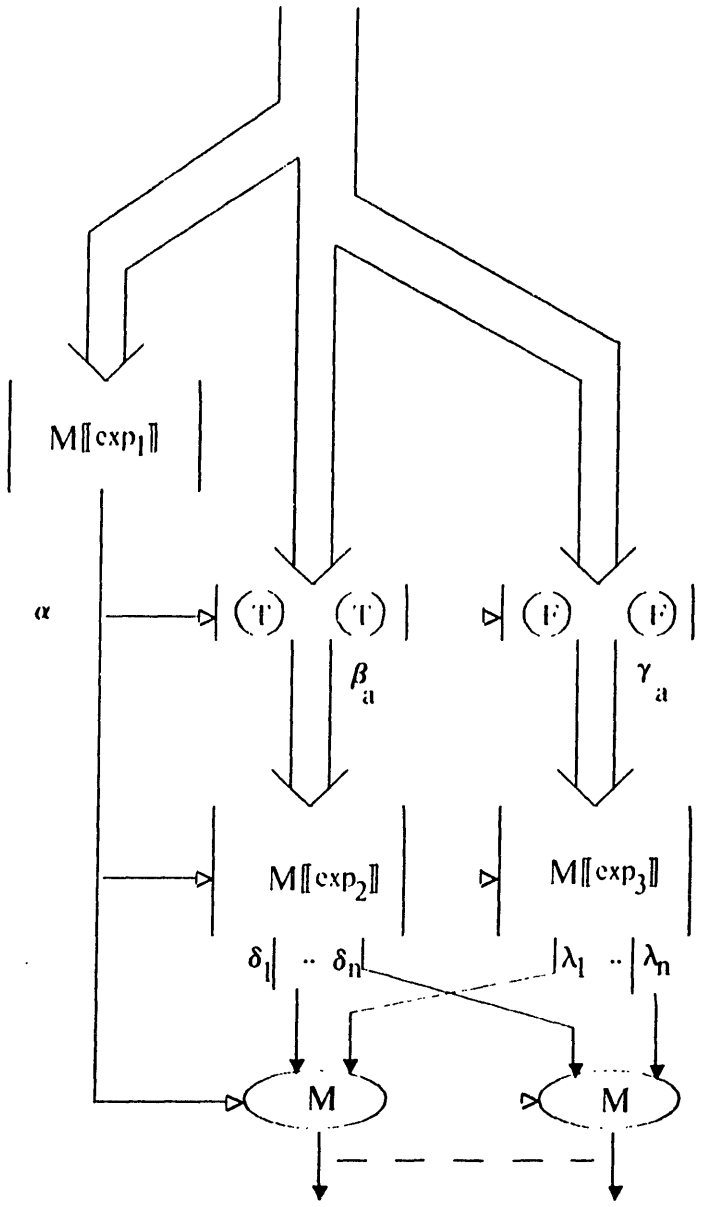


Figure 6.7. The SDFGL graph of mapping a simple conditional expression

Figure 6.8, derived by the application of the mapping rule of Figure 6.6 (without loss of generality, we assume there is only one input value name to expression x). From the graph, we note that the test expressions of the arms are evaluated in order until one becomes true, and the corresponding arm is selected. The expressions in the other arms will not be

```
if B1 then exp1
elseif B2 then exp2
elseif B3 then exp3
else exp4
endif
```

(a)

```
if B1 then exp1
else if B2 then exp2
     else if B3 then exp3
          else exp4
          endif
     endif
endif
```

(b)

Figure 6.8. A multi-armed conditional expression

evaluated at all. Furthermore, the test expressions of the arms following the selected arm are also not evaluated. This may result in a considerable saving of computational resources.

Unfortunately, the maximum depth of the T/F-gate network, as seen by the last arm, may grow linearly with the number of arms. The depth of the M-gate network may also grow linearly. Furthermore, the SDFGL graph representation becomes overwhelmed quickly by the T/F-gates and M-gates.

In this thesis, we propose a succinct version of the basic mapping rule for conditional expressions. It becomes particularly helpful in presenting the mapping rules of the **forall** and **for-construct** expressions, which usually have a multi-armed conditional expression as their range-partitioning expression. Our alternative representation of the mapping rule also gives hints about the machine design that may efficiently support such multi-armed conditional expressions.

Let us first consider the SDFGL graph in Figure 6.9. We can introduce some named subgraphs and reorganize the graph into Figure 6.10. First, subgraphs MB and MM are introduced. The subgraph MB evaluates testing expressions such as B_1, B_2 and B_3. It has an input port X, as well as five output ports: four boolean output ports, labeled 1 - 4 for each of the four arms, and a control output C which generates encoded control values. The

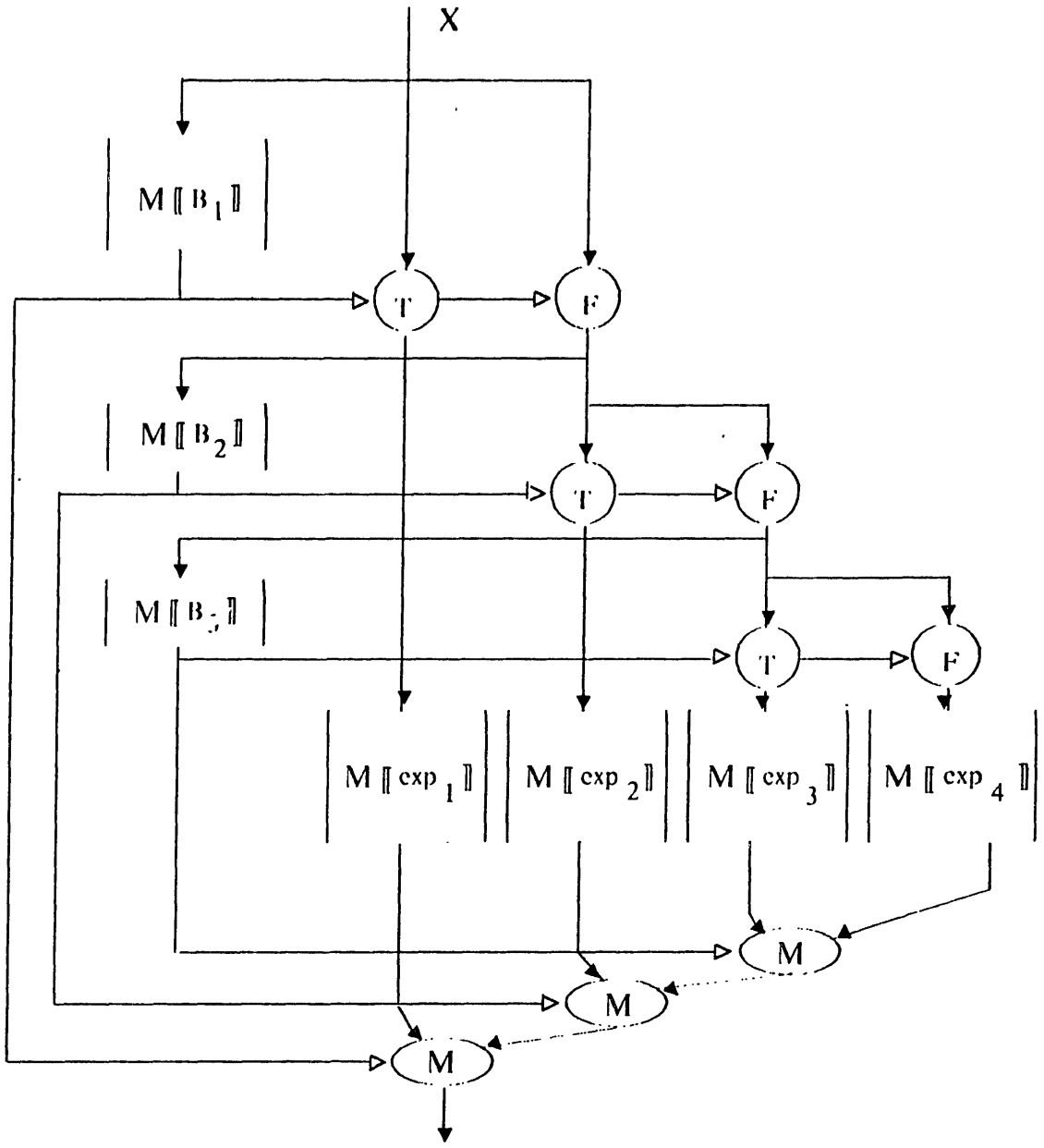


Figure 6.9. The SDFGL graph of a multi-armed conditional expression -- version 1

subgraph MM performs the function of the M-gate network in the old graph. It has four input ports labeled 1-4 for each of the four arms. It also has a control input port 0 which is usually connected to the control output port C of the corresponding MB subgraph.

The structure of MB is illustrated in Figure 6.11 (a), where B_1-B_3 are the graphs for

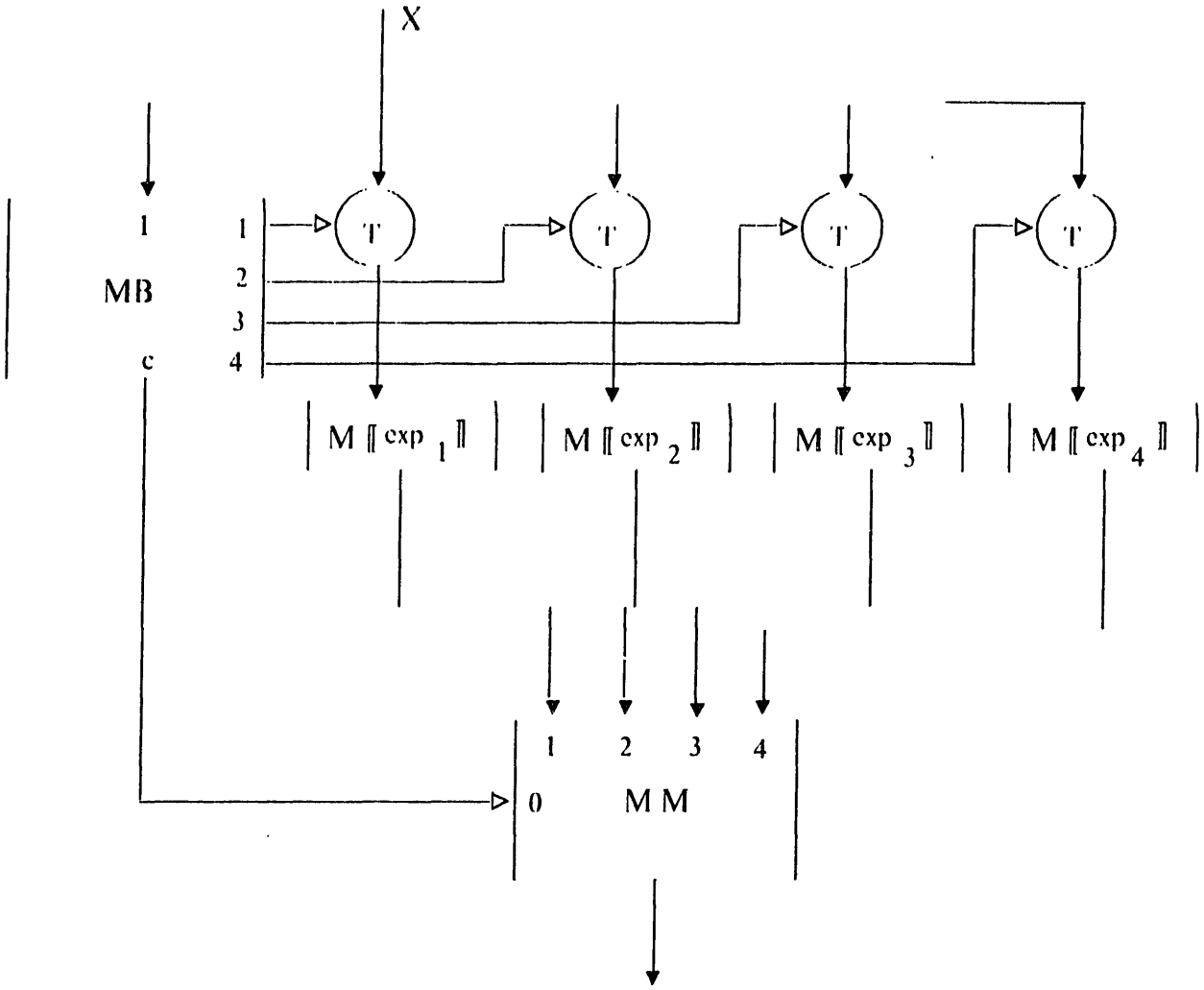
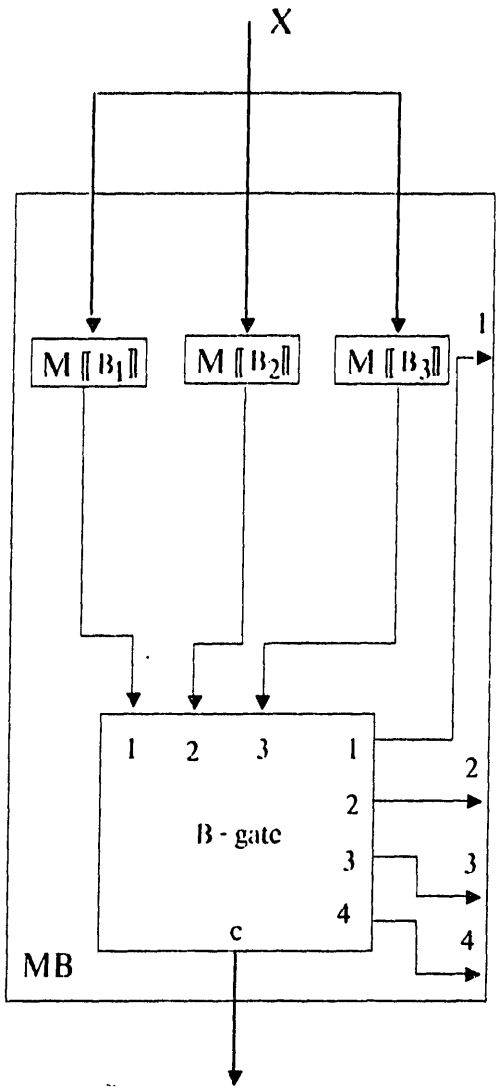


Figure 6.10. A SDFGL graph for a multi-armed conditional expression -- version 2

the test expressions. The function of the actor B-gate (branch gate) is illustrated by the truth table in Figure 6.11. If B_i (i = 1,2,3) evaluates to T, B-gate actor will generate T at the output port i and an encoded control value "i" at the output port C. Otherwise, the last arm is selected, and it will generate a T value at the output port 4. An appropriate encoding value "i" is also generated on the port C.[1] The subgraph MM, upon receiving the control value "i", will decode and forward the values at its input port i to the output port.

---

1. For convenience, we assume "i" is an integer value, encoding the information that the ith arm is selected.



input			output				
1	2	3	1	2	3	4	c
T	-	-	T	F	F	F	1
F	T	-	F	T	F	F	2
F	F	T	F	F	T	F	3
F	F	F	F	F	F	T	4

Figure 6.11. The MB subgraph

Another change made to the old graph is to replace the T/F-gate network by a row of T-gate actors, one for each arm. When an arm is selected, the input value of x is passed only to that arm (see Figure 6.11).

It is easy to see that the graph in Figure 6.10 will compute the same function as the graph in Figure 6.9. A slight difference is that in the new graph, the test expressions are always evaluated, while in the old graph, if a text expression has value T, later test expressions will not be evaluated. This difference is not important with respect to the kind

of range-partitioning conditional expressions in which we are interested, because the test expressions are usually quite simple and their evaluation will not diverge.

The B-gate (and the MB subgraph) as well as the MM subgraph may be implemented by ordinary graph actors. However, for the purpose of efficiency, they may also be implemented directly by graph actors supported by special instructions in the target machine. The latter possibility is discussed in Chapter 11.

Finally, the new version of the basic mapping rule for a multi-armed conditional expression is presented in Figure 6.12. The result SDFGL graph is shown in Figure 6.13. For simplicity, the structure of the MB subgraph is not included; it can be formed easily based on the principle illustrated in Figure 6.11.

6.5 The Mapping Rule for Array Selection Operations

So far, only one type of expressions in <primitive-exp> has not been discussed — an array selection operation. Let us consider the expression A[exp] where A is an array value name and exp is an expression that computes an index value. In the source language, A denotes an array value consisting of a series of element values along with low and high bounds indicating index limits for these values. Assuming the evaluation of exp returns an index value i. An array selection A[exp] selects the ith element value of the array A.

Figure 6.14 (a) illustrates the mapping rule for the expression A[exp]. The result SDFGL graph is shown in Figure 6.14 (b). The array selection operation is directly translated into the graph actor SEL, and its connection to the subgraph M⟦exp⟧.

The array operation A[exp] can also be conceptually written in another version such as SELECT(A,exp), where SELECT can be considered as an array operation construct in the source language equivalent to the role of "[" and "]" in the original expression. Thus, the mapping rules for a primitive expression op(exp_1,exp_2) as outlined in Section 6.1 can be directly applied to generate the above mapping rule.

M[[if B_1 then exp_1
elseifB_2 then exp_2
.
.
elseifB_{k-1} then exp_{k-1}
else exp_k
endif]]

def: G_i = M[[exp_i]] i = 1..k
H_i = M[[B_i]] i = 1..k-1
H = $H_1 \cup H_2 \cup ... \cup H_{k-1}$

remarks: #OUT(G_1) = 1, #OUT(G_2) = .. = #OUT(G_k) = n
input ports: IN(G_1) ∪ IN(G_2) ∪ .. ∪ IN(G_k) ∪ IN(H_1) ∪ IN(H_2) ∪ .. ∪ IN(H_{k-1})

output ports: (i ∈ {1...n})i

links: (i ∈ {1...k})α_i, (i ∈ {1...k}) (a ∈ IN(G_i))β_a^i, (i ∈ {1...k})(a ∈ OUT(G_i))δ_a^i, α_c

components:

M[[MB]] inputs: (a ∈ IN(H))a → a
outputs: (i ∈ {1...k}) → α_i, c → α_c
(i ∈ {1...k})(a ∈ IN(G_i))T-gate inputs: α_i → 1, a → 2
outputs: 1 → β_a^i
(i ∈ {1...k})M[[exp_i]] inputs: (a ∈ IN(G_i))β_a^i → a
outputs: (j ∈ {1...n}))i → δ_j^i
(j ∈ {1...n})M-gate inputs: α_c → 0, (i ∈ {1...k})δ_j^i → i
outputs: 1 → i

Figure 6.12. The mapping rule for multi-armed conditional expression

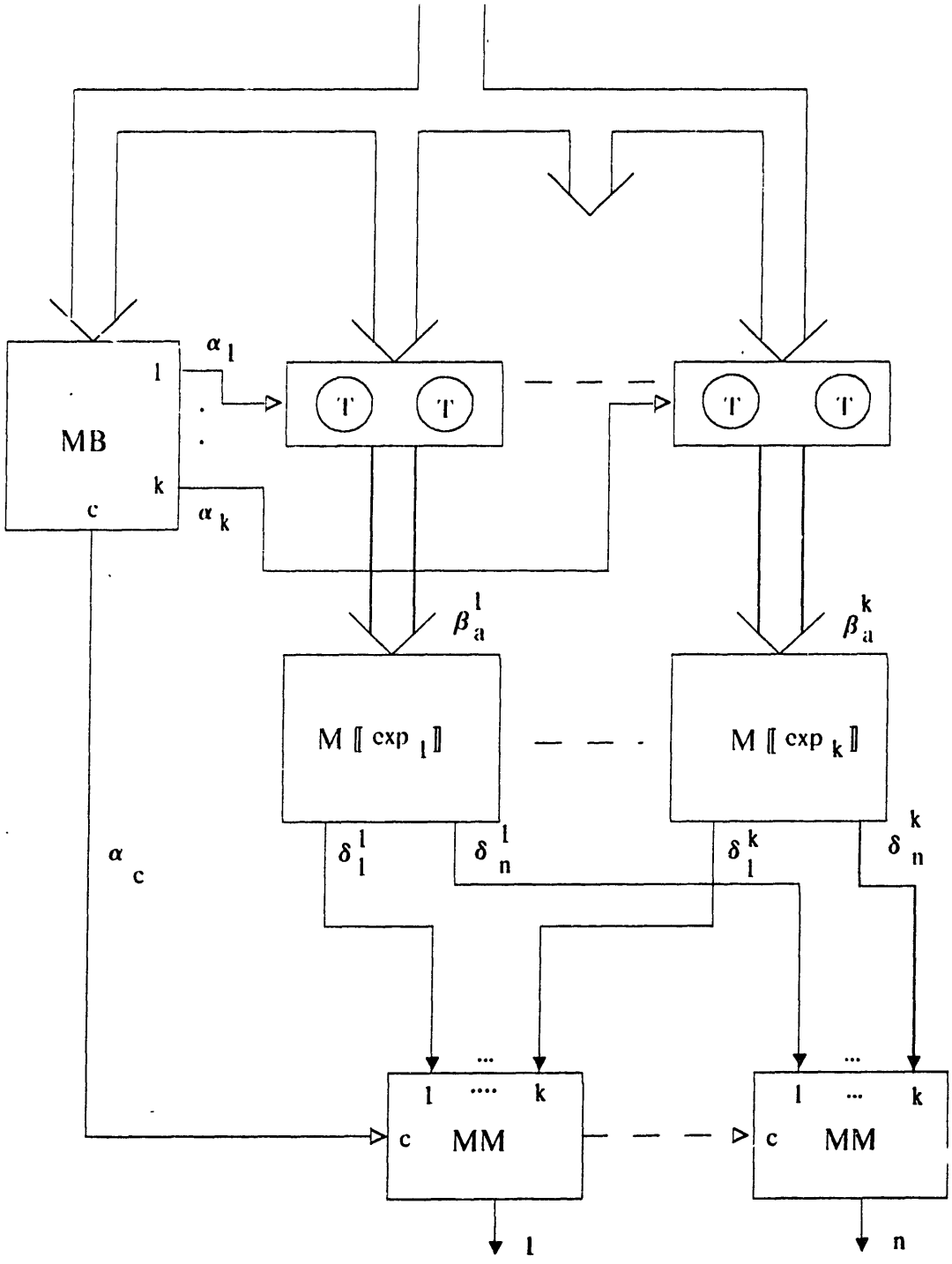


Figure 6.13. SDFGL graph showing the mapping of a multi-armed conditional expression

M[[id[exp]]]

def: G = [[exp]]

remarks: id is an array value name

input ports: IN(G)∪ id

output ports: 1

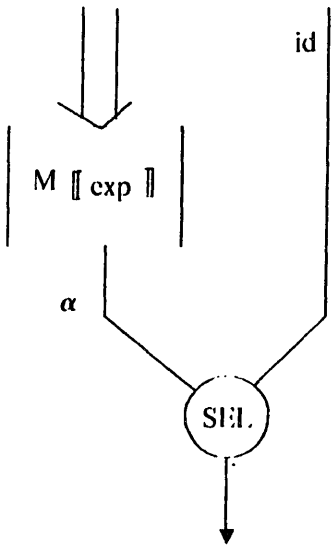
links: α

components:

M[[exp]] inputs: (a ∈ IN(G))a → a
outputs: 1 → α

SEL inputs: α → 1, id → 2
outputs: 1 → 1

(a)



(b)

Figure 6.14. The mapping rule for an array selection operation

6.6 Pipelining of the Graphs for Simple Primitive Expressions

The data flow graphs generated by the basic mapping rules described up to this point are acyclic [19] and all special actors are only used in forming conditional subgraphs (In terms of pipelining, MB and MM graphs can be considered as a multi-armed conditional subgraphs, and the principle of balancing simple conditional subgraphs can be extended easily to cover them). Hence, they can be balanced into maximally pipelined data flow graphs by the balancing scheme developed in Chapter 3. This fact is important because the bodies of the array construction expressions to be discussed later consist of such

expressions.

The pipelining of array selection operations needs more discussion. In order to conceptually use our pipelined execution model for static data flow graphs in Chapters 2 and 3, a SEL actor should receive as its input a sequence of tokens on both of its input ports. However, one of the input ports expects tokens carrying array values. The manipulation of array values directly in a data flow computer may be expensive [2]. As a result, the overhead may seriously degrade performance of the pipeline. This motivates our study of the optimization of array operations in later chapters.

6.7 An Overview of Mapping General Iteration Expressions

The basic translation algorithm described in [19] for an iterative expression such as the **for-iter** expression

for $id_1,id_2...id_k$ = exp
do iterbody
endfor

is outlined in Figure 6.15. It defines a separate mapping function M_I for the iterbody part. The graph M_I[[iterbody]] is an acyclic graph which has two lists of output ports I and R, and an iteration termination control output port labeled **iter?** (for simplicity, we use **?** to label the port). The set of ports in I is used to reiterate the values of the set of loop names redefined in the iterbody; the set of ports in R is used to return these values when the iteration is terminated; the output port **iter?** is used to signal which of the two possibilities has occurred.

For each loop value name id_1 - id_k in the iteration body, there is an FM-gate which will merge the values from the initialization expression (M[[exp]]) with the corresponding iteration output in the set of I ports of M_I[[iterbody]]. An FM-gate can be considered as an M-gate which has a built-in initial control input value F to ensure the initial data output

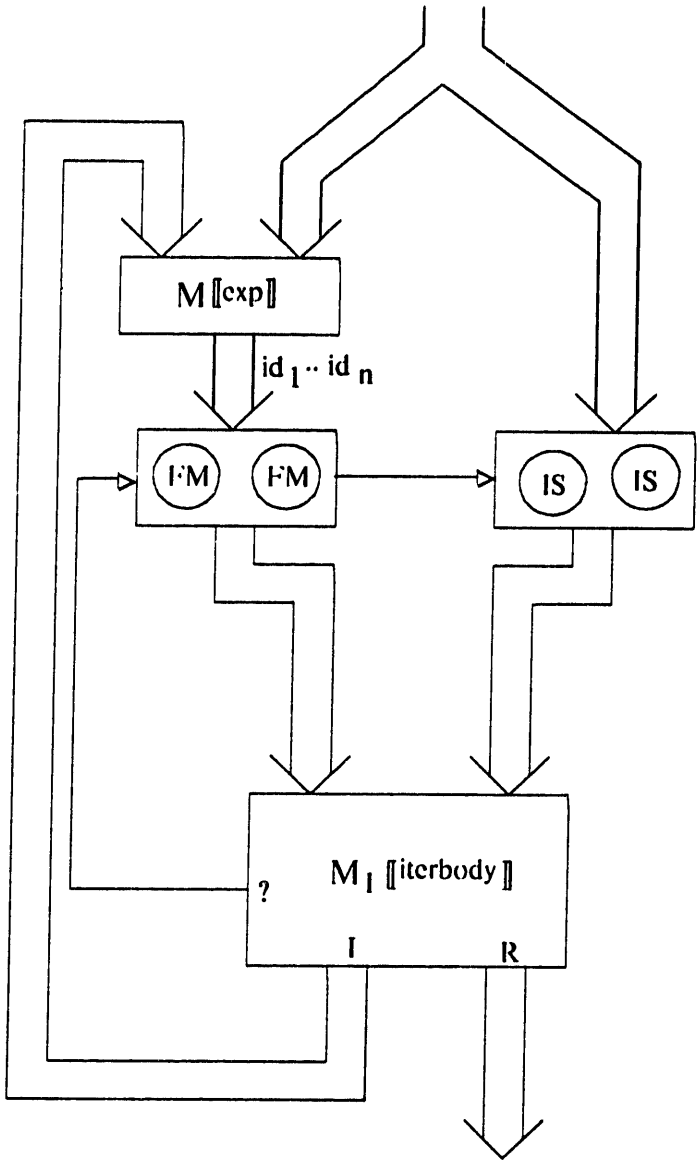


Figure 6.15. The mapping of a for-iter expression

value is selected from M[[exp]]. The control input of the M-gate is connected to the **iter?** output of M_1[[iterbody]]. There is also an IS-gate for each free value name in the iterbody controlled by **iter?** which also has a built-in initial control value F. Each of these will absorb and pass the first value received, and will keep generating the same value each time a T value is received from its control input port.

As stated before, we are mainly interested in the mapping schemes of **forall** and **for-construct** expressions, both are special cases of **for-iter** expressions. Since only the two kinds of expressions will be studied and extensively used, we do not specify the complete mapping rule of iteration expressions which can be found in [19]. Here we only outline the rule using a SDFGL graph in Figure 6.15. In Chapter 9, we will discuss the development of a special version of the mapping rule, and state its relation to the rule of mapping the **for-iter** expressions outlined above.

7. Mapping Scheme for One-Level Forall Expressions

In this chapter, we develop the basic mapping scheme, i.e., the basic mapping rule and the optimization procedure, for one-level **forall** expressions. In Chapter 8, we show how to extend the result to nested **forall** expressions.

In source programs for scientific computation, **forall** expressions often form a large portion of the code. Furthermore, there is usually massive parallelism embedded in such portions and the corresponding regularity of the array operations makes it very attractive for our pipelined code mapping schemes. Therefore, the **forall** construct deserves primary attention, and its mapping scheme is a most important part of the code mapping scheme developed in this thesis.

7.1 The Basic Mapping Rule

7.1.1 Pipelined Mapping Strategy

Let us first consider the one-level **forall** expression shown below

```
X =
        forall i in [0,m+1]
        construct
                if i = 0 then A[0]
                elseif i = m+1 then A[m+1]
                else
                        f(A[i-1],A[i],A[i+1],i)
                endif
        endall
```

For simplicity, the example code block has only one input array A, and the result array is X. We use f to denote a primitive expression which is the body of the code block.

Recall that **forall** is a parallel construct which states explicitly that there are no data

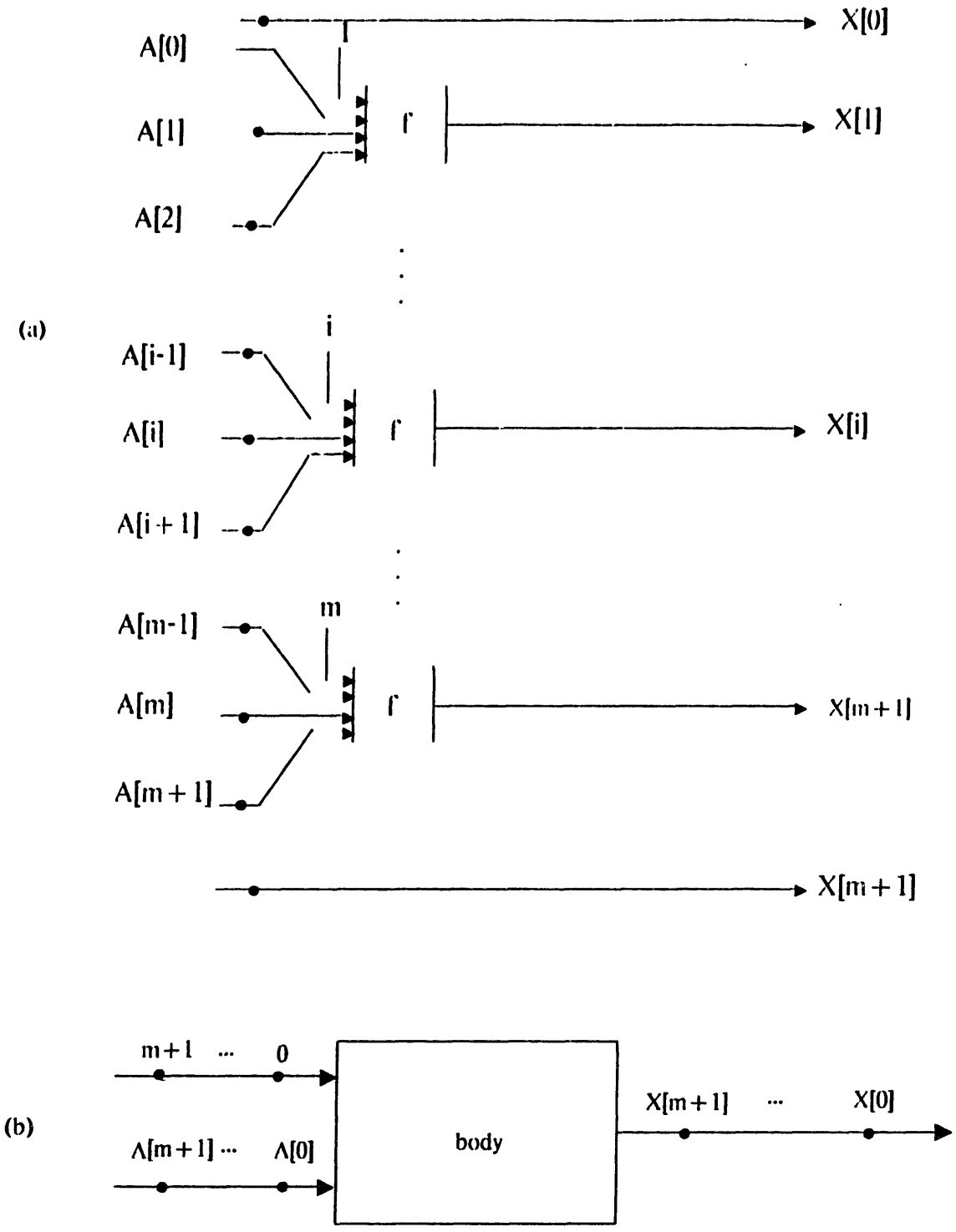


Figure 7.1. Parallel and pipelined mapping schemes for a forall expression

dependencies among elements of the array to be constructed. Therefore, we can choose a parallel mapping strategy, i.e., the graph consists of a copy of the program body for each array element, as shown in Figure 7.1 (a). Since the value of the index i is fixed for each copy, the top-most conditional vanishes. In order to perform such "full-parallel" mapping, the index bounds should be known before the data flow graph is generated. In the result graph, both the input array A and the output array X are in a parallel flattened representation.

In this thesis, we are mostly interested in a pipelined mapping scheme where computation is arranged in a way that the elements of the output array are generated in a pipelined fashion. Instead of providing multiple copies of the body, the pipelined mapping scheme uses one copy of the body and exploits the parallelism by means of pipelining. Therefore the element values of the input arrays of the code block, such as the array A in the above example, are consumed in a pipelined fashion. Since there are no data dependencies among the computations of the different array elements, the result array does not need to be fed back as an input to the body. Thus the pipelined mapping strategy does not introduce a cycle in the graph. This becomes a very important feature when maximum pipelining of the result graph is desired. Such a pipelined mapping strategy is illustrated in Figure 7.1 (b).

The potential advantages of the pipelined mapping scheme include the saving of considerable program memory space and the effective use of actors in data flow graphs. Furthermore, the overhead storage for the input/output arrays can be reduced or even eliminated. We will come back to this point after we present the mapping rule in the next section.

7.1.2 The Basic Mapping Rule

In the pipelined mapping scheme, a **forall** expression is equivalent to a special case of the **for-construct** expression. For example, the above **forall** expression is equivalent to the **for-construct** expression below

```
X =
    for i from m + 1 to 0
            T from array_empty
    construct
            if i = 0 then A[0]
            elseif i = m + 1 then A[m + 1]
            else
                    f(A[i-1],A[i],A[i + 1],i)
            endif
    endall
```

in terms of the result array value computed. Such **for-construct** expressions have no real data dependencies among the array elements generated in each iteration. Therefore the mapping rule for general **for-construct** expressions can be simplified to construct the basic mapping rule of **forall** expressions. However, the bulk of our discussion of the basic mapping scheme for a general **for-construct** expression will not be presented until Chapter 9. In this chapter, we merely present a simplified version tailored for mapping **forall** expressions. The version is straightforward enough to be understood easily, without going into the detail of the more general scheme for **for-construct** expressions. In this discussion, we will use some named SDFGL subgraphs to encapsulate such detail, and the reader may find a description of their internal structure in Chapter 9.

As mentioned above, the basic pipelined mapping rule of a **forall** expression can be derived directly from the basic mapping rule of an equivalent **for-construct** expression. However, the latter usually imposes a certain order in which the elements of the result array

M[[**forall** id **in** [exp_1,exp_2] **construct** exp **endall**]]

def

Λ = IN(M[[exp]]) - {id}
L = IN(M[[exp_1]])
H = IN(M[[exp_2]])
B = $\Lambda \cup L \cup H$

conditions

the result array is to be generated in the major normal order

remarks

#L = #H = 1, exp_1 and exp_2 are of type integer.
$l \leq h$ (where l = val(exp_1), h = val(exp_2))

input ports: $(a \in B)a$

output ports: 1

links: $(i \in \{1...5\})\alpha_i$, $(a \in \Lambda)\beta_a$

components:

($a \in \Lambda$)IS-gate
inputs: $\alpha_4 \rightarrow 1$, $a \rightarrow 2$
outputs: $1 \rightarrow \beta_a$

M[[exp_1]] inputs: $(a \in L)a \rightarrow a$
output: $1 \rightarrow \alpha_1$

M[[exp_2]] inputs: $(a \in H)a \rightarrow a$
output: $1 \rightarrow \alpha_2$

IGEN inputs: $\alpha_1 \rightarrow 1$, $\alpha_2 \rightarrow 2$
outputs: $1 \rightarrow \alpha_3$, $2 \rightarrow \alpha_4$

M[[exp]] inputs: $(a \in \Lambda)\beta_a \rightarrow a$, $\alpha_3 \rightarrow 1$
output: $1 \rightarrow \alpha_5$

AGEN inputs: $\alpha_4 \rightarrow 1$, $\alpha_3 \rightarrow 2$, $\alpha_5 \rightarrow 3$
output: R → 1, I → SINK

Figure 7.2. The mapping rule for a forall expression

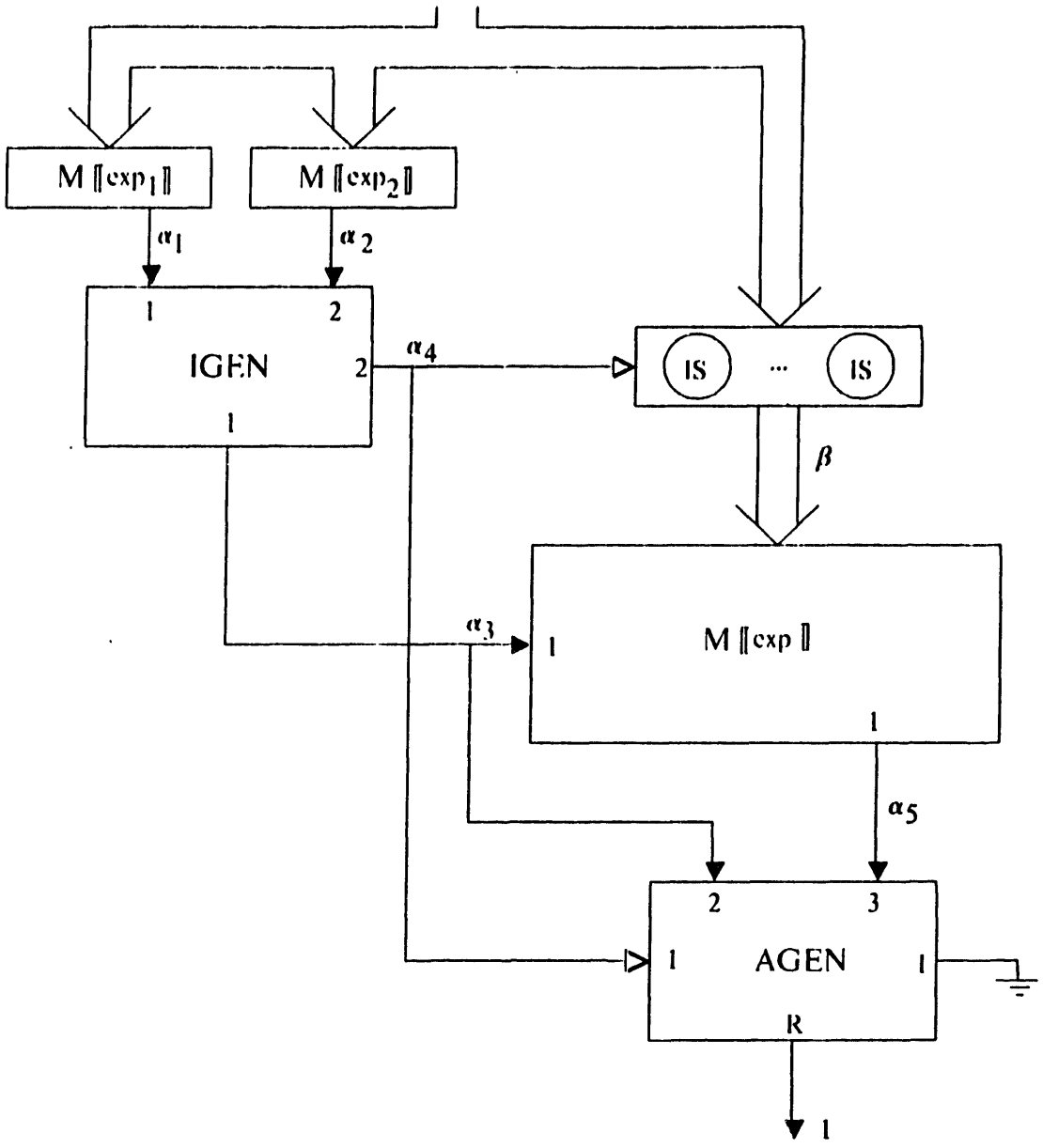


Figure 7.3. The SDFGL graph of mapping a forall expression

are generated. For example, the **for-construct** expression in the above example specifies the order of the index i as 1 to n. The index value name i controls the progress of the iteration (see Chapter 9). A **forall** expression does not demand any specific order to generate its elements. Therefore, the pipelined mapping strategy of a **forall** expression

must also state the preferred order in which the elements of the result array are to be produced, which is called the *generation order* of the array.[1] We include such information as part of our mapping rule representation. In Figure 7.2, we show the basic mapping rule for a one-level **forall** expression. The Figure 7.3 shows the result SDFG1. graph.

First note that in the <condition> part, it explicitly states in which order the elements of the result array are to be generated. In this case, the generation order goes from X[l] to X[h] (assuming X is the name of the result array), i.e., the array is generated in major normal order. The subgraph IGEN behaves as an "index generator". It has two input ports (labeled 1,2) for the low and high index limits respectively, and generates a sequence of h-l+1 index values: l...h at its output port (output port 1). Note that val(exp_1), val(exp_2) denote the values of exp_1, exp_2. Under the the condition l < h,[2] the order of IGEN conforms to the major normal order suggested in the <condition> part. IGEN also generates, at another output port (output port 2), a sequence of control values to control the progress of the iteration. In this case, the control sequence is T^mF, where m = h-l+1.

The result array is internally represented by a sequence of values carried by tokens on the output port of M[[exp]], i.e., α_5 in Figure 7.3. To assemble these elements into a result array A, another subgraph AGEN is used. AGEN has a control input port 1 which receives the control value sequence from the corresponding output of IGEN. The other two input ports 2,3 are for the sequence of index values and their corresponding array element values. AGEN has two output ports labeled R and I, which correspond to the R and I output ports in the M_1[[iterbody]] described in Section 6.7. In Figure 7.3, only the R output port is actually used. Since there are no data dependencies between array elements, no iteration path between the array value and the body is needed. Thus, the I output port is assigned to

---

1. As stated in Section 5.1, the two major orders are important to this thesis. A more thorough discussion of the generation orders of an array and other related concepts can be found in Chapter 8.
2. If l, h are compile-time computable, this condition can be automatically checked. Otherwise, it may be specified as an attribute provided by the user, or derived from other sources.

a SINK node.[1] The role of AGEN, under the control of the sequence of control input values, is to assemble the sequence of element values into the result array according to their corresponding indices. We can observe that the body expression is evaluated exactly m times, once for each index value in the range l...h, and these values become the element values of the result array.

For convenience, we introduce in Figure 7.4 a simplified SDFGL for the mapping of **forall** expressions. The links designated for the control values from IGEN are denoted by the dotted lined box passing through IGEN, AGEN and the IS-gate actors. The link between IGEN and AGEN for index values is omitted from the graph since it always exists.

In Chapter 9, we will examine the internal structure of IGEN and AGEN, and how they relate to the mapping of **for-construct** expressions. We will also see that the above mapping rule of **forall** expressions is a special case of that for a **for-construct** expression which is derived from Brock's translation algorithm.

As an example, in Figure 7.5 we apply the basic mapping rule to the primitive **forall** expression in Section 4.2.1. Note that the T-gates for array value A leading to each arm of the conditional expression are omitted for simplicity.

The mapping rule in Figure 7.2 is based on the condition that the generation order of the result array is a major normal order. We can also specify in the mapping rule (in the <condition> part) that the result array is to be generated in major reverse order. The only change in the mapping rule would be to reverse the connection of the two input ports of IGEN. The IGEN will then "count down" from h to l, generating the indices h, h-1...l

As we will show in Chapter 9, the graph M[[exp]] is generated recursively by applying the basic mapping rule to the body, and there is no specific restriction that it must be a simple expression. Thus, although our discussion is centered on one-level primitive **forall**

---

1. The role of SINK is to model a perfect sink (see Chapter 3) which can absorb input tokens. We denote it by the symbol shown in the figure.

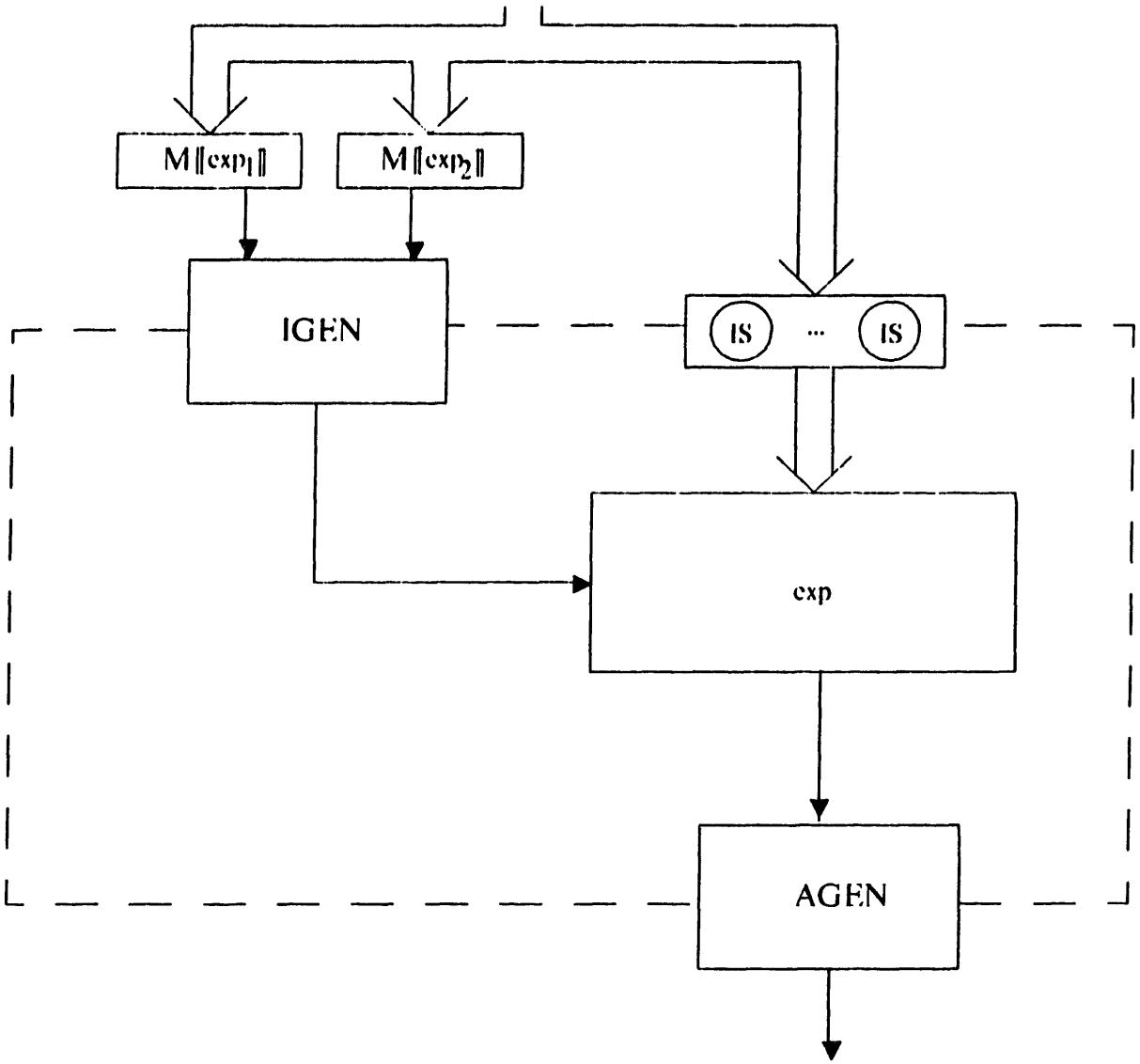


Figure 7.4. A simplified version of Figure 7.3

expressions in this chapter, the basic mapping rule presented in Figure 7.2 also applies to a **forall** expression the body of which is a PIPVAL expression, including another **forall** expression.

If the code block is a one-level primitive **forall** expression, the subgraph M[[exp]] is

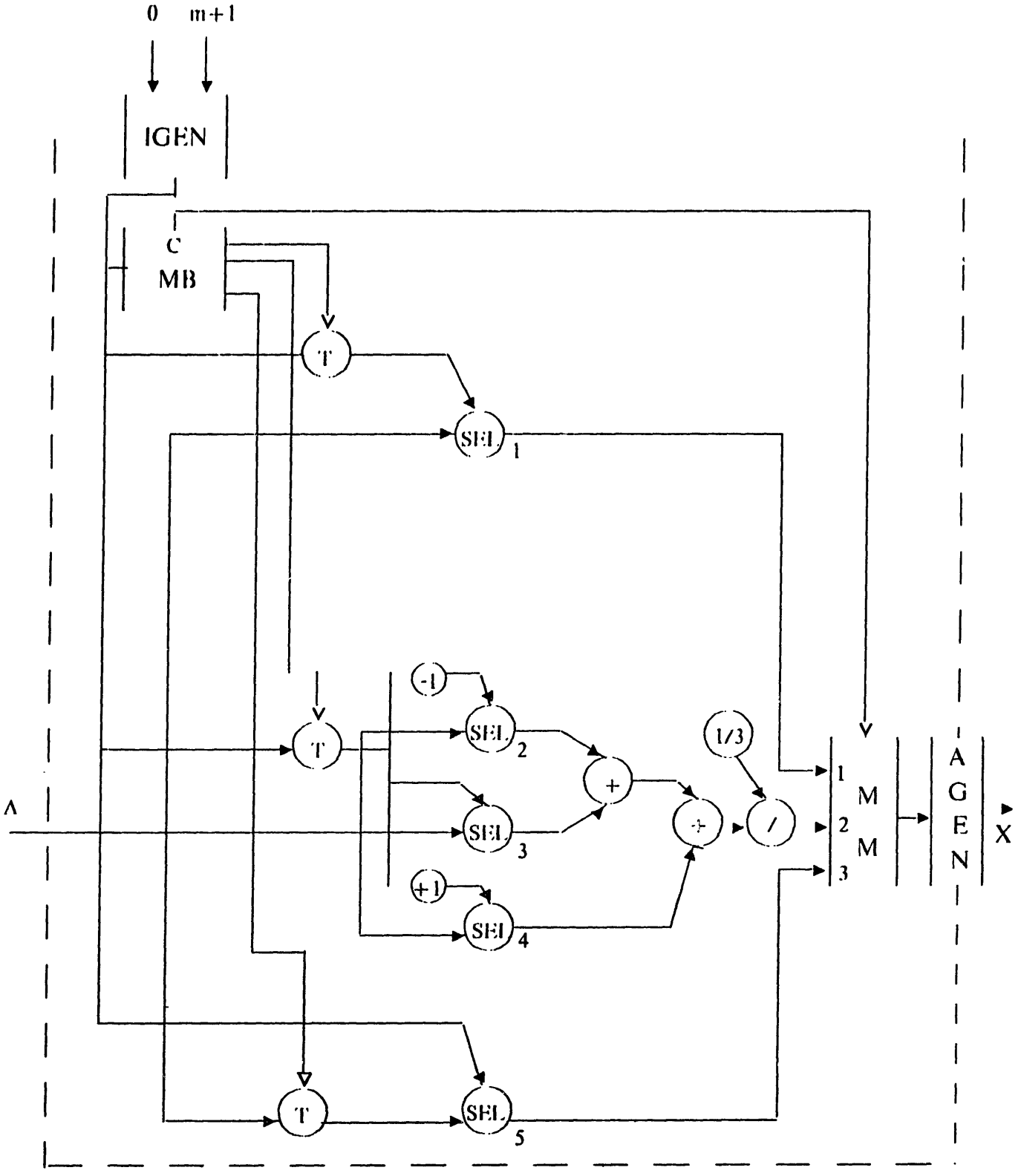


Figure 7.5. The SDFGL graph of mapping a one-level forall expression

acyclic. Since there is no feedback path between AGEN and its input, Map[[exp]] can be executed in a maximally pipelined fashion, provided the IGEN and AGEN subgraphs behave as a perfect pipelined source and sink. An implementation of the two subgraphs is discussed in Chapter 11. Another factor which affects the performance of the pipelining is how the elements of an input array are delivered to the code block and how elements of the result array are used. This factor is the major focus of the next section.

7.2 Optimization Of Array Operations

The basic mapping rules transform each PIPVAL array selection operation into a graph actor SEL, which performs the role of a "subscripted read" operation in conventional machines. It also generates a subgraph AGEN, which performs the role of a series of "subscripted write" operations. Therefore, the data flow graph generated directly by the basic mapping scheme may involve a considerable number of array operations, which are expensive in a data flow computer [2,3].

In many situations, however, two code blocks generating and using an array as a producer-consumer pair can be organized so that elements of the array are directly transmitted between the two blocks without using memory as an intermediate storage at all! This not only substantially saves storage space, but also removes all array operations, thus eliminating the overhead of array memory operations and the data traffic in the processor/memory interconnection network. Such optimization is the topic of this section.

7.2.1 An Example

Assume, in the data flow graph shown in Figure 7.5, that the input array A has index range [0,m + 1], the same as that of the code block. If the generation order of the elements of A is the same as the order in which they are consumed by the SEL actors in the body expression, it is perfectly possible to remove these SEL actors. The key is to arrange the mapping such that the elements of A used in the computation are pipelined passing

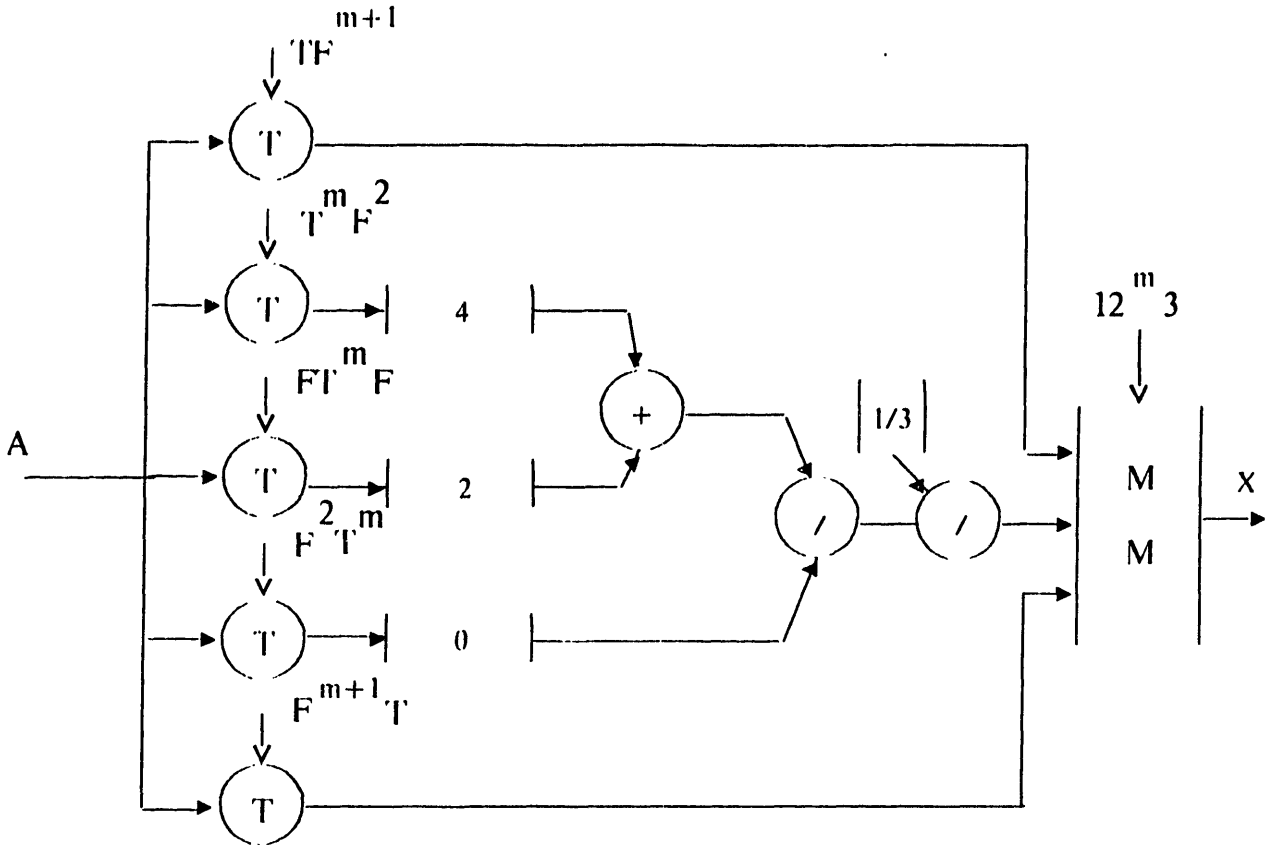


Figure 7.6. Result of Removing Array Operations in Figure 7.5

through in the right order and the unused values are discarded without causing jams.

In Figure 7.6 we show a data flow graph for the example in Figure 7.5 which satisfies these requirements. Compared with Figure 7.5, we see that each SEL actor (labeled 1 - 5 respectively) is replaced by a T-gate actor. The subgraphs computing the index expressions for each SEL actor are replaced by the proper boolean sequences, providing the control input for the corresponding T-gate actors.

As shown in the table of Figure 7.7, these T-gate actors act as filters which let through the element values in exactly the index subrange needed for the computation. Note that the 4th column in the table denotes the range of indices for elements selected by each array

SEL	index expression	index range	selection range	control value seqsuence
1	i	[0,0]	[0,0]	$T F^{m+1}$
2	i - 1	[1,m]	[0,m-1]	$T^{m} F^{2}$
3	i		[1,m]	$F T^{m} F$
4	i + 1		[2,m+1]	$F^{2} T^{m}$
5	i	[m+1,m+1]	[m+1,m+1]	$F^{m+1} T$

Figure 7.7. A table of selection index ranges for array slection operations in Figure 7.5

selection operation (or SEL actor) in the code block. It is called the *selection index range* of the corresponding SEL actor. We can note how the three array selection operations (i.e., A[i-1],A[i],A[i+1]) in the arm corresponding to the index range [1,m] are treated. Necessary skews are introduced by inserting FIFOs of proper size in the paths of A[i-1] and A[i] respectively.

The MM actor which merges the values from the three arms is also controlled by a sequence of encoded control values 12^m3. It passes the two boundary values (port 1 and 3) come from the first and the last arms, and selects the other m values from the middle arm (port 2). The result array X is represented as a sequence of element values X[0]...X[m+1] at the output port of MM. If the elements of X are also to be consumed by succeeding code blocks in the same order in which they were generated, there is no need to assemble them into an array value, thus eliminating the overhead of storing them in array memory

and handling the array value at runtime. Accordingly, the graph shown in Figure 7.6 also excludes the subgraph AGEN. As we can see, the graph after the transformation does not contain any array actors.

The basic ideas behind the optimization of array operations are straightforward. In the next section, we formulate how to perform such transformation.

7.2.2 Optimization of Array Selection Operations with Compile-Time Computable Selection Index Ranges

Consider the range-partitioning expression of a **forall** expression shown in Figure 7.8, which is a one-level primitive **forall** expression with index i ranging from l to h ($l \leq h$). We assume the bounds of the index range (i.e. l,h) of the code block are compile-time computable constants. The range-partitioning conditional expression is said to be in *standard form* if the following holds: $l_1 = 1$, $l_m = h_{m-1} + 1$ ($m = 2...k-1$) and $h_{k-1} \leq h$.

Assume, that the above **forall** has an input array A with the same index range [l,h]. In the data flow graph produced by the basic mapping scheme, A is accessed by SEL actors

```
X =
        forall i in [l,h]
        construct
                if l1 <= i <= h1 then exp1
                elseif l2 <= i <= h2 then exp2
                        .
                        .
                elseif lk-1 <= i <= hk-1 then expk-1
                else
                        expk
                endif
        endall
```

Figure 7.8. A range-partitioning conditional expression

which correspond to array selection operations A[i+b] in the source program. A key parameter in optimization is the selection index range of the array selection operations. Here we concentrate on situations where the selection index ranges are compile-time computable, and leave other situations to the next section.

If any arm in the body (exp_1-exp_k) does not contain further conditional expressions with predicates depending on the index i, the selection index range for A [i+b] can be easily computed. The example of **forall** expression used in Figure 7.5 belongs to such a class.

Case 1: A one-level primitive **forall** expression where the top range-partitioning expression is in its standard form as shown in Figure 7.8, and the following conditions hold: (1) the bounds of all subranges are compile-time constant; (2) exp_1 - exp_k do not contain any conditional expressions whose predicate expression depends on i, while having A accessed in any of its subexpressions.

Now let us discuss several major steps in the optimization procedure. without loss of generality, it is assumed to have only one input array: A.

1. Selection index range and control sequence computation

Let A[i+b] be an array selection operation which resides in the subexpression exp_m. Its selection index range [x,y] can be computed directly from the subrange $[l_m,h_m]$, by noting that

$$x = l_m + b \qquad (7.2.1)$$

$$y = h_m + b \qquad (7.2.2)$$

The only constraint is $l \leq x \leq y \leq h$. If this was not observed, the compiler would signal an

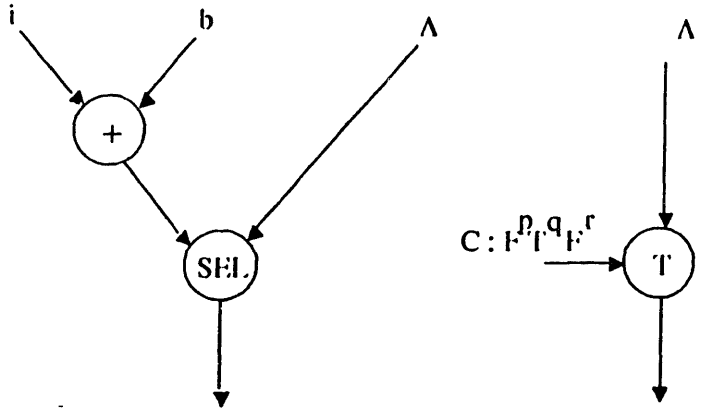
out-of-bound exception. A bound-checking routine should be available to handle such exceptions. For the purpose of our discussion, we assume that all array selection operations are bound correct.

As shown in Figure 7.6, each SEL actor and its index calculation subgraph can be transformed into a properly controlled T-gate actor, provided that the array A is represented by a sequence of element values. Each T-gate is controlled by a boolean pipeline C in the form $F^pT^qF^r$. Such a transformation is illustrated in Figure 7.9 (a) and (b). From the selection index range [x,y] and the index range [l,h] of the array A the parameters p,q,r can be computed by

$$p = x - l \tag{7.2.3}$$

$$q = y - x + 1 \tag{7.2.4}$$

$$r = h - y \tag{7.2.5}$$



(a) (b)

note: A[i+b] has selection index range [x,y] where x, y are defined in (7.2.1) and (7.2.2)

$$p = x\text{-}l,\ q = y\text{-}x+1,\ r = h\text{-}y$$

Figure 7.9. The optimization of a SEL actor

If the elements of A are generated in the major normal order, a control sequence $C = F^{p}I^{q}F^{r}$ is constructed, from (7.2.3) - (7.2.5). The T-gate will then select exactly the element values in the selection index range in the right order. If the generation order of A is major reverse order, the control sequence should also be reversed: $C = F^{r}I^{q}F^{p}$.

2. Skew introduction and Skew FIFO adjustments

FIFOs should be introduced in order to achieve necessary skews and avoid jams. We propose that this be done in the following way. Let us still use the input array A as an example. Let $A[i+b_1]...A[i+b_t]$ be the set of array selections in one arm (say, the m-th

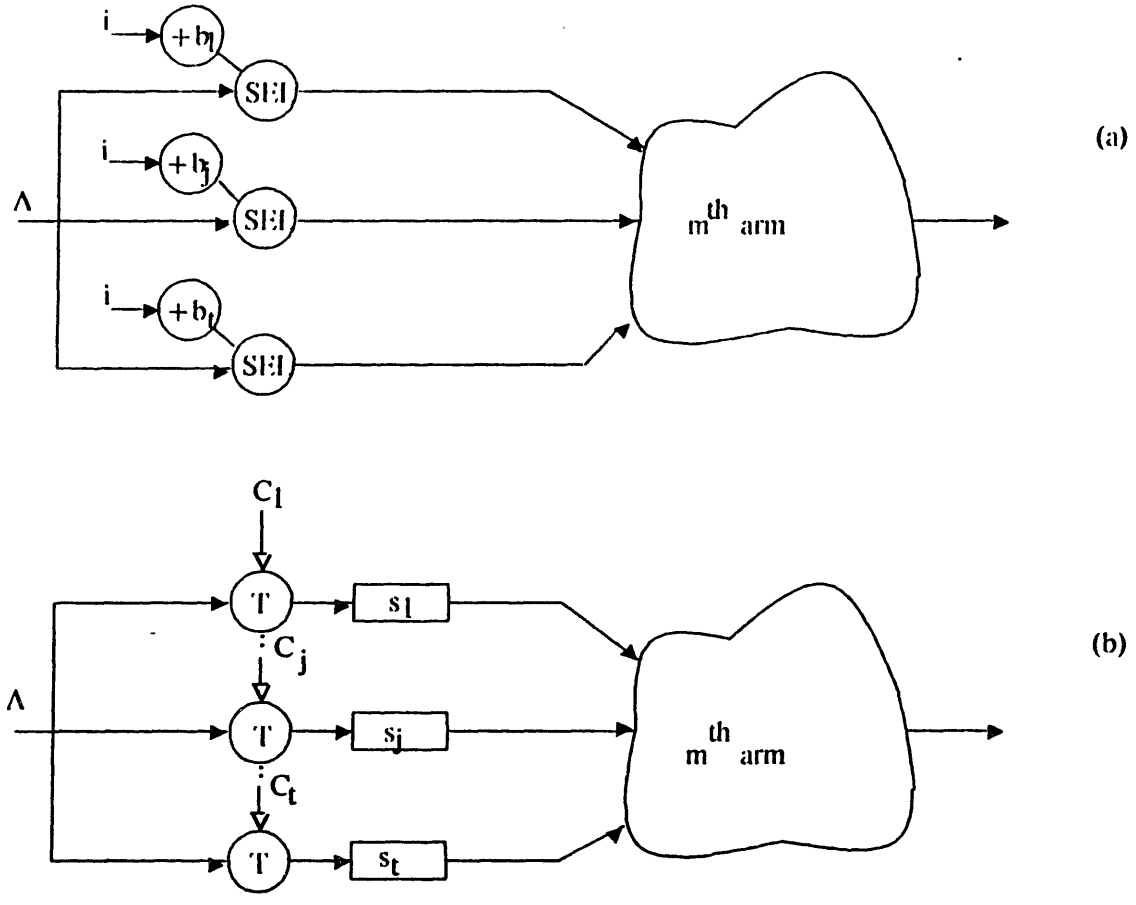


Figure 7.10. Skews in the optimization of array selection operations

arm), and let all the corresponding SEL actors be replaced, as outlined above, by properly controlled T-gate actors. Without loss of generality, we assume $b_1 < b_2 < ... < b_t$. This situation is shown in Figure 7.10 (a) and (b). Skews are introduced through FIFOs, where $s_j = 2^*(b_t - b_j)$ for $j = 1...t$.

3. Other simplifications.

At compile-time we can also compute the encoded value sequence for the control input of the MM subgraph as $1^{t1}2^{t2}...k^{tk}$, where

$$t_i = h_i - l_i + 1 \qquad i = 1...k-1 \qquad (7.2.6)$$

$$t_k = h - h_{k-1} \qquad (7.2.7)$$

as shown in Figure 7.11. After this is done, the MB subgraph for the range-partitioning conditional expression may not be in use and becomes "dead code" which can be removed. Such may also happen to the IGEN subgraph, if all of its outputs are no longer used.

The efficient implementation of the optimization depends on target machine design,

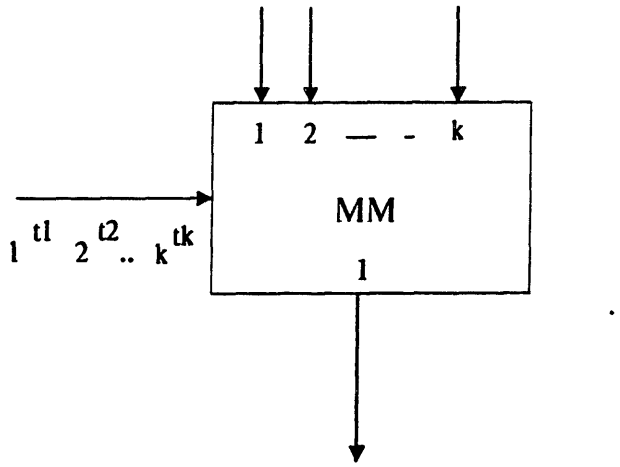


Figure 7.11. The Control Sequence of MM actor

in particular machine support of boolean and encoded control sequences, as well as the T-gate and MM actors. A brief discussion is included in Chapter 11.

Now, we summarize the above transformation in the form of an optimization procedure OPMAP for the case-1 **forall** expression. This procedure, as well as other procedures to be presented later, takes as input a data flow graph generated by the basic mapping rules, and transforms it into a graph in which array actors such as AGEN and SEL actors are removed or replaced by ordinary graph actors. The optimization procedure is shown in Figure 7.13. Since the output array is to be generated in the major normal order, the removal of AGEN in Step 0 is justified. The validity of the transformation of each SEL actor and its index calculation subgraph (Steps 1 — 3) is based on the conditions listed in the condition part of the procedure, and the validity of the selection index range calculation of the corresponding array selection operation A[i+b] are based on (7.2.1) to (7.2.5). The skew adjustments (step 3) are based on our earlier analysis. The remaining steps are straightforward.

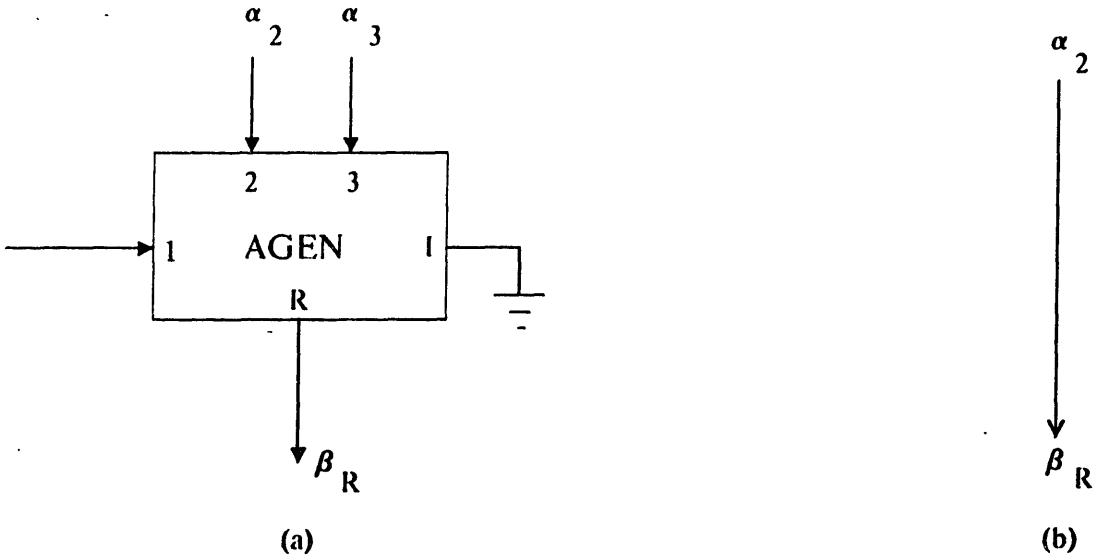


Figure 7.12. The optimization of AGEN

'rocedure OPMAP'

Inputs: a data flow graph G derived by the application of the basic mapping rule to a one-level primitive forall expression E in case-1 standard form.

Outputs: a data flow graph G' with array operations in G optimized

Conditions: (1) G is mapped under the condition that the result array is to be generated in major normal order, (2) each input array has major normal generation order.

Method: Assume E is writen in the form as Figure 7.8. Remove AGEN in G properly. Let S be the set of input arrays in E. For each A in S, computes the selection index ranges for all its array selection operations. Replace the corresponding array actors in G with properly controlled T-gate actors.

The Steps:

Step 0: Replace all AGEN subgraph as shown in Figure 7.12.

Step 1: If S = Ø then go to step 5.

Step 2: Let A ∈ S. For each A[i+b] computes the values p,q,r according to (7.2.1) to (7.2.5). Find the SEL actors and their index calculation subgraph for each A[i+b]. Replace each SEL and its index calculation subgraph by a T-gate actor controlled by C: $F^pT^qF^r$ as illustrated by Figure 7.9 (a) and (b).

Step 3: For each arm of the range-partitioning conditional expression, perform the buffer size adjustment on all T-gates introduced in Step 3, as shown in Figure 7.10.

Step 4: S := S -{A}, goto Step 1.

Step 5: remove the arc connecting MB to the control input of MM (where MB and MM are associated with the range-partitioning conditional expression). An encoded sequence is computed from (7.2.6) and (7.2.7) and provided as the control input for MM as shown in Figure 7.11. Remove MB if it is not used.

Step 6: Remove IGEN, if it is not used.

Step 7: Stop.

Figure 7.13. The optimization procedure for case-1 forall expression

Now let us apply the optimization procedure OPMAP to the data flow graph in Figure 7.5. The table of valid selection ranges and control sequences shown in Figure 7.7 is

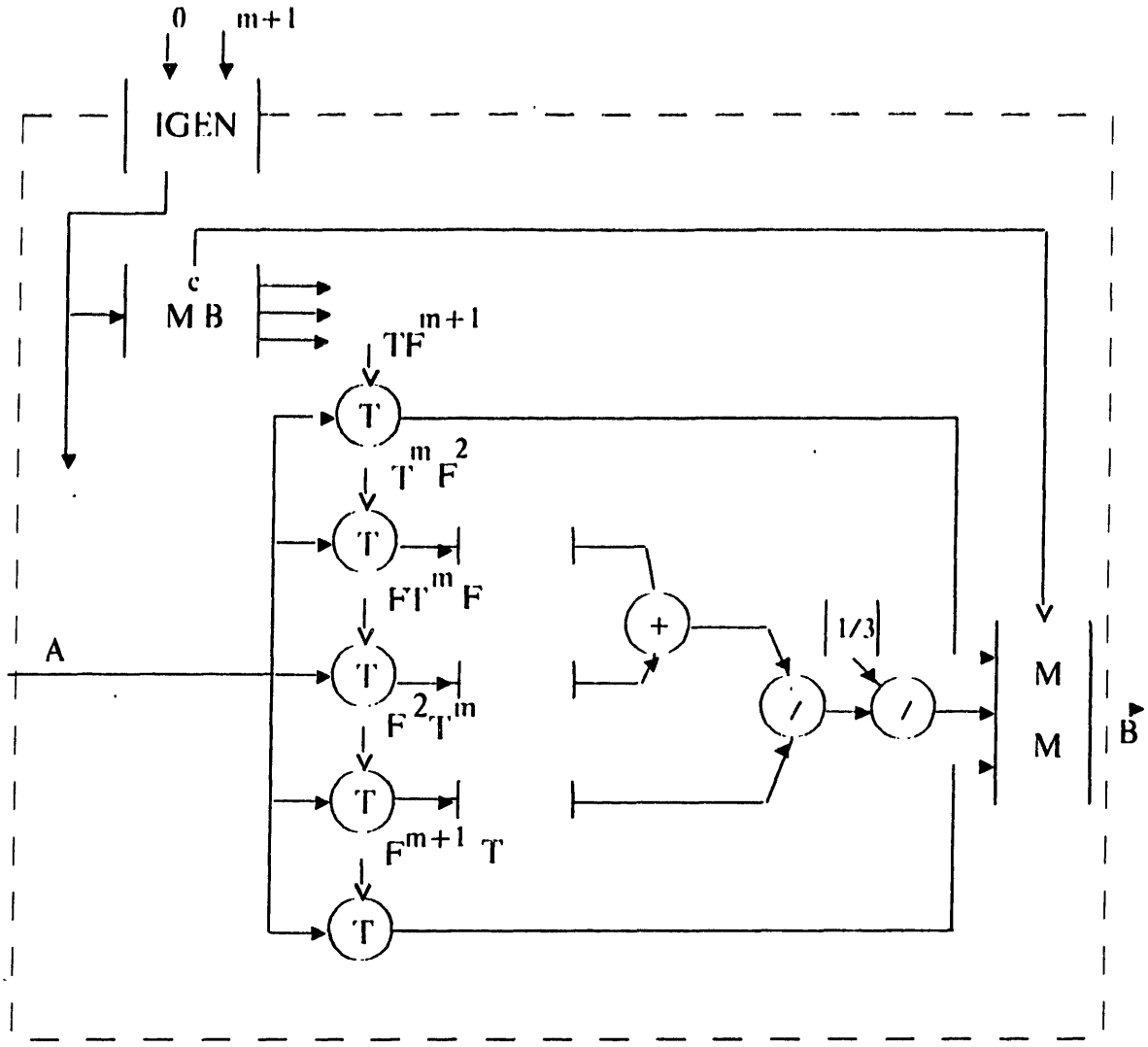


Figure 7.14. An example of optimization by OPMAP-- after Step 2

automatically computed by step 2. After step 2, the graph is transformed into the form shown in Figure 7.14, where all array actors are removed and replaced by T-gate actors. Step 3 adjusts the size of the FIFOs, and the result is shown in Figure 7.15. Step 5 replaces the control link from MB to MM by a proper encoding control value sequence, as shown in Figure 7.16. Step 5 and Step 6 eliminate MB and IGEN respectively. The final result of the OPMAP procedure is shown in Figure 7.17. We note that it is the same as the graph in

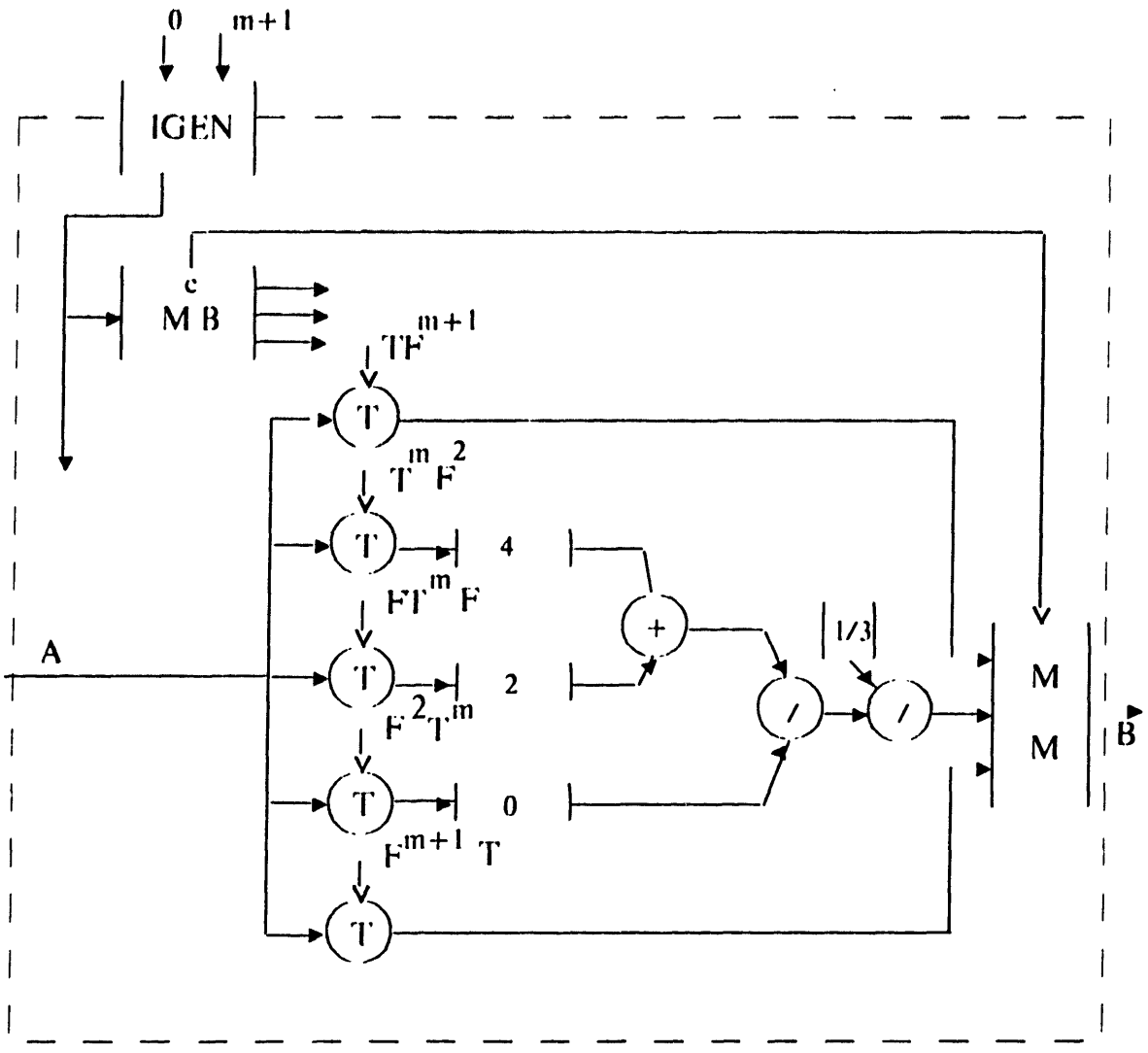


Figure 7.15. Continued from Figure 7.14-- after Step 3

Figure 7.6, as we expected.

In some cases, a primitive **forall** expression may not be directly expressed in the case-1 form, but can be transformed into this form by some simple source level transformation. For example, the top level range-partitioning conditional expression is not in the standard form, as illustrated in the following expression:

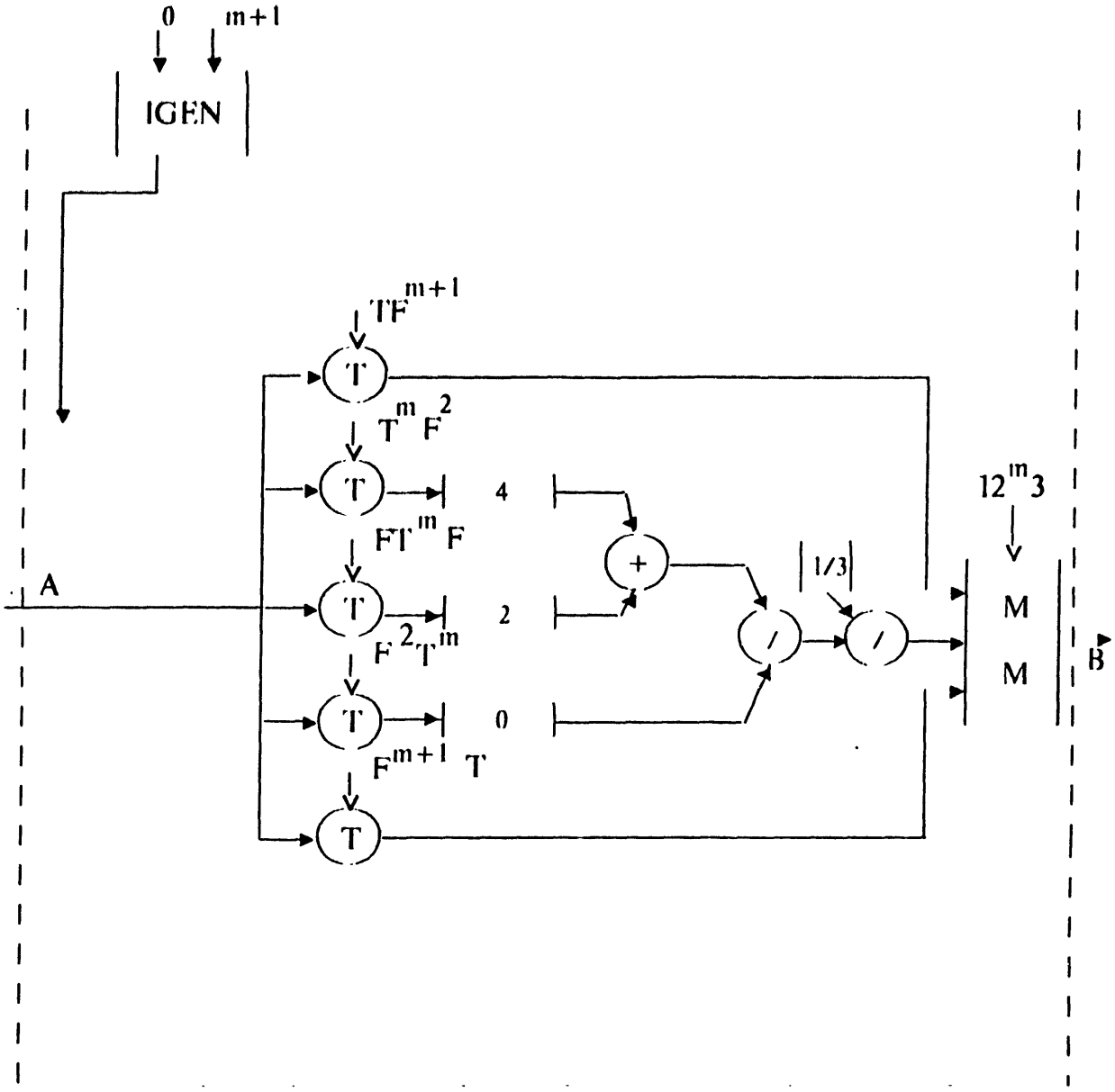


Figure 7.16. Continued from Figure 7.15-- after Step 5

if $i >= l_1 \mid i < l_2$ **than** exp_1
elseif $i >= l_2 \mid i < l_3$ **than** exp_2

.
.

elseif $i >= l_{k-1} \mid i < l_{k-1}$ **then** exp_{k-1}
else exp_k
endif

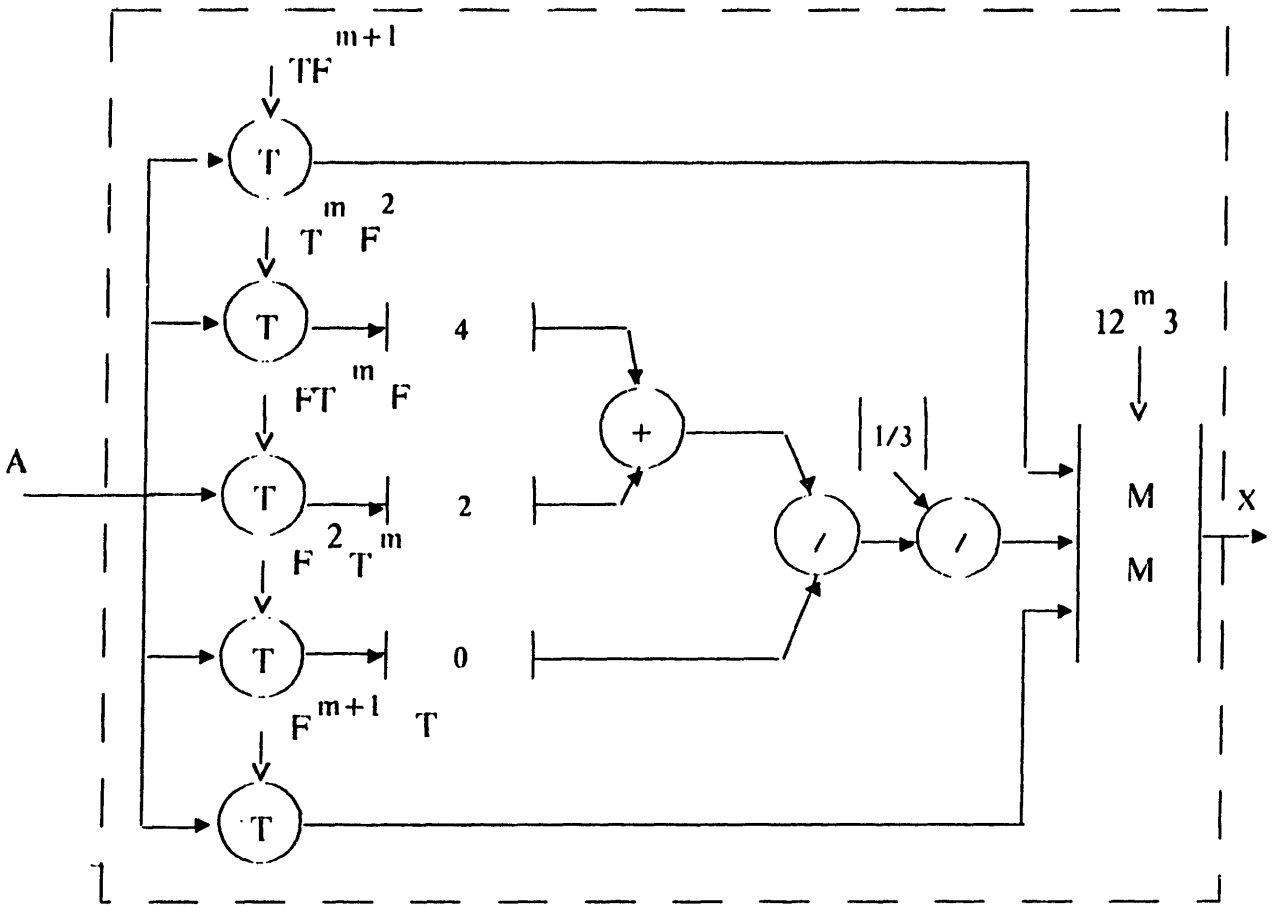


Figure 7.17. Continued from Figure 7.16-- Result

The above expression can easily be transformed into standard form by replacing l_2 by h_1, l_3 by h_2...., where $h_1 = l_2 - 1$, $h_2 = l_3 - 1$...., etc. Then all "<" signs in the predicate expression can be replaced by $\leq$ signs. After such transformation, the expression is in the standard case-1 form.

In another situation, the range-partitioning conditional expression may be in the form of a nested conditional expression. For example, let us consider the following range-partitioning expression:

```
if l1 <= i | i <= a then
        if l1 <= i | i <= b then exp1
        else exp2
        endif
elseif l3 <= i | i <= h3 then exp3
else exp4
endif
```

where $l_3 = a + 1$, $b < a$. This expression can also be transformed into the following standard case-1 form:

```
if l1 <= i | i <= h1 then exp1
elseif l2 <= i | i <= h2 then exp2
elseif l3 <= i | i <= h3 then exp3
else exp4
endif
```

where $h_1 = b$, $l_2 = b + 1$, $h_2 = a$.

From now on, we will assume all **forall** expressions in the above forms are transformed into standard case-1 form before the application of the optimization procedure.

In general, the graph derived after the optimization procedure may not be optimally balanced. We may need to apply balancing techniques, if maximum pipelining is desired.

7.2.3 Selection Index Ranges Not Computable at Compile-Time

In some situations, the selection index ranges of array selection operations are not compile-time computable. This happens, for example, when the subrange limits in the range-partitioning expression are not compile-time constants. In this section, we discuss

how to extend the optimization procedure to this situation, i.e.

Case 2 - as case 1 (section 7.2.2), except that $l_2...l_{k-1}$, $h_1...h_{k-1}$ are not compile-time constants.

We still assume that index range [l,h] is known at compile-time. Let us consider the selection index range [x,y] of A[i+b] in an arm, say exp_m. The key equations (7.2.1), (7.2.2) are still valid for computing x,y. The only difference is that the values of l_m,h_m are not compile-time constant, e.g., they may be results of some other expressions. This, in turn, will affect the computation of equations (7.2.3)-(7.2.5) for control sequence parameters p,q,r.

We propose a solution which introduces additional data flow graphs for computing (7.2.1)-(7.2.5) for the array selection operations as shown in Figure 7.18 and Figure 7.19. The subgraph RGEN plays the role of computing p,q,r from l_m,h_m, according to (7.2.1) to (7.2.5). The subgraph CGEN will generate the control sequence $F^pT^qF^r$ from p,q,r at runtime. Step 2 in the OPMAP procedure in Figure 7.13 can easily be modified to

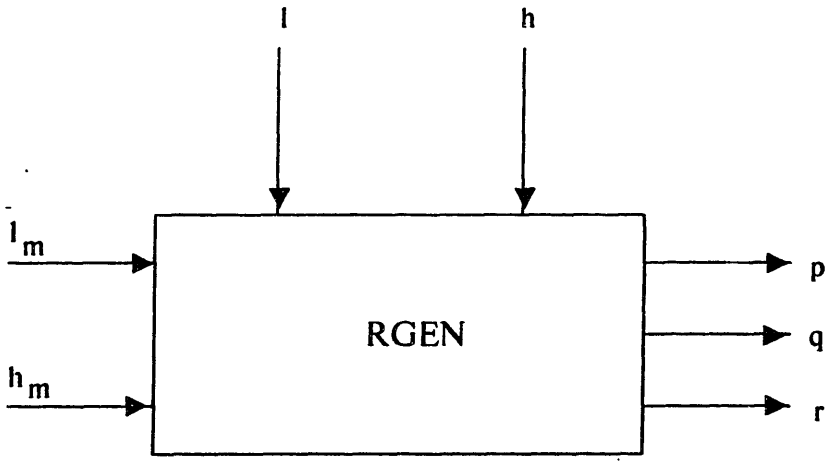


Figure 7.18. The RGEN subgraph for computing the control sequence parameters

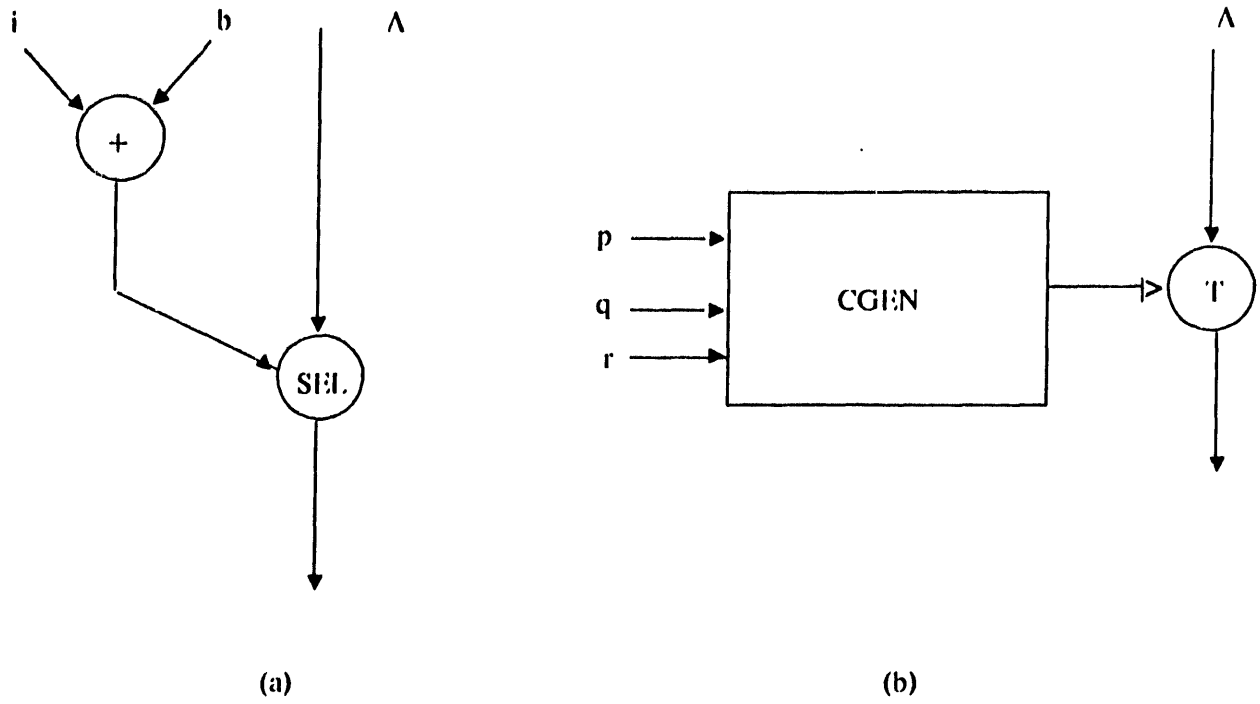


Figure 7.19. The optimization of a SEL actor using CGEN subgraph

accommodate such changes. Steps 5-6 are no longer needed because the encoded sequence is computed naturally by the IGEN and MB subgraphs. We omit the optimization procedure which can be constructed easily.

7.3 Pipelining of One-Level Primitive **forall** Expressions

Let us consider the behavior of the result data flow graph of a one-level primitive **forall** expression after an adequate optimization procedure is successfully applied. We are particularly interested in case-1 expressions where each SEL actor is simply replaced by a T-gate, and AGEN is replaced by a single arc.

Recall that the core of the graph before optimization — the graph derived from the basic mapping rule — is acyclic. The optimization procedure obviously preserves the acyclic nature of the graph. Furthermore, it eliminates AGEN and sometimes even IGEN.

Best of all, all SEL actors are removed and replaced by ordinary graph actors. Thus, the graph after optimization is the same as any acyclic data flow graph without array actors. Hence it can be maximally pipelined if appropriate balancing is performed.

8. Mapping Scheme for Multi-Level Forall Expressions

In this chapter, we extend the basic mapping rules and optimization procedures developed in the last chapter to nested class-1 **forall** expressions.

8.1 Representation of Multi-dimensional Arrays

8.1.1 Flattened Representation of Multi-dimensional Arrays

As in VAL, a multi-dimensional array in PIPVAL is conceptually a one-dimensional structure whose elements are arrays. For example, a two-dimensional array of integers is equivalent to a one-dimensional array whose elements are one-dimensional arrays of integers. We call this model for multi-dimensional arrays a *vector of vectors* model. In contrast, some languages such as Fortran use a *flat* model, where the arrays of lowest dimension are concatenated to make a one-dimensional array.

In developing the basic mapping rules for multi-dimensional arrays, the vector-of-vectors model is a better choice, because it facilitates graph construction in a recursive manner. However, in the data flow graph, the array values represented in this model are expensive to manipulate in target data flow computers.

As in the one-level case, the goal of the optimization procedure is to effectively remove the array operations and replace them with ordinary graph actors. In dealing with multi-level **forall** expressions, this requires the use of the flattened array representation. In a data flow graph, a multi-dimensional array can be flattened at any level, and each level can have a flattened representation, like that of a one-dimensional array. In our discussion, we are most interested in complete flattening, i.e., an array flattened in all dimensions, as illustrated by the following example.

Let us consider A — a two-dimensional array of integer values whose first and second dimensions have index ranges (1,m) and (1,n) respectively. Figure 8.1 (a) shows one representation of the array, where only the first dimension is flattened. That is, A is

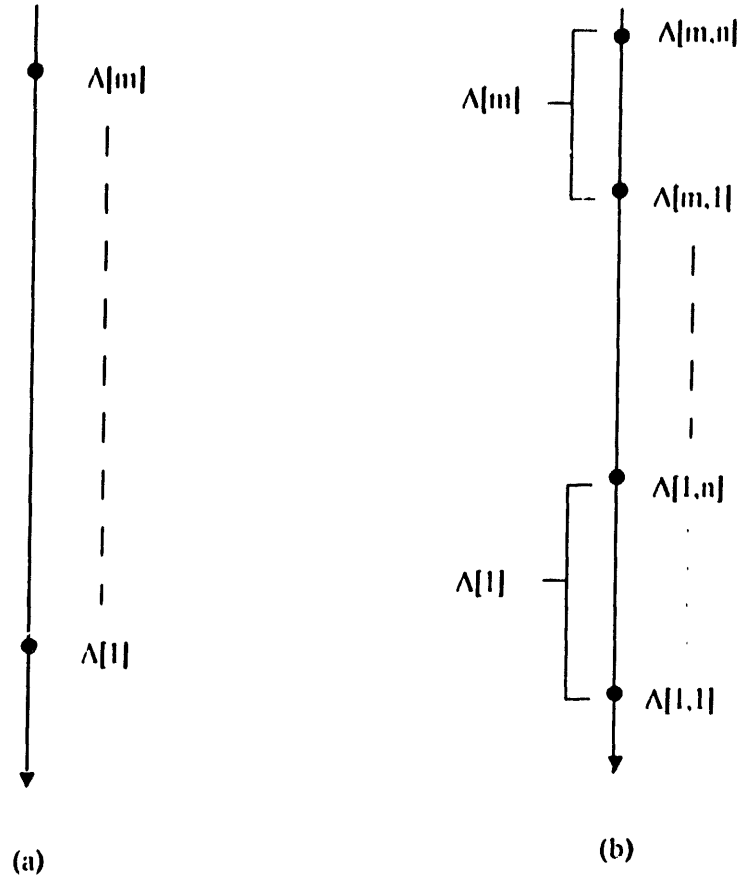


Figure 8.1. Flattened data flow representations of a two-dimensional array

represented by a sequence of m one-dimensional array values A[1]...A[m] carried by tokens on a single arc in certain order, where A[i] denotes the array value for the ith row of the array A. We can also represent the array in a complete flattened fashion as shown in Figure 8.1 (b), where A is represented as a sequence of m×n tokens arriving at one single arc in a certain order.

8.1.2 Index Vectors and Their Orders

The order of flattened representation introduced for a one-dimensional array also needs to be extended to the multi-dimensional case.

Consider a k-dimensional array A where $k \geq 1$. An element of A can be selected by a k-dimensional index vector $\mathbf{i} = (i_1,i_2...i_k)$ via an array selection operation $A[i_1,i_2...i_k]$, where

$$\mathbf{i} \in N \times N \times ... N$$

with N being the set of non-negative integers. Furthermore, there is a set of index limit constraints, one for each index, such as

$$l_j \leq i_j \leq h_j \qquad (j = 1...k)$$

where l_j and h_j correspond to the low and high limits of the jth dimension respectively.

Later we will often need to refer to the order of array elements in a flattened data flow representation. This can be defined easily in term of an ordering of index vectors. Let us define the order among index vectors.

Definition Let $\mathbf{i} = (i_1,i_2...i_k)$, $\mathbf{j} = (j_1,j_2...j_k)$ be two index vectors. We define $\mathbf{i} \mapsto \mathbf{j}$ (read as **i** less than **j**), iff there exists t ($1 \leq t \leq k$) such that $i_t < j_t$, and $i_s = j_s$ for s = 1,2...t-1.

Now consider the following function f: $V \rightarrow V$, where V is the set of k-dimensional index vectors, such that

$$f(\mathbf{i}) = (i_1+b_1,i_2+b_2...i_3+b_3) \qquad \mathbf{i} = (i_1,i_2...i_k) \qquad (8.1)$$

where each element of f(**i**) is a simple affine function of the corresponding index in $\mathbf{i} = (i_1,i_2...i_k)$. The function f is called a *simple affine* function of **i**. Now, let us prove the following theorem.

Theorem 8.1 If $\mathbf{i} = (i_1,i_2...i_k)$, $\mathbf{j} = (j_1,j_2...j_k)$ are two index vectors in V such that $\mathbf{i} \mapsto \mathbf{j}$, then the following holds: $f(\mathbf{i}) \mapsto f(\mathbf{j})$, where $f(\mathbf{i}) = (i_1+b_1,i_2+b_2...i_3+b_3)$.

Proof of Theorem 8.1.

Since $\mathbf{i} \mapsto \mathbf{j}$, there exists t ($1 \le t \le k$) such that $i_t < j_t$, $i_s = j_s$ for s = 1,2...t-1. Hence, $i_t+b_t < j_t+b_t$, and and $i_s+b_s = j_s+b_s$ for s = 1,2...t-1. As a result, $f(\mathbf{i}) \mapsto f(\mathbf{j})$ and the theorem holds. □

8.1.3 Major Orders in Flattened Data Flow Representation

When an array A is completely flattened, it is represented as a sequence of element values carried by tokens on a single arc at successive moments. Among the many possible orders of the flattened data flow representation of an array, two orders are of most interest to us: the *major normal order* and the *major reverse order.* The two orders are called *major orders.* The concept of major order has been used in the discussion of one-dimensional arrays. Now we define it more carefully.

Definition Let a_i,a_j be two elements of an array A with index vectors i,j respectively. An order of A (on an arc) is called a major normal order if $a_i \mapsto_p a_j$ implies $\mathbf{i} \mapsto \mathbf{j}$, and vice versa. Similarly, an order of A is called a major reverse order iff $a_i \mapsto_p a_j$ implies $\mathbf{j} \mapsto \mathbf{i}$, and vice versa.

When an array A has a major order on an arc, we also say it is represented in a major order on that arc. For example, the representation of the array A in Figure 8.1 (b) is in a major normal order on the arc. For convenience, we say an array is generated in a major order by the graph of a code block (or simply, by a code block), if it is represented in a major order on the output arc of the graph. In this case, we also say that the *generation order* of A (by the code block) is a major order.

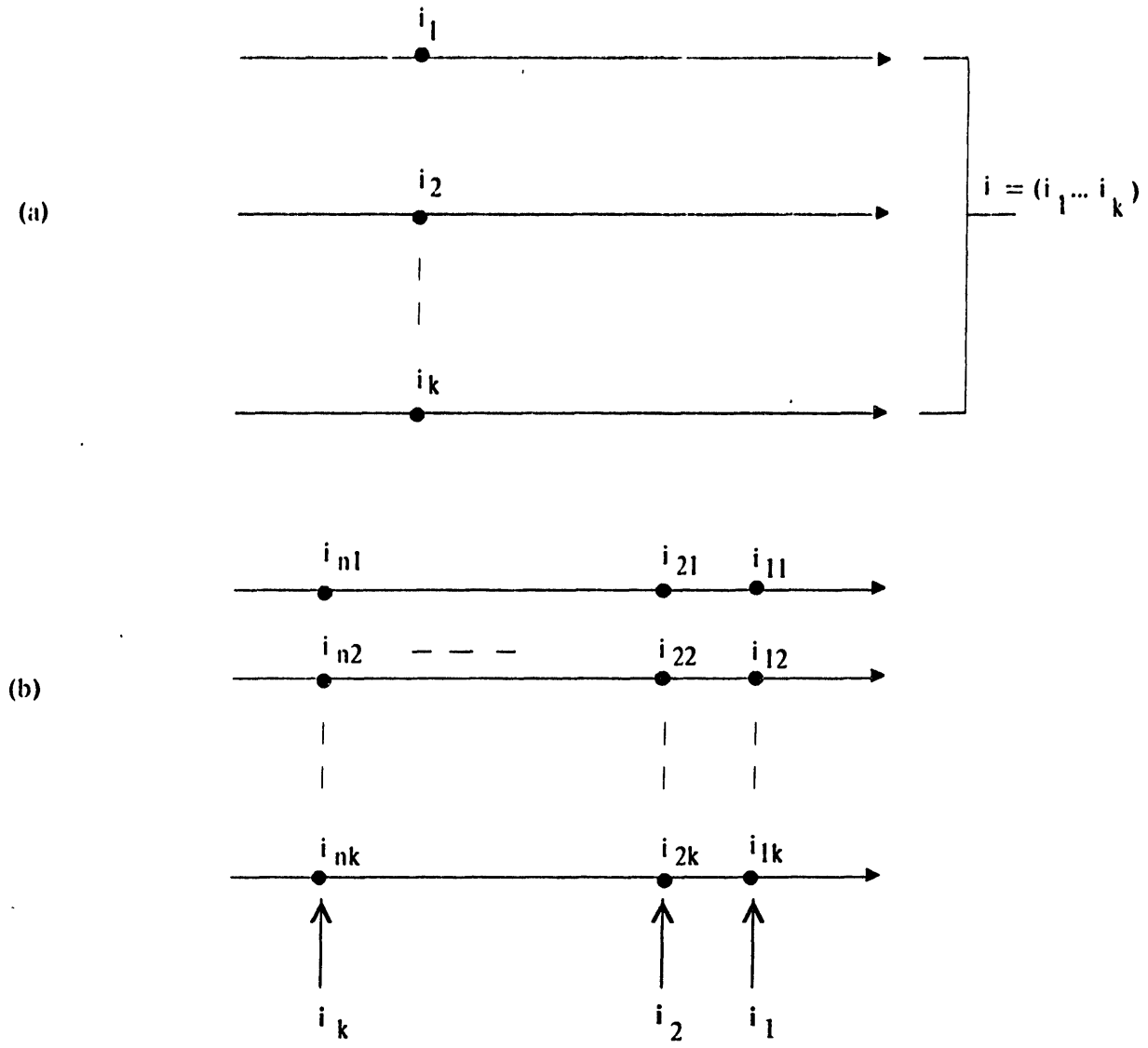


Figure 8.2. A pipelined representation of a sequence of index vectors

The concept of generation order can be applied naturally to index vectors. Let us represent an index vector $i = (i_1,i_2...i_k)$ by k tokens carrying values $i_1...i_k$ respectively, conveyed on a group of k arcs as shown in Figure 8.2(a). Then a sequence of n k-dimensional index vectors $i_1 = (i_{11},i_{12}...i_{1k})$, $i_2 = (i_{21},i_{22}...i_{2k})$, ... , $i_n = (i_{n1},i_{n2}...i_{nk})$ can be represented as shown in Figure 8.2(b). Now we can extend the concept of pipelined

order to the sequence of index vectors such that $\mathbf{i}_1 \mapsto_p \mathbf{i}_2$ iff $\mathbf{i}_{1j} \mapsto_p \mathbf{i}_{2j}$ for $j = 1...k$.. Thus, a sequence of index vectors is said to be generated by a graph in a major order if it is represented in a major order on the corresponding output arcs of the graph.

Let F denote a data flow graph which computes a simple affine function f in (8.1) as shown in Figure 8.3. Assume a sequence of index vectors is presented at the input ports of F. F will generate a sequence of index vectors at its output ports. It is easy to see that F preserves the pipelined order between the two sequences. Let $\mathbf{i}_1$, $\mathbf{i}_2$ be two index vectors at the input and $\mathbf{i}_1 \mapsto_p \mathbf{i}_2$. Their corresponding output index vectors $\mathbf{j}_1$, $\mathbf{j}_2$ must satisfy $\mathbf{j}_1 \mapsto_p \mathbf{j}_2$. If the input vectors are represented in major normal order, i.e., we have $\mathbf{i}_1 \mapsto \mathbf{i}_2$, then we must also have $\mathbf{j}_1 \mapsto \mathbf{j}_2$ since $\mathbf{j}_1 = f(\mathbf{i}_1)$, $\mathbf{j}_2 = f(\mathbf{i}_2)$ and the function F is a simple affine function. As a result the sequence of output index vectors will also be in major normal order.

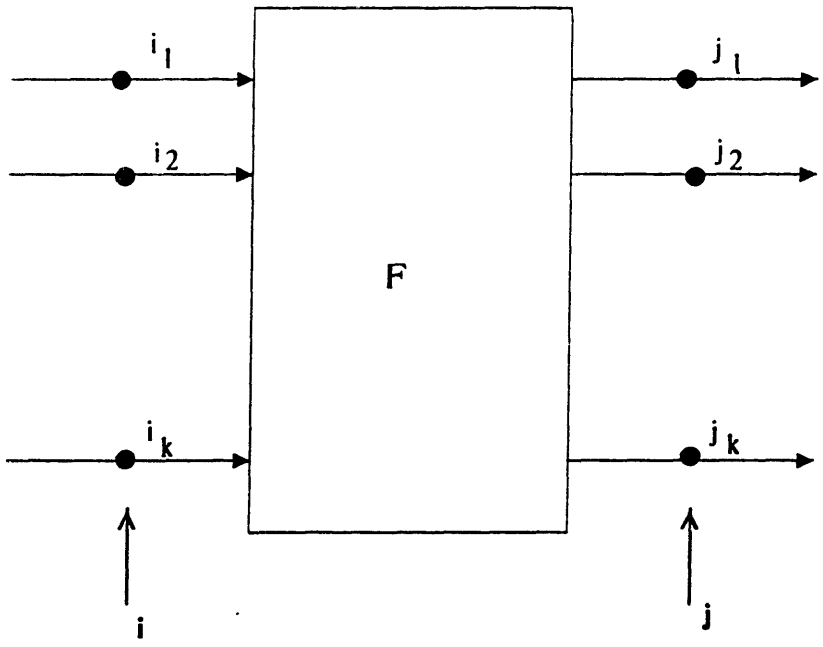


Figure 8.3. Orders between input and outputs of an affine function

8.2 Mapping of Two-Level Forall Expressions

Let us start with the mapping of two-level **forall** expressions to illustrate how the basic mapping rule for a one-level **forall** expression can be extended to a multi-level **forall** expression.

Consider an example of a two-level **forall** expression, shown in Figure 8.4, which is known as the (two-dimensional) *model problem* in PDE applications. (This example is also used in Chapter 4, we include a copy here for the readers convenience.) This code block takes a two-dimensional input array A and constructs another two-dimensional array X. It can be considered as a one-level **forall** expression the body of which consists of another **forall** expression. As indicated at the end of Chapter 7.1, the mapping rule for one-level **forall** expressions can be applied to such a two-level nested **forall** expression, simply by recursively applying the mapping rule to its body. For example, Figure 8.5 shows the result

```
X =
    forall i in [0,m+1]
    construct
        if i = 0 then A[i]
        elseif i = m+1 then A[i]
        else
            forall j in [0,n+1]
            construct
                if j = 0 then A[i,j]
                if j = n+1 then A[i,j]
                else
                    (A[i,j-1] + A[i,j+1]
                     + A[i-1,j] + A[i+1,j])/4
                endif
            endall
        endif
    endall
```

Figure 8.4. The model problem: version 1

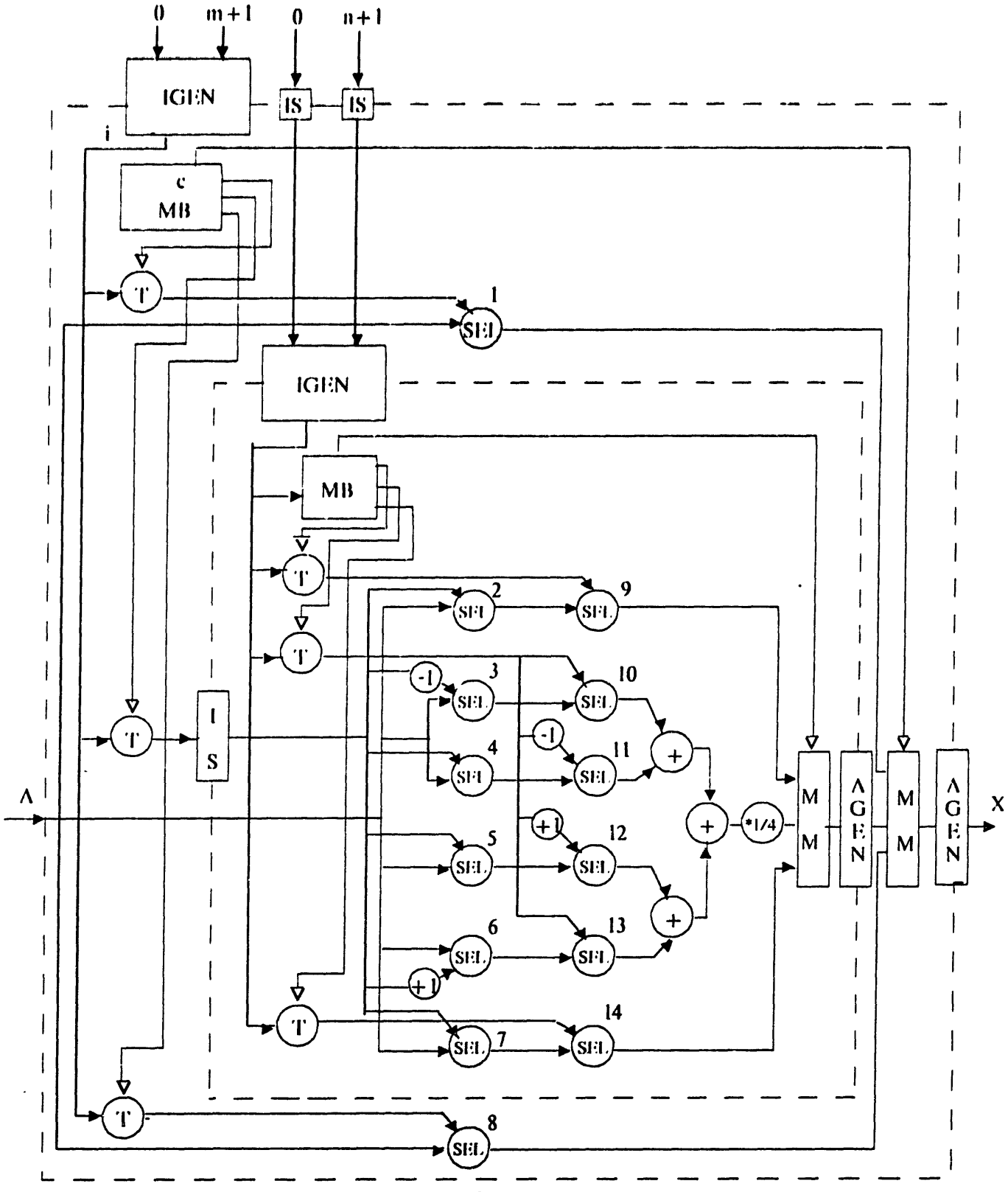


Figure 8.5. The SDFGL graph of mapping a two-level forall expression

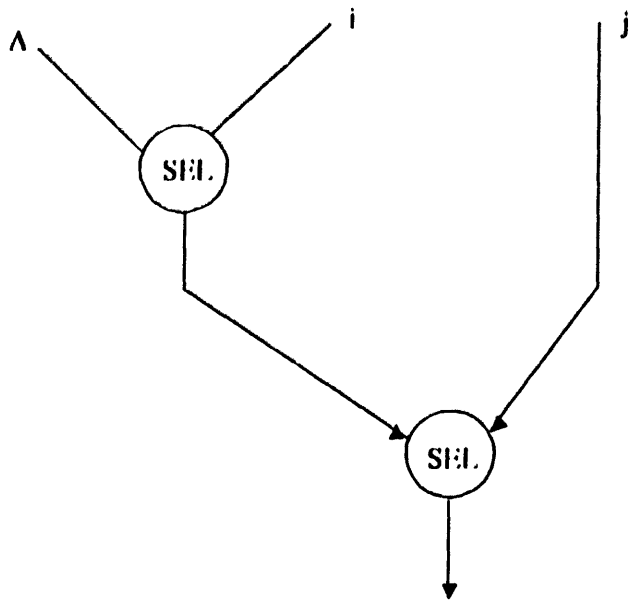


Figure 8.6. The data flow graph for mapping A[i,j]

of mapping the expression in Figure 8.4, using the mapping rule under the mapping condition that the result array is to be generated in major normal order. Note how the nested structure of the **forall** expression is reflected in the result graph. The subgraph inside the inner dashed line box corresponds to the inner **forall** expression It constructs a sequence of m one-dimensional arrays: row 1 through row m of the result array X.

Now let us study how the two-dimensional array selection operations in the body are handled in the mapping. As stated earlier, an array selection operation in the form A[i,j] is equivalent to A[i][j] where A[i] can itself be considered an array, i.e., the ith row of A. The mapping rule for a one-dimensional array selection operation can be extended directly to the two-dimensional case. Figure 8.6 shows the result of recursive application of the basic mapping rule in section 6.5 to A[i,j]: two SEL actors in series correspond to the selection operations by index i and j respectively. One can easily understand the above result by considering A[i][j] as SELECT(SELECT(A,i),j). Thus, the mapping rule associated with binary operators can be recursively applied to derive the graph shown in Figure 8.6.

It is interesting to note the different representations of the result array at different stages in the graph. At the output port of the inner MM actor, the array is represented by a sequence of (m+2) x (n+2) element values carried by tokens on the arc directed to the inner AGEN at successive moments. That is, the array is completely flattened and generated in major normal order. The rule of the inner AGEN is to "pack" the sequence of elements values into m+2 one-dimensional arrays, each corresponding to one row of the array X. Thus, at the output port of the outer MM, the array is represented as a sequence of (m+2) tokens, each carrying a one-dimensional array value on the arc directed to the input of the outer AGEN. Similarly, the outer AGEN will assemble the m+2 one-dimensional array values into the result array represented by one token carrying a two-dimensional array value X on the arc from its output port.

8.3 Optimization of Two-level Primitive Forall Expressions

The data flow graph derived from the basic mapping rule for a two-level **forall** expression may contain certain array actors. The overhead may be even more significant than in the one-level case, because it often involves nested AGENs and SEL actors which must be able to generate and transmit sequences of array values, and store them in memory when necessary. As before, the goal of optimization is to remove the AGENs and the SEL actors and replace them with ordinary graph actors. Thus, the result data flow graph may show much improved performance in terms of pipelining. The optimization procedure to be presented is based on the one-level case developed in Section 8.2.

8.3.1 Consistent Array Selection Orders

In this section, we extend the concept of the selection order of an array selection operation — the order in which the elements of the array are used — to the multi-dimensional case.

Let us consider an array selection operation $A[i+b_1, j+b_2]$ in a two-level primitive

```
forall i in [0,m+1]
construct
        .
        .
    forall j in [0,n+1]
            .
            .
        A[i+b1,j+b2]
            .
            .
    endall
        .
        .
endall
```

Figure 8.7. An array slection operation in a two-level primitive forall expression

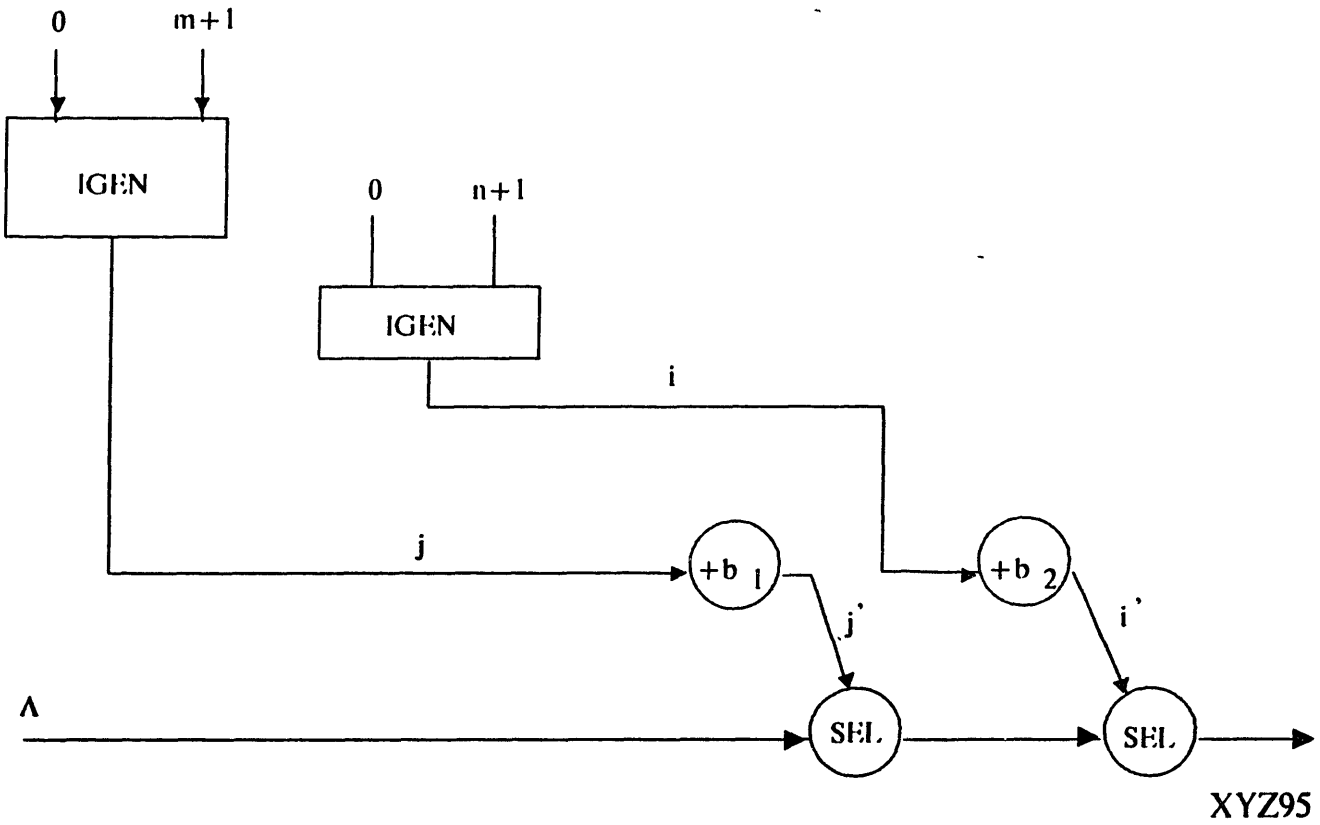


Figure 8.8. The selection order of an array selection operation

forall expression as shown in Figure 8.7. The data flow graph is generated by the basic mapping rule as shown in Figure 8.8, where $A[i+b_1, j+b_2]$ is mapped into two SEL actors in series. The selection order of $A[i+b_1, j+b_2]$ is the same as the pipelined generation order of its index vector $(i+b_1, j+b_2)$, which is represented by a sequence of pairs of tokens on the two arcs labeled i',j'. This order — the *selection order* of A[i+b1, j+b2] — is the same as the generation order of index vector (i,j), represented by the sequence of pairs of tokens on the two arcs labeled i,j (see argument at the end of Section 8.1).

An important condition of optimization of A[i+b1, j+b2] is that its selection order should match the generation order of A. This also implies that if more than one selection operation of A exists, they all should have the same selection order. If this requirement is met, we say the array A has a *consistent selection order* in the code block (or its graph); or equivalently, we say the code block (or its graph) has a consistent selection order with respect to A.

Thus, if the body in the above example contains only array selection operations in the form of $A[i+b_1, j+b_2]$, the **forall** expression has a consistent selection order with respect to array A. This condition is very important to the application of the optimization procedure to be developed next. Therefore, it is included in the optimization procedure as a key attribute associated with a **forall** block.

As a remark, there are cases where a code block may not have a consistent selection order with respect to its input arrays. For example, in Figure 8.7, if the body of the inner **forall** expression contains both $A[i+b_1, j+b_2]$ and $A[j+b_3, i+b_4]$, then the **forall** expression does not have a consistent order with respect to A.

8.3.2 An Example

Let us briefly study optimization of the data flow graph of the two-level **forall** example derived from the basic mapping rule as illustrated in Figure 8.5. There are 14 SEL actors which can be divided into two groups. The first group, called *level-1* SEL actors,

consists of eight SEL actors on the left (labeled 1-8); the second group, called level-2 SEL actors, consists of 6 SEL actors on the right (labeled 9-14). The selection indices of the level-1 and level-2 SEL actors correspond to the indices of the level-1 and level-2 **forall** expressions, e.g., i and j, respectively.

Now consider the optimization of the SEL actors. Note that the **forall** expression has a consistent selection order with respect to A. We assume that array A is generated in major normal order. Furthermore, we assume that X is also to be consumed in major normal order. The optimization includes the removal of SEL actors, as well as AGENs. The principle of optimization is the same as that for the one-level case outlined in the last chapter. Since the selection order of all array selection operations of the input array A is the same as its generation order, each SEL actor can be replaced by an ordinary graph actor such as a properly controlled T-gate actor. The key is that the sequence of element values of A used in the computation must be passed in order, while the unused elements should be discarded without causing jams.

As before, we need to derive the set of selection index ranges (for all SEL actors) and the corresponding control value sequences (boolean pipelines). In the example under discussion, the set of ranges are compile-time computable, using the same principles found in (7.2.1) to (7.2.5). The table in Figure 8.9 lists these parameters for the eight level-1 SEL actors. Note that each value in the selection range selects one row of the array. Therefore, $d = n+2$ control values are needed to select or discard values in one flattened row. The selection ranges and control sequences of the six level-2 SEL actors are listed in Figure 8.10. They are similar to those found in a one-dimensional array.

Therefore, an optimization similar to that for the one-level case can be performed. The result graph of the optimization for our current example is shown in Figure 8.11. All AGEN and SEL actors have disappeared from the result graph. Note the FIFO sizes for the level-1 SEL actors. Recall that the FIFOs are introduced together with certain T-gates to hold skewed values and prevent jams. A skew of 1 of index i results in one row of

SEL.	index expression	index range	selection range	p	q	r	control value sequence
1	i	[0,0]	[0,0]	0	1	m+1	$C1 : T^{d} F^{(m+1)d}$
2	i	[1,m]	[1,m]	1	m	1	$C2 : F^{d} T^{md} F^{d}$
3	i-1		[0, m-1]	0	m	2	$C3 : T^{md} F^{2d}$
4	i		[1,m]	1	m	1	$C4 : F^{d} T^{md} F^{d}$
5	i		[1,m]	1	m	1	$C5 : F^{d} T^{md} F^{d}$
6	i+1		[2, m+1]	2	m	0	$C6 : F^{2d} T^{md}$
7	i		[1,m]	1	m	1	$C7 : F^{d} T^{md} F^{d}$
8	i	[m+1, m+1]	[m+1, m+1]	m+1	1	0	$C8 : F^{(m+1)d} T^{d}$

Figure 8.9. The selection index ranges and control sequences -- level-1 SEL actors

excessive array element values, which should be held by the FIFO. Thus the size of a FIFO associated with a level-1 SEL actor is in the unit of d = n-2. For example, if we use A[i+1] as a reference, the skew of i for A[i-1] and A[i] are 2 and 1 respectively. Hence FIFOs of size 4d and 2d are introduced respectively.

In the next section, we present the extended optimization procedure. Using this procedure, we will be able to derive the simple result graph in Figure 8.11.

SEL.	index expression	index range	selection range	p	q	r	control value sequence
9	j	[0, 0]	[0, 0]	0	1	n+1	$C9 : TF^{n+1}$
10	j	[1, n]	[1, N]	1	n	1	$C10 : FT^{n}F$
11	j-1		[0, n-1]	0	n	2	$C11 : T^{n}F^{2}$
12	j+1		[2, n+1]	2	n	0	$C12 : F^{2}T^{n}$
13	j		[1, n]	1	n	1	$C13 : FT^{n}F$
14	j	[n+1, n+1]	[n+1, n+1]	n+1	1	0	$C14 : F^{n+1}T$

Figure 8.10. The selection index ranges and control sequence -- level-2 SEL actors

8.3.3 Optimization Of Array Selection Operations with Compile-time Computable Selection Index Ranges

Let us consider an input array A to a **forall** code block which has a consistent selection order with respect to A. Without loss of generality, A is assumed to have the same index ranges as the index range of the code block. In this section, we look at the case in which the selection index range for each array selection operation $A[i+b_1, j+b_2]$ is compile-time computable.

Assume the code block to be handled is the two-level **forall** expression shown in Figure 8.12. The top level structure of the body expression consists of a range-partitioning conditional expression with respect to the index i. It is similar to that in Figure 7.8 (see Chapter 7), except that each exp_s (s = 1...k) may now consist of a one-level **forall** expression. All the subrange limits should be known at compile-time in order to compute

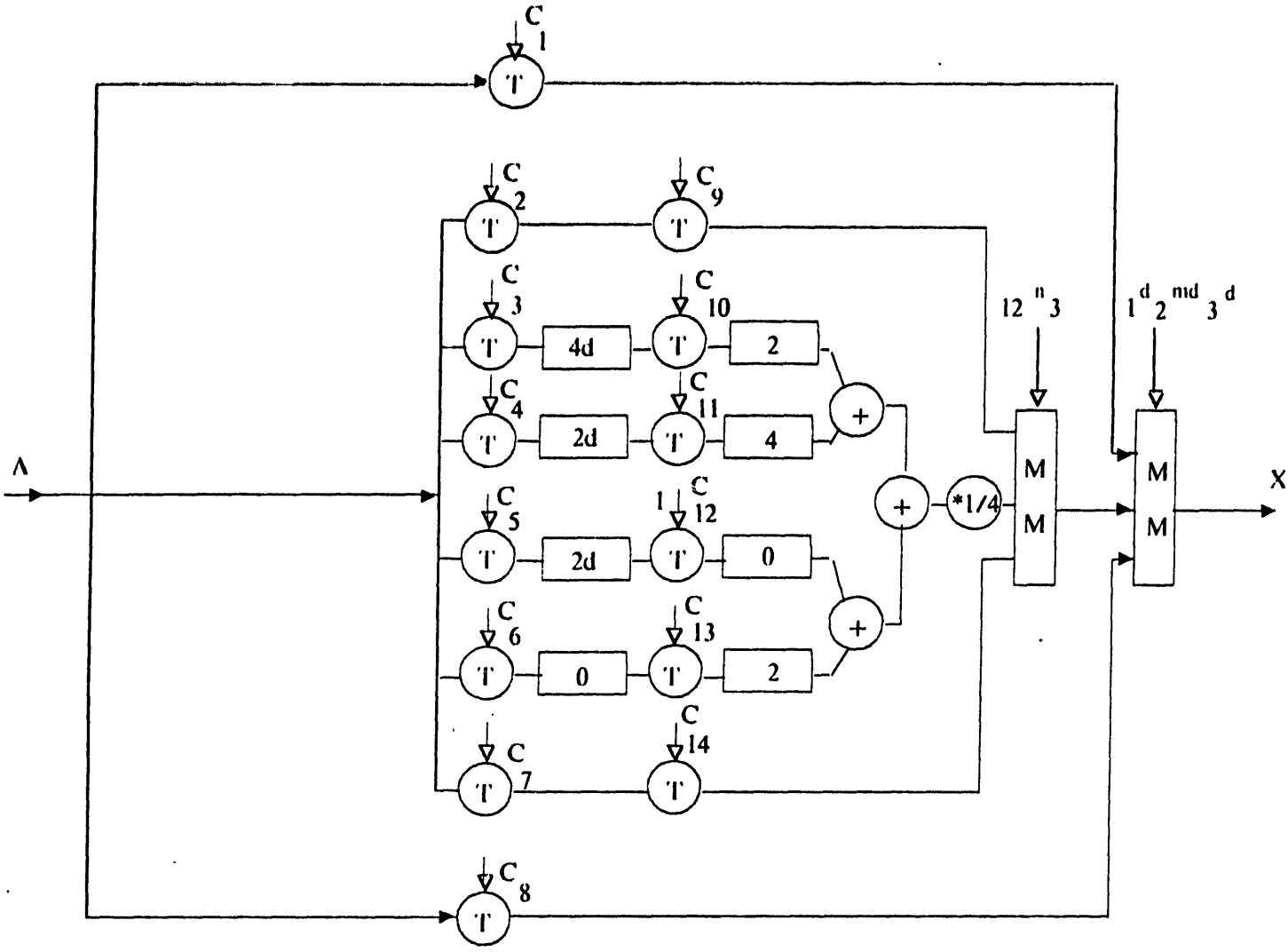


Figure 8.11. Result of Removing Array Actors in Figure 8.5

the selection index subranges for all level-1 SEL actors. A similar condition is needed for the level-2 **forall** expressions. Thus, this case is a two-level version of the case-1 **forall** expressions introduced in Chapter 7.

Case-1 (two-level): A two-level primitive **forall** expression where the range-partitioning expression is in its standard form (see Figure 8.12). Without loss of generality, it is assumed to have only one input array: A. The

```
X =
        forall i in [l,h]
        construct
                if l1 <= i <= h1 then exp1
                elseif l2 <= i <= h2 then exp2
                        .
                        .
                elseif lk-1 <= i <= hk-1 then expk-1
                else
                        expk
                endif
        endall
```

Figure 8.12. Standard case 1 form

bounds of all subranges are compile-time constant. In addition, the range exp_1 - exp_k does not contain any conditional expressions whose predicate expression depends on i and has A accessed in any of its subexpressions. If an exp_s (s = 1...k) consists of a one-level **forall** expression, it must be in one-level case-1 form.

Now let us consider how to extend the optimization procedure to the two-level case-1 **forall** expression. Recall that an array selection operation $A[i+b_1, j+b_2]$ is translated by the basic mapping rule into a series of two SEL actors as shown in Figure 8.6. As before, the SEL actors are replaced by T-gates controlled by proper control value sequences. The key is to compute the selection ranges of i and j for $A[i+b_1, j+b_2]$. This can be performed by using the same equations (7.2.1) - (7.2.5). When constructing the control value sequences, each value of i in its selection range corresponds to one row of A. Thus when performing the optimization for a level-1 SEL actor, both the control sequence and the size of the FIFO should be weighted by d, where d is the size of one row of A (i.e., d = l-h+1), as shown in Figure 8.13 and Figure 8.14. The handling of level-2 SEL actors is the same as

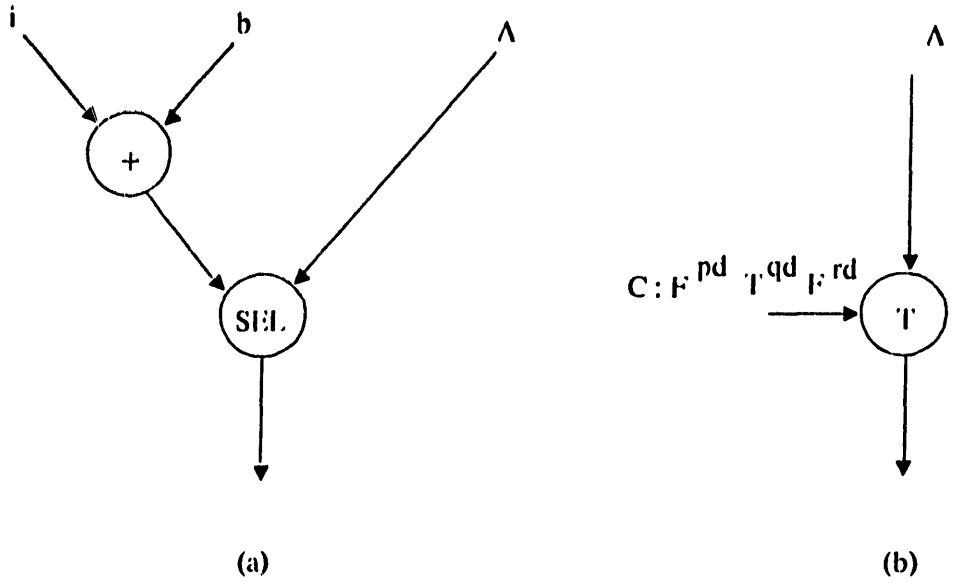


Figure 8.13. Optimization of a Level-1 SEL actor

that described for one-dimensional cases, except that the selection range for $j + b_2$ should be computed in terms of its corresponding level-2 **forall** expressions.

Thus, the optimization procedure for a one-level primitive **forall** expression can be extended to a two-level primitive **forall** expression. Such extension is straightforward and is shown in Figure 8.15.

Let us apply this optimization procedure to the **forall** expression in Figure 8.5. After step 2, the graph is transformed into the form shown in Figure 8.16, where all array actors are removed and replaced by T-gate actors. The selection index range and control value sequences for each level-1 and level-2 SEL actor in the graph (listed in Figure 8.9 and Figure 8.9) are computed by Step 2. Step 3 adjusts the size of the FIFOs, as shown in the result graph in Figure 8.17. Step 5 replaces the control link from MB to MM by a proper encoding control value sequence and removes MB, as shown in Figure 8.18. Step 6 eliminates IGEN. The final result is shown in Figure 8.19. This graph is the same as the

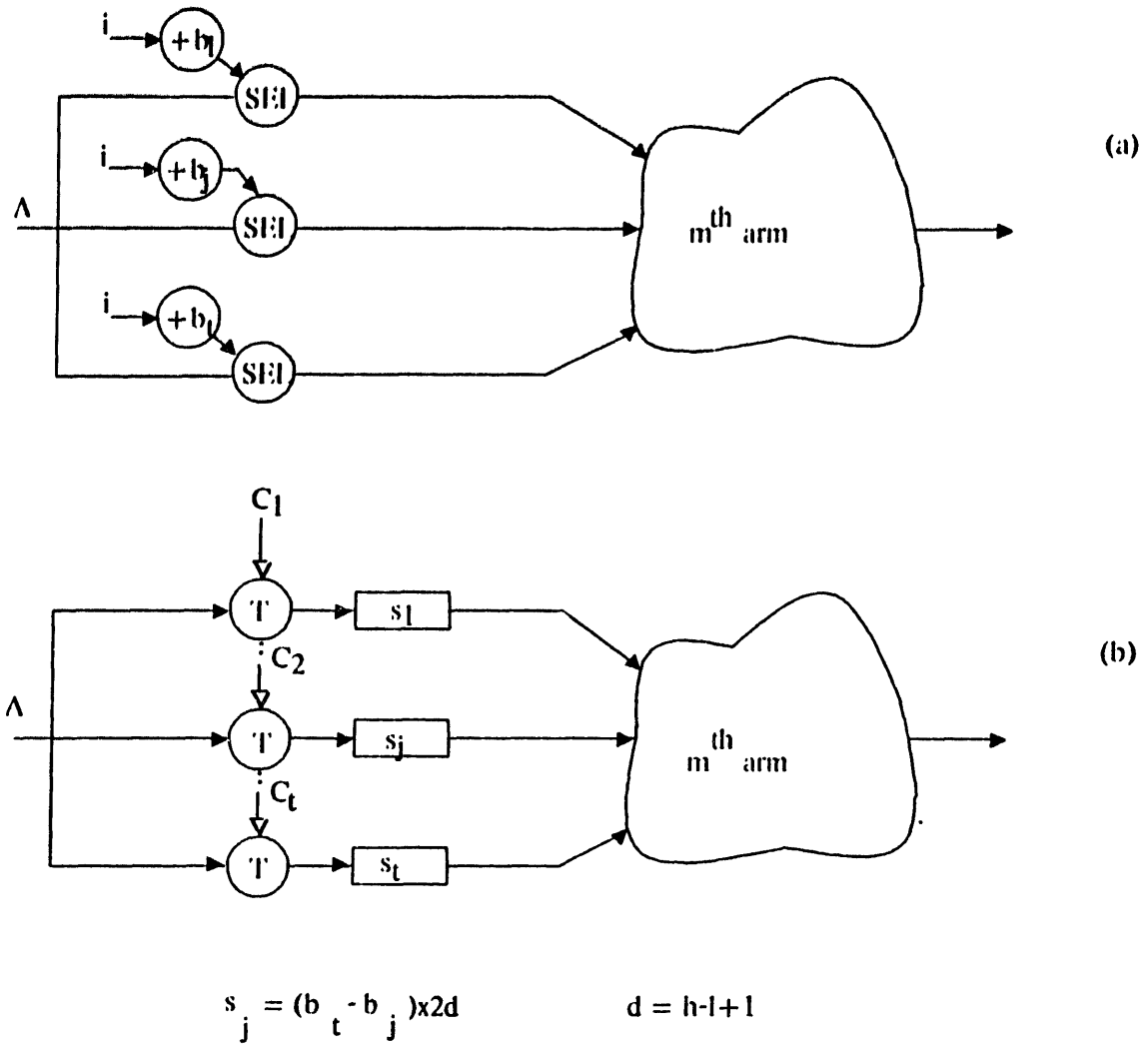


Figure 8.14. Skews in the optimization with weighted buffer size

graph in Figure 8.11.[1]

As in the one-level cases, there are situations in which a two-level **forall** expression is not initially in the standard form of Case 1, but can be transformed into that form. We assume such transformation is done before the optimization procedure is performed.

When the selection index range is not compile-time computable, the optimization

---

1. When no confusion may occur, we omit some arrows in Figure 8.16 - Figure 8.19 for simplicity.

Procedure OPMAP

Inputs: a data flow graph G derived by the application of the basic mapping rule to a two-level primitive forall expression E in case-1 standard form.

Outputs: a data flow graph G' with array operations in G optimized

Conditions: (1) G is mapped under the condition that the result array is to be generated in major normal order, (2) each input array has a major normal generation order, (3) all array selection operations of A have a consistent selection order, and it is the same as the generation order.

Method: Assume E is written in the form as Figure 8.12. Remove AGEN in G properly. Let S be the set of input arrays in E. For each A in S, compute the selection index ranges for all its array selection operations. Replace the corresponding array actors in G with properly controlled T-gate actors.

The Algorithm:

Step 0: Replace all (level-1, level-2) AGEN subgraphs as shown in Figure 7.12.

Step 1: If $S = \emptyset$ then go to step 5.

Step 2: Let $A \in S$. For each array selection operation on A compute the values p,q,r according to (7.2.1) to (7.2.5). Find the level-1 SEL actors and their index calculation subgraph for each $A[i+b_1]$, and replace them by a T-gate actor controlled by C: $F^{pd}T^{qd}F^{rd}$ with corresponding p,q,r as illustrated by Figure (a) and (b), where d = (h-l+1). Also find the level-2 SEL actors and their index calculation subgraphs for each $A[j+b_2]$, and replace them by a T-gate actor controlled by C: $F^pT^qF^r$ with corresponding p,q,r as illustrated by Figure (a) and (b) (where i is replaced by j).

Step 3: For each arm of the range-partitioning conditional expression, perform the buffer size adjustment on all level-1 T-gates introduced in Step 3 as shown in Figure 8.14, where d = (h-l+1). Perform a similar transformation for all level-2 T-gates but note that d = 1.

Step 4: S := S -{A}, goto Step 1.

Step 5: For each range-partitioning expression of the forall expressions in both levels, remove the arc connecting MB to the control input of MM, and provide an encoding value sequence as shown in Figure 7.11. (Note that the sequence for level-1 forall should be weighted by d.) Remove MB if it is not used.

Step 6: Remove IGEN, if it is not used.

Step 7: Stop.

Figure 8.15. The optimization procedure for two-level case-1 forall expressions

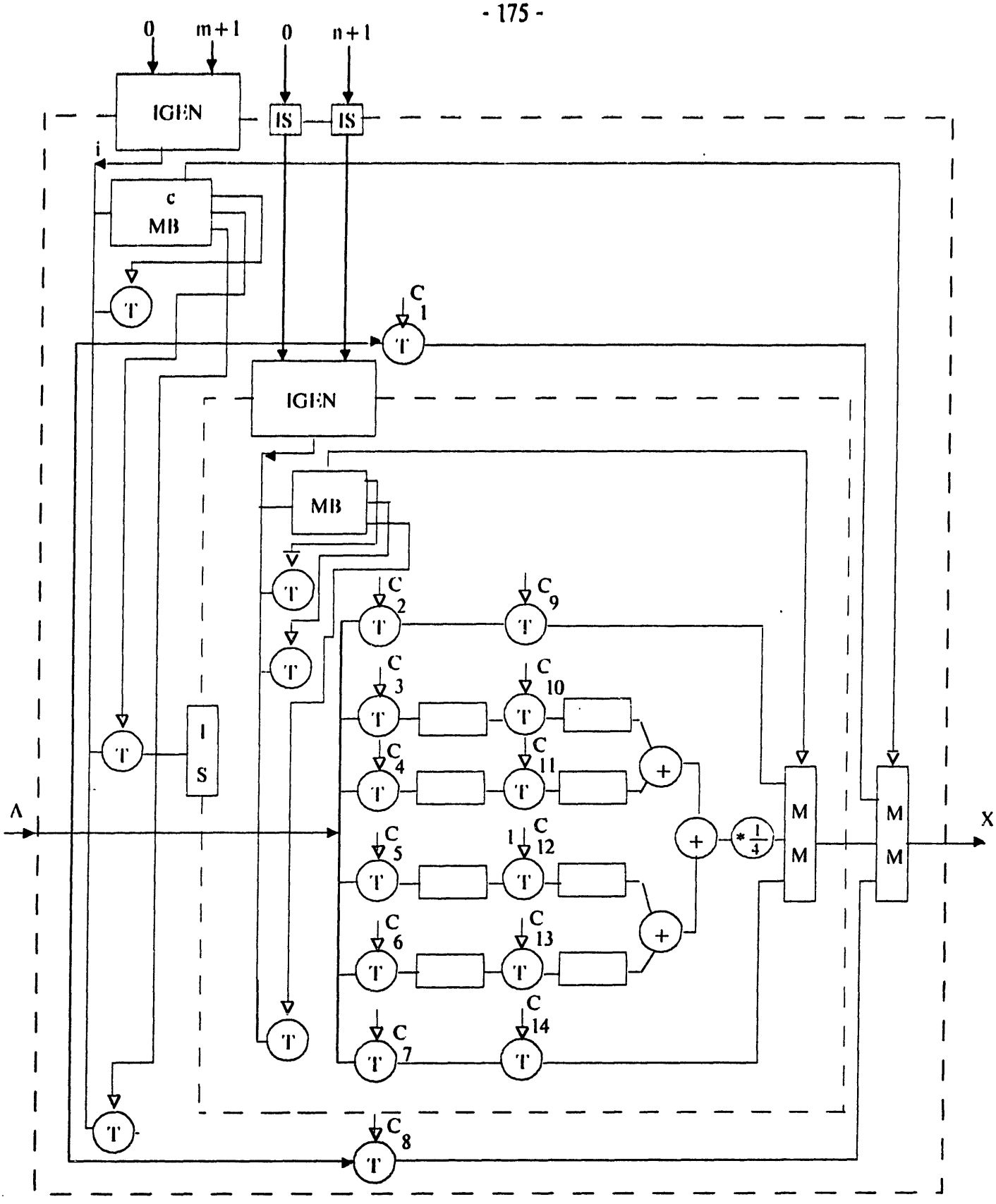


Figure 8.16. A two-level forall optimization example -- after Step 2

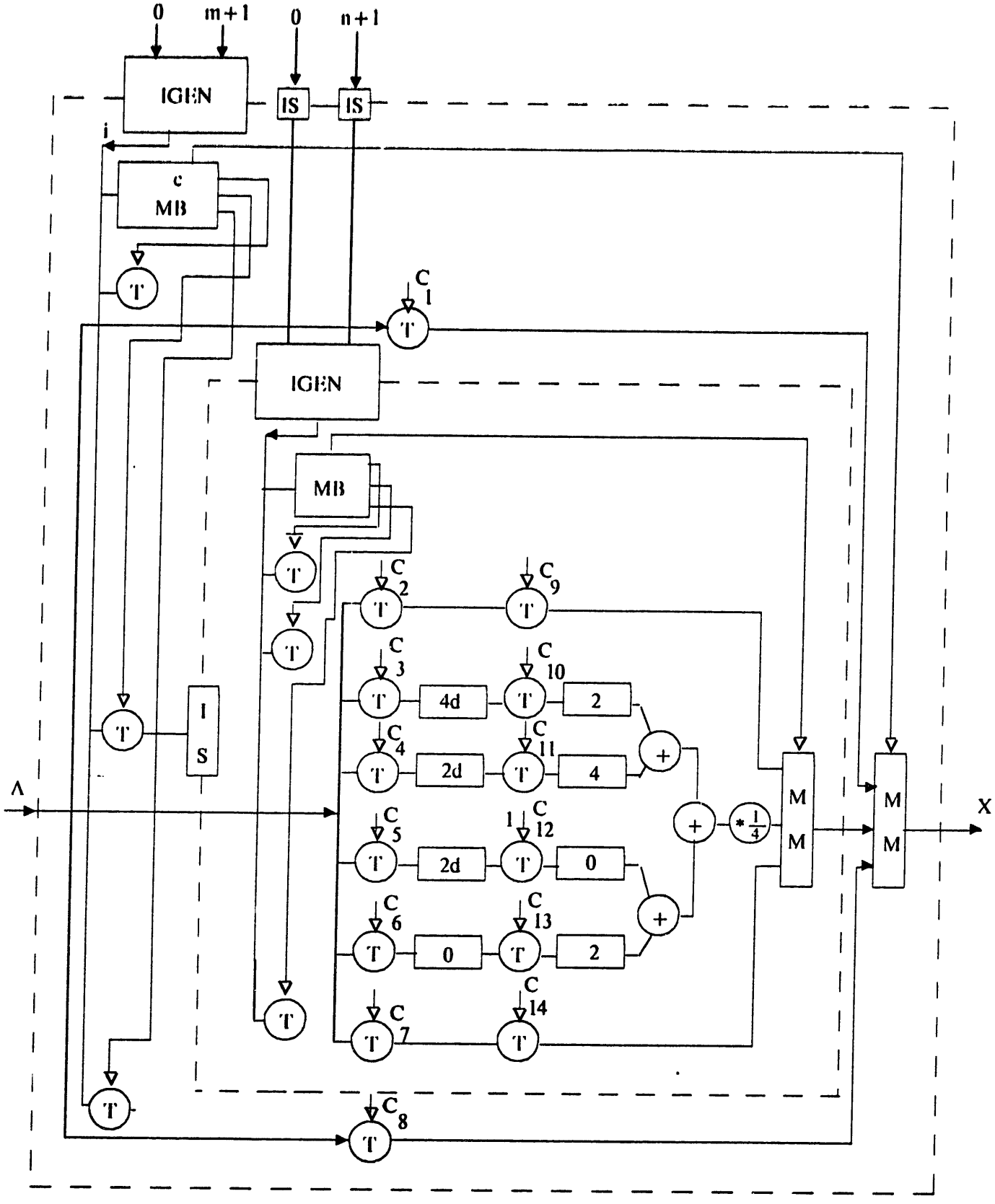


Figure 8.17. Continued from Figure 8.16 -- after step 3

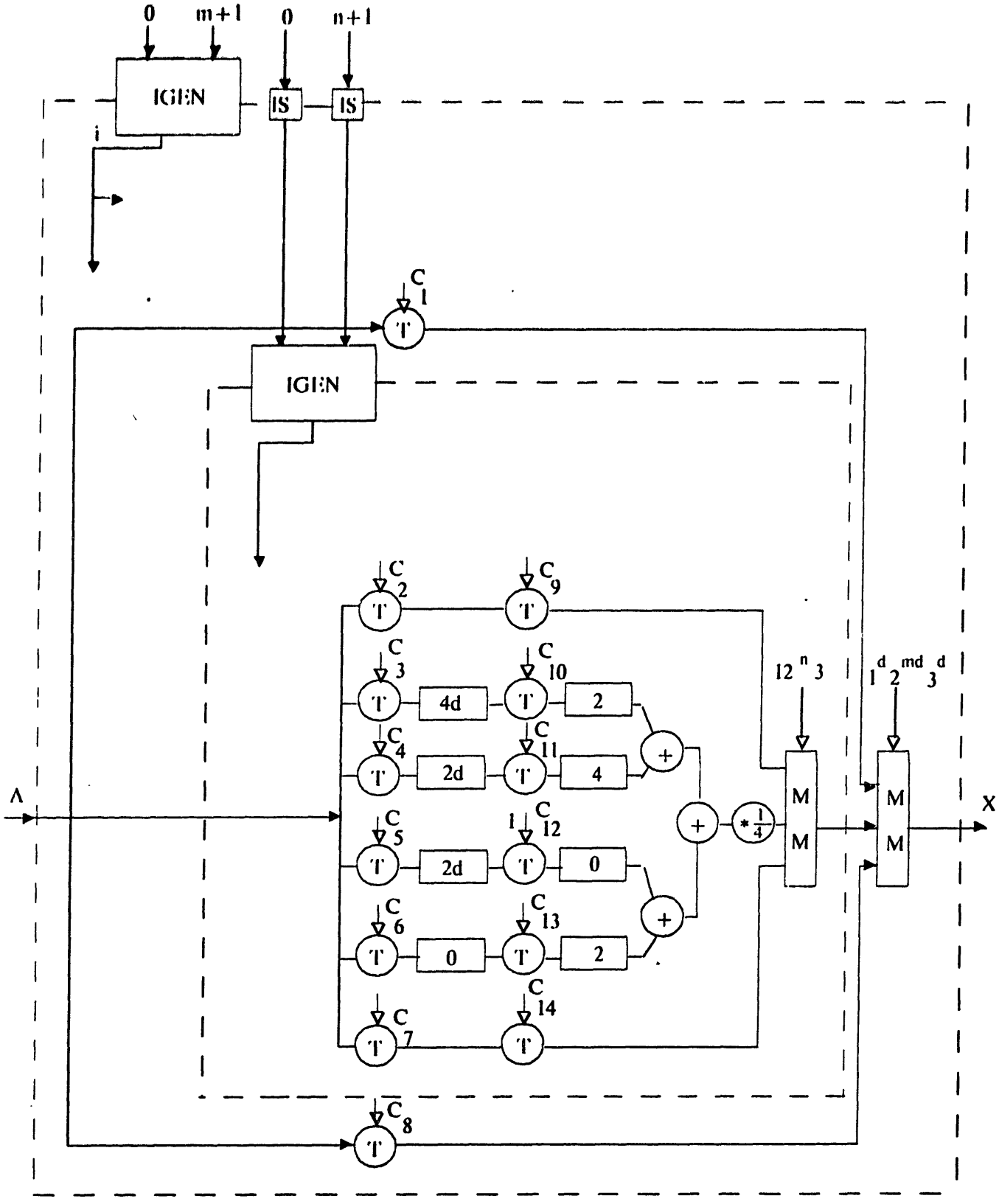


Figure 8.18. Continued from Figure 8.17 -- after step 5

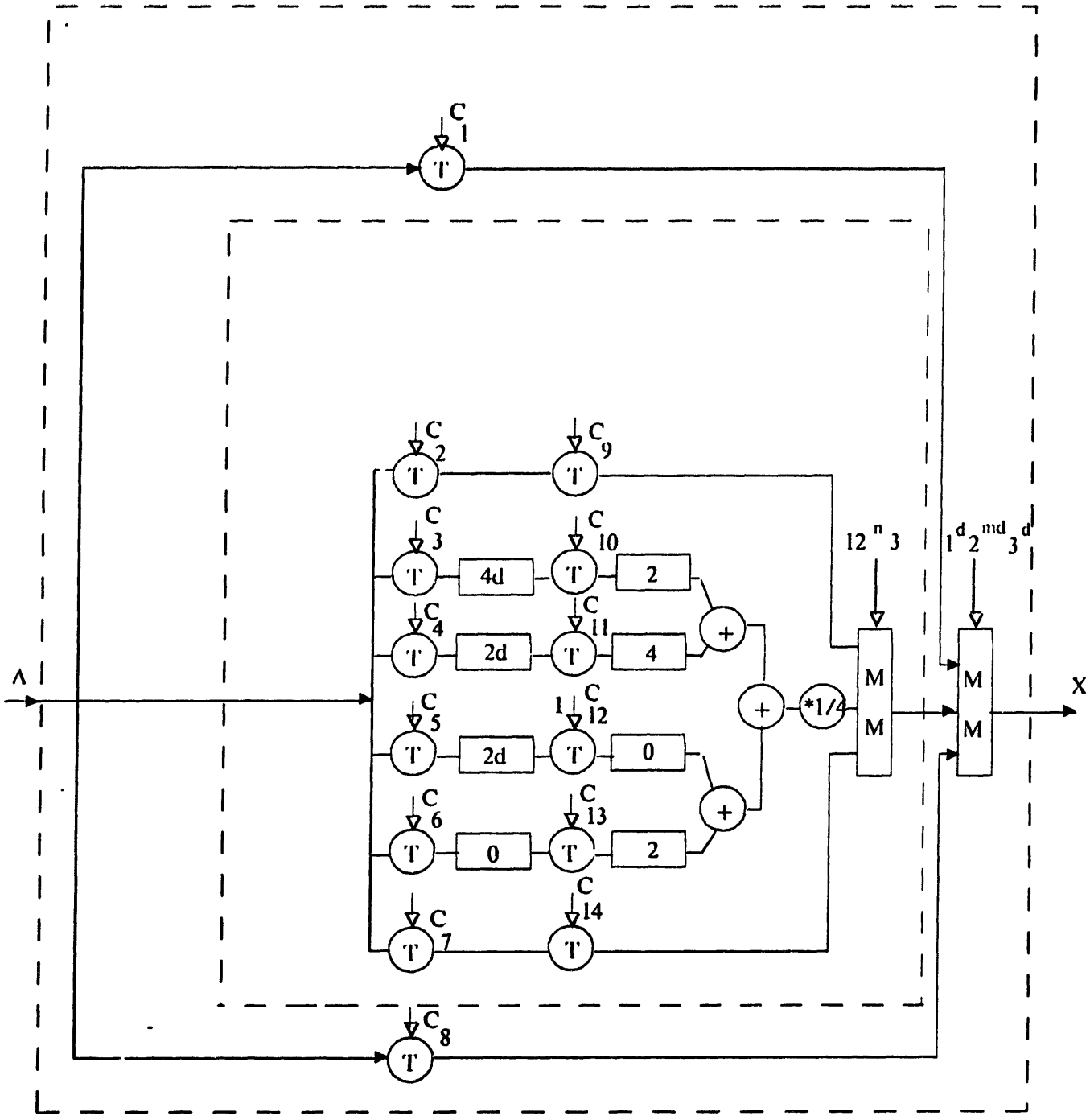


Figure 8.19. Continued from Figure 8.18 -- final result

procedure is not directly applicable. However, it is often possible to introduce extra code to generate the control value sequences at runtime, just as discussed for the one-level case in Section 7.2.3.

8.4 Multi-level Forall Expressions

The mapping of a multi-level **forall** expression can be performed using the same principle as that for a 2-level **forall** expression. That is, the graph of a k-level **forall** expression can be recursively constructed from that of a (k-1) level **forall** expression, etc. Therefore, the mapping of a multi-level **forall** expression is based on the mapping rule of a one-level **forall** expression. As for the two-dimensional case, the mapping rule for an array selection operation of a multi-dimensional array $A[i_1, i_2,...i_k]$ can be derived by extending that for a one-dimensional array.

The optimization procedure for a multi-level **forall** expression can also be constructed by applying the principles discussed for handling a 2-level **forall** expression (which in turn is based on the optimization procedure of one-level **forall** expressions). Note that the concept of consistency of selection orders with respect to a multi-dimensional array can be directly extended from that for a two-dimensional array. Again, we are most interested in cases where the selection index ranges are compile-time computable. We omit the details of such extensions.

9. The Mapping Scheme for For-construct Expressions

In this chapter, we present the mapping scheme for another important array creation construct in PIPVAL — the **for-construct** expression. One important aspect is to illustrate how our basic mapping rule of the **for-construct** expressions is derived from simplification of the more general mapping algorithm for iteration expressions [19]. In fact, we have already seen the pipelined mapping scheme of **forall** expressions treated as a special case of the **for-construct** expressions. In mapping **for-construct** expressions, we need to properly introduce and handle feedback paths in the result data flow graph.

9.1 The Basic Mapping Rule

Recall that a PIPVAL **for-construct** expression can be considered as a special case of a Val **for-iter** expression. Let us consider the one-level **for-construct** expression shown below where the body expression is denoted by f.

```
x =
    for i from 1 to n
        T from array_empty
    construct
        f(i,T,A)
    endfor
```

Here we assume f denotes an expression with one input array A. As stated in Chapter 4, the above expression is equivalent to the following Val **for-iter** expression

```
x =
    for  i = 1,
        T = array_empty
    do
        if i > n then T
```

```
        else
            iter
                T = T[i: f(i,T,A)],
                i = i + 1
            enditer
        endif
    endfor
```

Thus, the basic mapping rule for a **for-construct** expression can be developed based on that for a **for-iter** expression outlined in Section 6.7. Our concern is how to use the features of **for-construct** expressions to simplify the construction of the corresponding data flow graphs.

Let us study the data flow graph derived after applying the basic mapping rule to the above **for-iter** expression, as shown in Figure 9.1. Compared with the graph in Figure 6.15, we can see that the graph inside the dotted-lined box corresponds to the graph of M_I[[iterbody]]. The set I of reiteration ports (see Section 6.7) of the iterbody consists of the ports I1,I2 for i and T — the index value name and the temporary array name being constructed. The iter? output is derived from a index limit check of i ($i \leq n$). The result output port R of the iterbody is the port of the result array X. At the beginning, the loop names i and T are initialized to 1 and **array_empty** respectively. Each time through an iteration, the value computed by f is "appended" to the array T at the index value i by the array append actor. After n iterations, the test $i \leq n$ will return an F value. The iteration will be terminated and return T as the result array X.

A very useful feature of the graph is that we can partition it into three different parts, and restructure the graph as shown in Figure 9.2. The role of the first part — enclosed in the dotted-line box IGEN — is: (1) to generate the sequence of index values 1...n for i (port 1); (2) to provide the boolean value sequence T^nF as the iter? output to control when to terminate the iteration (at its output port 2). The role of the second part — enclosed in the dotted-line box AGEN — is to pack the sequence of element values computed by f into a

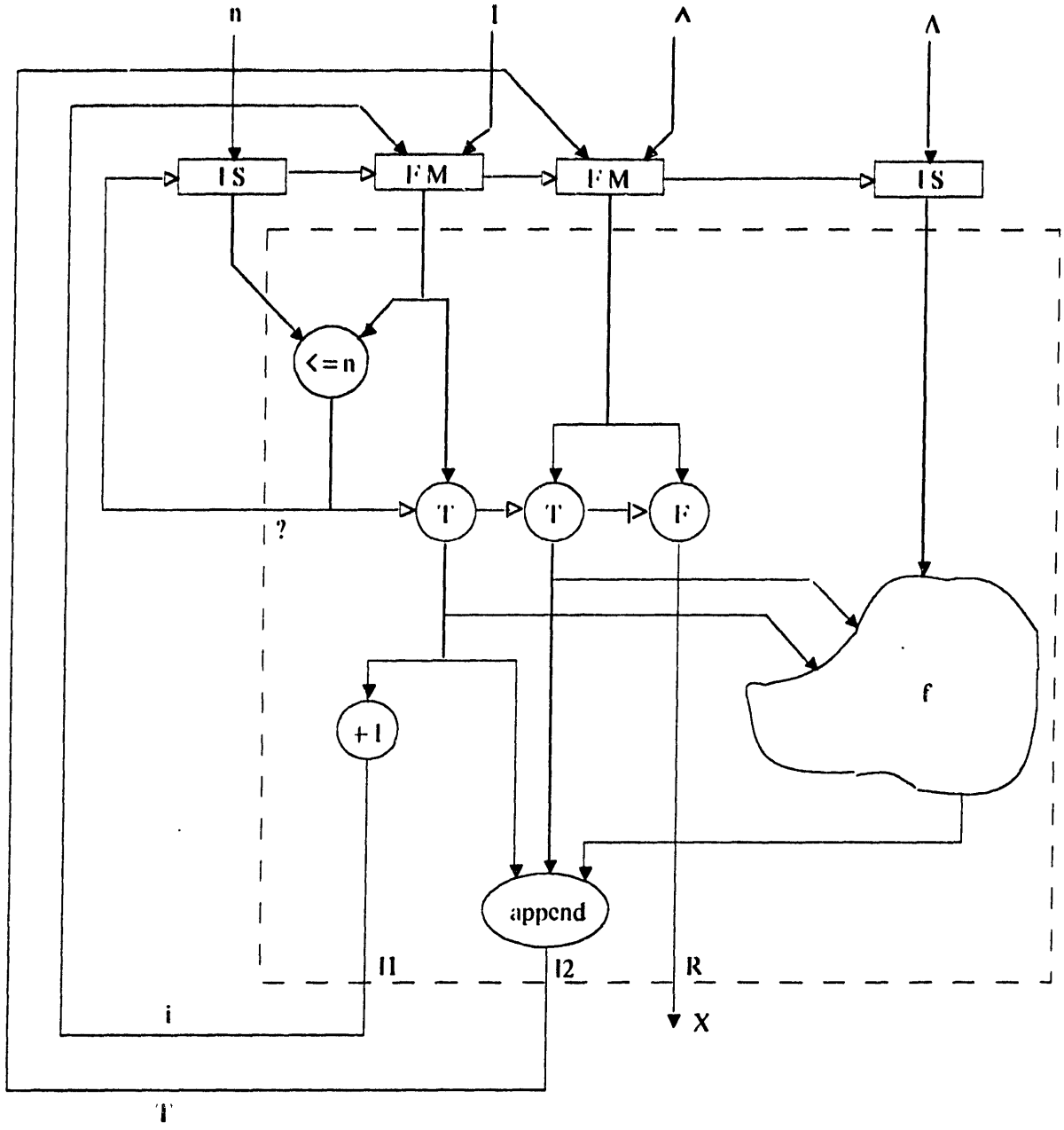


Figure 9.1. The data flow graph of a for-construct expression

result array. The element values and their corresponding indices are taken from input ports 3 and 2 respectively. Under the control value iter? from the input port 1, the sequence of element values is assembled into the internal array T at each iteration and the array value is delivered at R when the iteration terminates. The major role of the third part

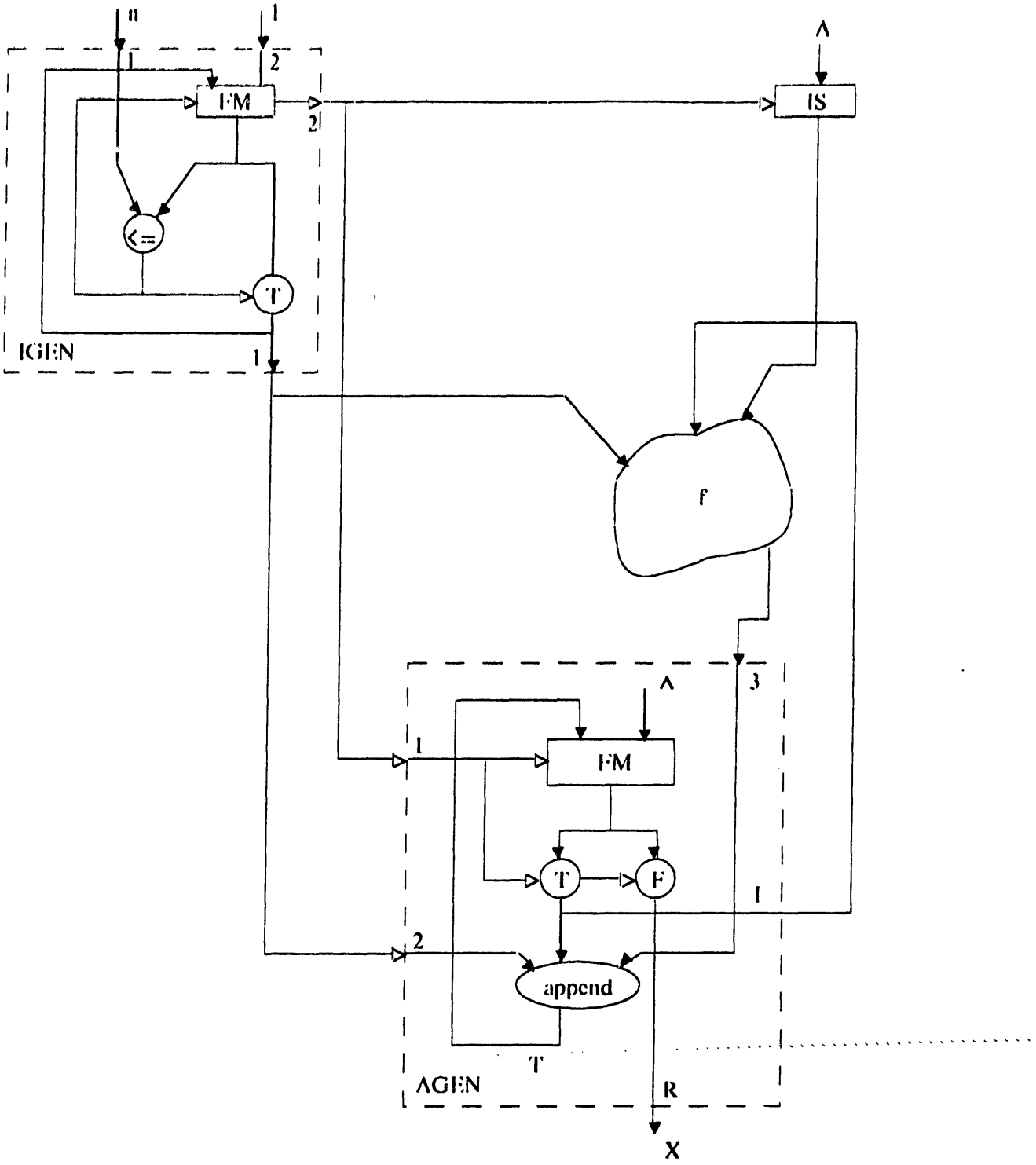


Figure 9.2. The Functions of IGEN and AGEN subgraphs

is to compute f and generate a sequence of element values..

In AGEN, an append actor is used which corresponds to the VAL array append

operation [4]. In the data flow graph of AGEN associated with a **for-construct** expression, such append operations always start with an empty array **array_empty** (denoted by $\wedge$ in the graph), whose bounds are known at the start of array construction. Furthermore, the pattern in which each element is to be "appended" to the array is regular — the array is filled up at consecutive indices in the index range, one element for each index value. For our present discussion, we assume the **array_empty** constant operated on by such a series of append operations will always be performed correctly. The impact of regularity on efficient implementation of array operations on the target machine will be addressed in Chapter 11.

We can observe that the graph for any **for-construct** expression can be so partitioned, we introduce two named subgraphs IGEN, AGEN to denote the corresponding parts. In fact, we have also seen how these named subgraphs are used conveniently in the construction of the data flow graphs for **forall** expressions. A discussion of implementation issues of these subgraphs is included in Chapter 11.

The simple structure of the result graph motivates the development of the following basic mapping rule for the **for-construct** expressions. A result graph is explicitly constructed from the subgraphs IGEN, AGEN and the graph of the body expression, the latter is derived by recursive application of the set of basic mapping rules to the body. Thus the need of a separate mapping rule for the iterbody (such as M_1 outlined in Chapter 6) is met by explicitly utilizing the structure embedded in AGEN and IGEN subgraphs. This mapping rule is presented in Figure 9.3. The corresponding SDFGL graph is shown in Figure 9.4. Note that the feedback path is introduced in the result graph by using the I output port of AGEN to reiterate the array to the body expression.

Figure 9.5 shows the result of the application of the basic mapping rule to the first-order linear recurrence in Figure 9.6. Here we use the same convention of simplified notation introduced for **forall** expressions (see Section 7.1). Note the role of the feedback link from the I output port of AGEN.

M[[for id from exp_1 to exp_2
T from array_empty
construct exp endfor]]

def

Λ = IN(M[[exp]]) - {T,id}
L = IN(M[[exp_1]])
H = IN(M[[exp_2]])
B = $\Lambda \cup L \cup H$

conditions

the result array is to be generated in the major normal order
i.e., l < h (where l = val(exp_1), h = val(exp_2))

remarks

#L = #H = 1, exp_1 and exp_2 are of type integer.

input ports: (a $\in$ B)a

output ports: 1

links: (i $\in$ {1...5})α_i, (a $\in \Lambda$)β_a, β_1

components:

(a$\in\Lambda$)IS-gate inputs: $\alpha_4 \rightarrow 1$, a $\rightarrow 2$
outputs: $1 \rightarrow \beta_a$
M[[exp_1]] inputs: (a$\in$L)a $\rightarrow$ a
output: $1 \rightarrow \alpha_1$
M[[exp_2]] inputs: (a$\in$H)a $\rightarrow$ a
output: $1 \rightarrow \alpha_2$
IGEN inputs: $\alpha_1 \rightarrow 1$, $\alpha_2 \rightarrow 2$
outputs: $1 \rightarrow \alpha_3$, $2 \rightarrow \alpha_4$
M[[exp]] inputs: (a$\in\Lambda$)$\beta_a \rightarrow$ a, $\alpha_3 \rightarrow 1$, $\beta_1 \rightarrow$ I
output: $1 \rightarrow \alpha_5$
AGEN inputs: $\alpha_4 \rightarrow 1$, $\alpha_3 \rightarrow 2$, $\alpha_5 \rightarrow 3$
output: R $\rightarrow$ 1, $1 \rightarrow \beta_1$

Figure 9.3. The basic mapping rule for for-construct expressions

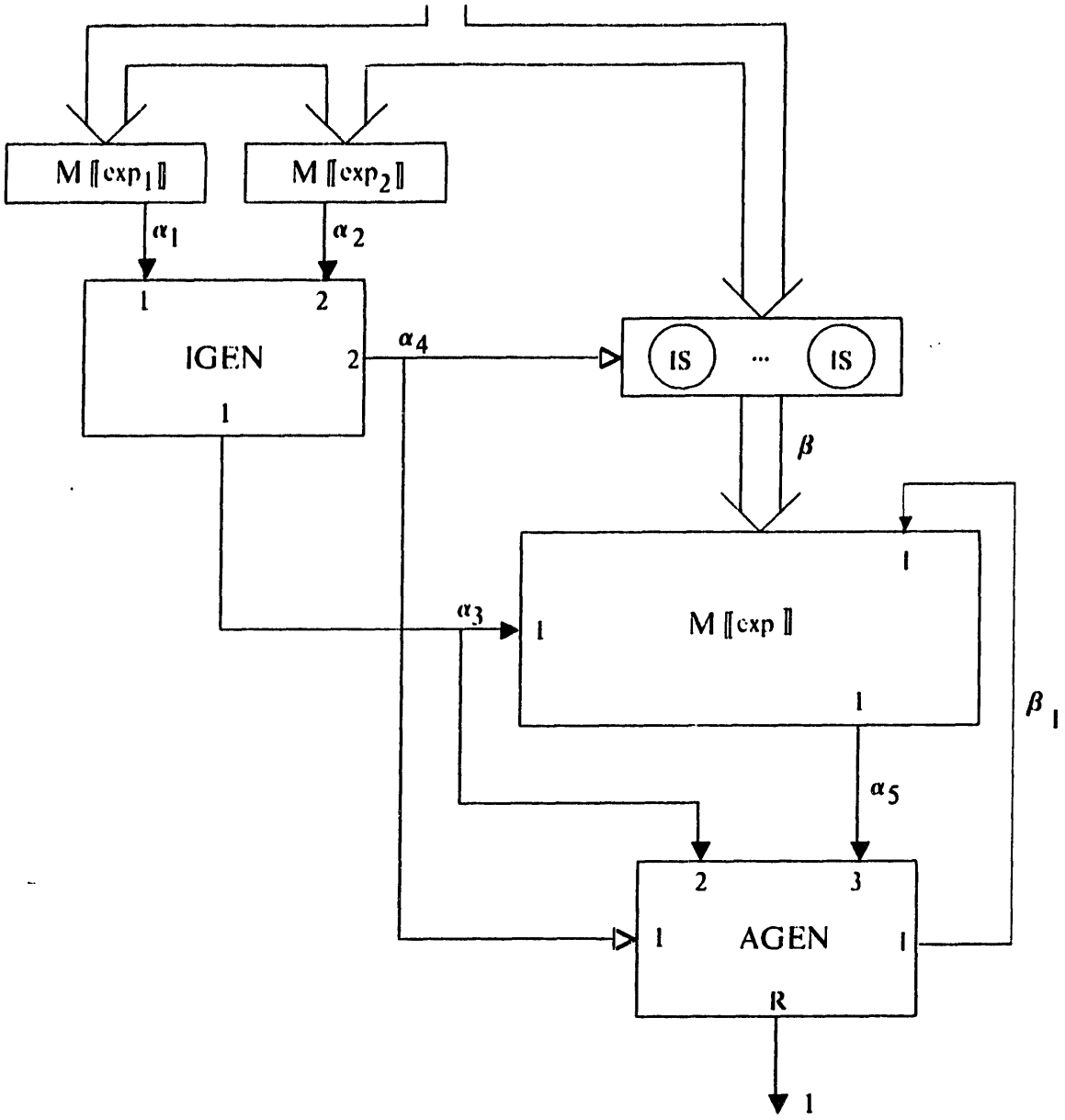


Figure 9.4. The SDFGL graph from the mapping of a for-construct expression

The above basic mapping rule can also be recursively applied to a multi-level **for-construct** expression, as it is in the case of the **forall** expressions. Figure 9.8 shows the result graph for the mapping of the two-level **for-construct** expression illustrated in Figure 9.7 (it is the same as the example in Section 4.3, here we include a copy for the reader's

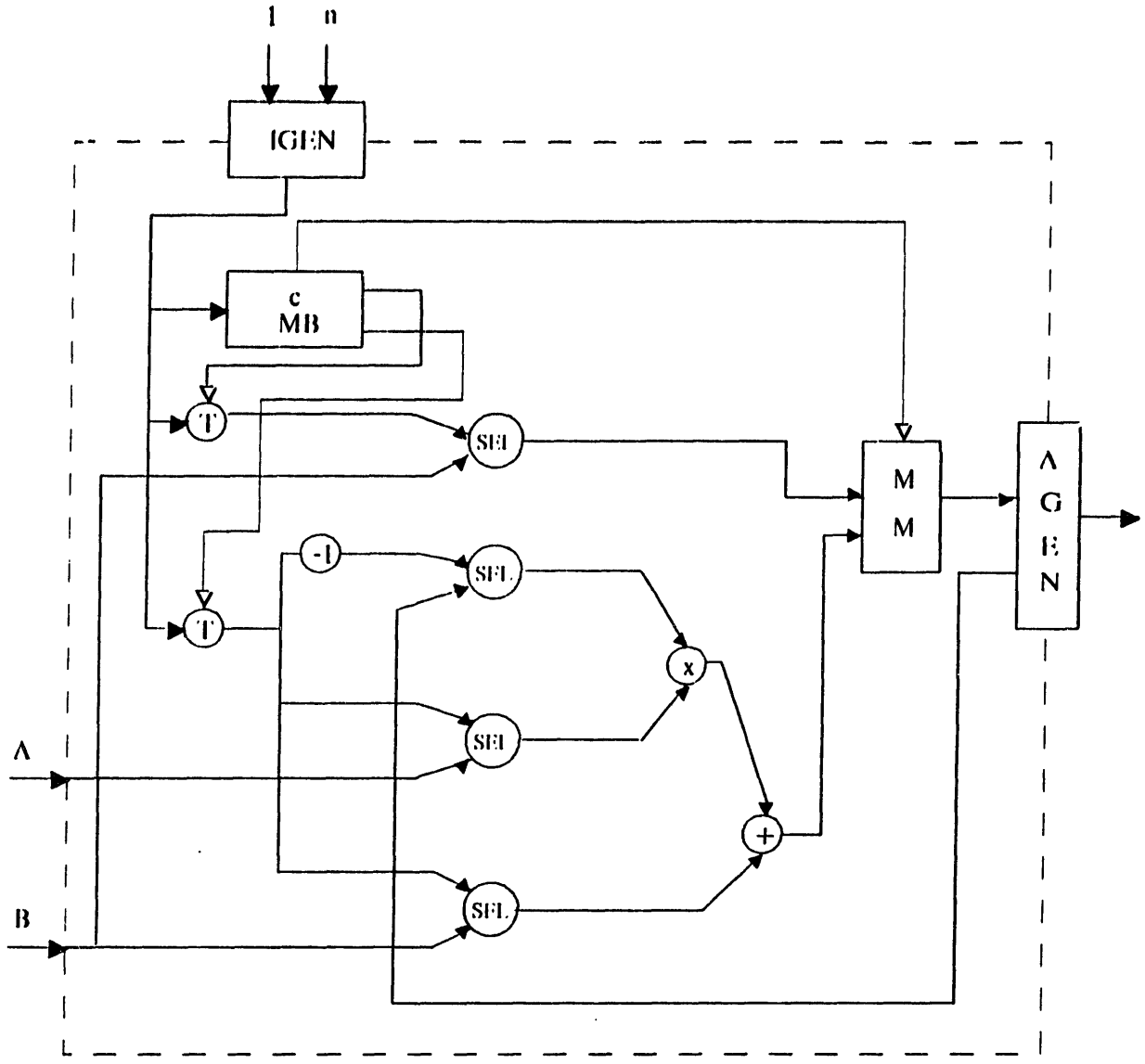


Figure 9.5. The data flow graph from mapping a first-order linear recurrence

convenience). Note the similarities between this graph and the graph in Figure 8.5 A major difference is the two feedback links introduced from the two AGEN subgraphs to the body of the graph. As before, for simplicity we omit the T-gate and IS actors for both the input arrays A,B and the two internal arrays.

```
X =
        for     i from 1 to n
                T from array_empty
        construct
                if i = 1 then B[i]
                else
                        A[i]*T[i-1] + B[i]
                endif
        endfor
```

Figure 9.6. A first-order linear recurrence

9.2 The Optimization Procedures

As with primitive **forall** expressions, the array operations in the data flow graphs of a primitive **for-construct** expression generated by the basic mapping rule may be removed and replaced by ordinary graph actors. Such optimization can be performed only if the generation orders of the input arrays match the selection orders of the corresponding array selection operations, and the generation order of the result array matches the order in which it will be used by the succeeding code blocks. The difference is that the data dependency defined by a **for-construct** expression usually demands a certain generation order of the result array as well as the selection order of the input arrays. Therefore, we do not have the flexibility of choosing the order as we do for **forall** expressions. In this thesis, we are only interested in the case where the result array is generated in one of the major orders.

The principles of the optimization procedures for the primitive **for-construct** expressions are similar to that of **forall** expressions. As before, the key is to compute the selection index ranges of array selection operations and their corresponding SEL actors. In

```
U'' =
        for   i from 0 to m+1
              T1 from array_empty
        construct
              if i = 0 then U[i]
              elseif i = m+1 then U[i]
              else
                    for   j from 0 to n+1
                          T2 from array_empty
                    construct
                          if j = 0 then U[i,j]
                          else j = n+1 then U[i,j]
                          else
                                (U[i+1,j] + U[i,j+1]
                                 + T1[i-1,j] + T2[i,j+1])*1/4
                          endif
                    endfor
              endif
        endfor
```

Figure 9.7. An example of a two-level for-construct expression

our discussion, we are mainly interested in the situation where these selection index ranges are compile-time computable. In particular, we are interested in the primitive **for-construct** expressions, which are in a similar form as the case-1 **forall** expressions. We will not repeat the definition of such **for-construct** expressions, which can be easily deduced from its **forall** counterparts.

As anticipated, the optimization procedure for a case-1 **for-construct** expression is very similar to that of a case-1 **forall** expression. The major difference is the need to handle the feedback paths introduced for the internal arrays. This need has the following two impacts. First, the optimization of array operations should include not only the input arrays generated by other program blocks, but also the internal arrays which are reiterated

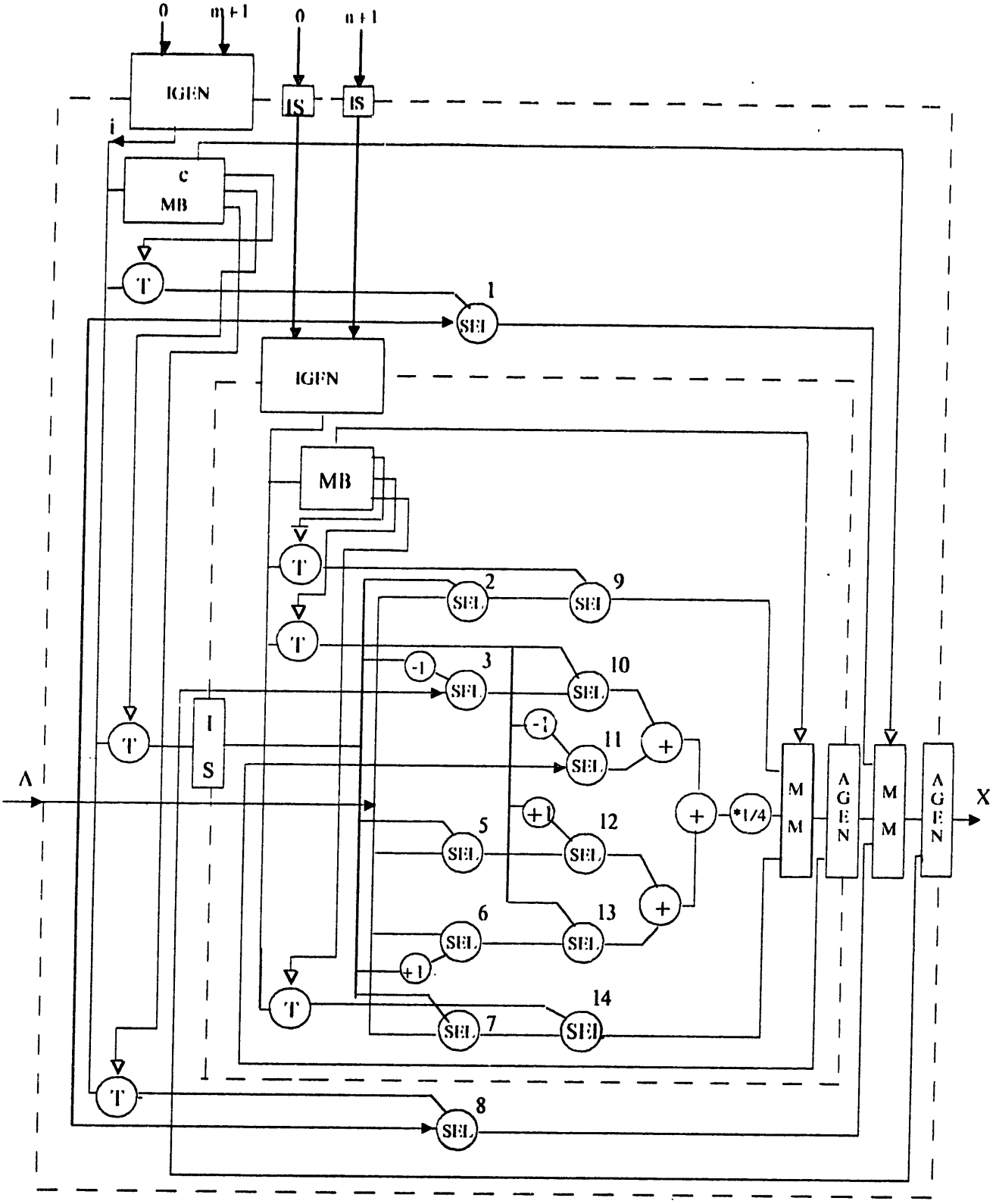


Figure 9.8. The mapping of a two-level for-construct expression

to be used inside the body. This is easy to handle because the index range of an internal array is the same as that of a result array, and hence is the same as that of the corresponding **for-construct** expression. Moreover, it is compile-time computable for a case-1 expression. The generation order of any internal array is the same as that of the result array. Since we are only interested in the situation where the selection orders of all selection operations for an internal array are the same as the generation order of the corresponding internal array, the selection operations for any internal array can be handled just like the selection operations of other input arrays.

Another impact is in the optimization of AGEN subgraphs. Remember that when an AGEN is removed, the array is "flattened" and becomes a sequence of element values carried by tokens on one arc which is then branched into the output arc (from output port R) and the arc for the feedback link (from the output port I). In order to hold the sequence of element values of the result array, certain FIFOs should be introduced on both the

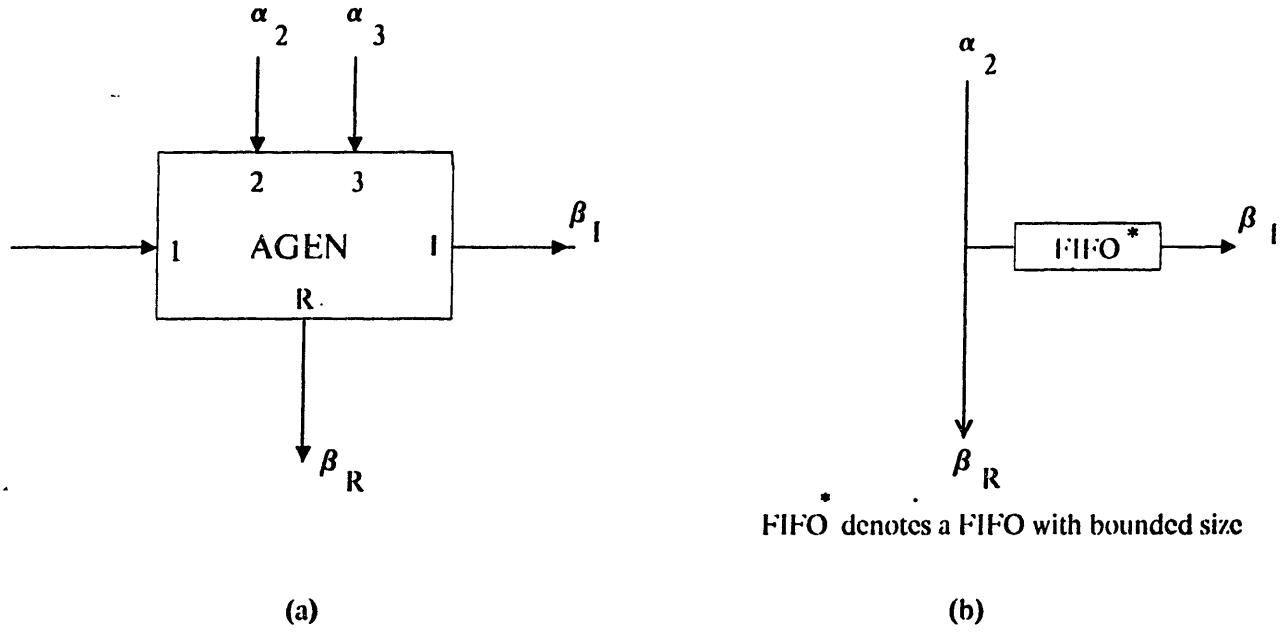


Figure 9.9. Optimization of AGEN in a for-construct expression

output arc and the feedback arc. The former will be discussed in Chapter 11. The latter is only visible inside the code block under consideration. The problem of introducing FIFOs of a precise size into a cyclic graph for balancing purposes is still open as addressed in Chapter 3. Nonetheless, the maximum size of the FIFO holding the element values of the internal array on the feedback link is the same as the size of the corresponding result array -- a compile-time constant. Thus, we can introduce a FIFO of bounded size on the feedback link. The implementation of such FIFOs on the target machine will be discussed in Chapter 15.

The optimization procedure for case-1 primitive **for-construct** expression is very similar to the corresponding optimization procedure of the **forall** expressions presented in the last two chapters. We will not discuss the procedure in detail, but merely point out that

SEL	index expression	index range	selection range	p	q	r	control value sequence
1	i	[0,0]	[0,0]	0	1	m+1	$C1 : T^{d} F^{(m+1)d}$
2	i	[1,m]	[1,m]	1	m	1	$C2 : F^{d} T^{md} F^{d}$
3	i-1		[0, m-1]	0	m	2	$C3 : T^{md} F^{2d}$
4							
5	i		[1,m]	1	m	1	$C5 : F^{d} T^{md} F^{d}$
6	i+1		[2, m+1]	2	m	0	$C6 : F^{2d} T^{md}$
7	i		[1,m]	1	m	1	$C7 : F^{d} T^{md} F^{d}$
8	i	[m+1, m+1]	[m+1, m+1]	m+1	1	0	$C8 : F^{(m+1)d} T^{d}$

Figure 9.10. The selection index ranges and control sequences -- level-1 SEL actors

SEL.	index expression	index range	selection range	p	q	r	control value sequence
9	j	[0, 0]	[0, 0]	0	1	n+1	C9 : TF^{n+1}
10	j	[1, n]	[1, N]	1	n	1	C10 : $FT^{n}F$
11	j-1		[0, n-1]	0	n	2	C11 : $T^{n}F^{2}$
12	j+1		[2, n+1]	2	n	0	C12 : $F^{2}T^{n}$
13	j		[1, n]	1	n	1	C13 : $FT^{n}F$
14	j	[n+1, n+1]	[n+1, n+1]	n+1	1	0	C14 : $F^{n+1}T$

Figure 9.11. The selection index ranges and control sequences -- level-2 SEL actors

the only change is in step 0, which should now be performed according to Figure 9.9. The selection operations of the internal array can be treated just as that of other input arrays. An example in the next section illustrates such changes.

9.3 An Example of Optimization

Let us study the optimization process of the data flow graph in Figure 9.8. An optimization procedure will compute the selection index ranges and control sequences for the level-1 and level-2 SEL actors in the graph, as shown in the two tables of Figure 9.10 and Figure 9.11. Both the AGEN subgraphs and the SEL actors can be removed by a optimization procedure as shown in Figure 9.12. After further optimization steps, the MB and IGEN subgraphs are removed. The result is shown in Figure 9.13. It is interesting to note how the above optimization process is similar to that for the **forall** expression in Figure 8.4 (see also Figure 8.16 to Figure 8.19).

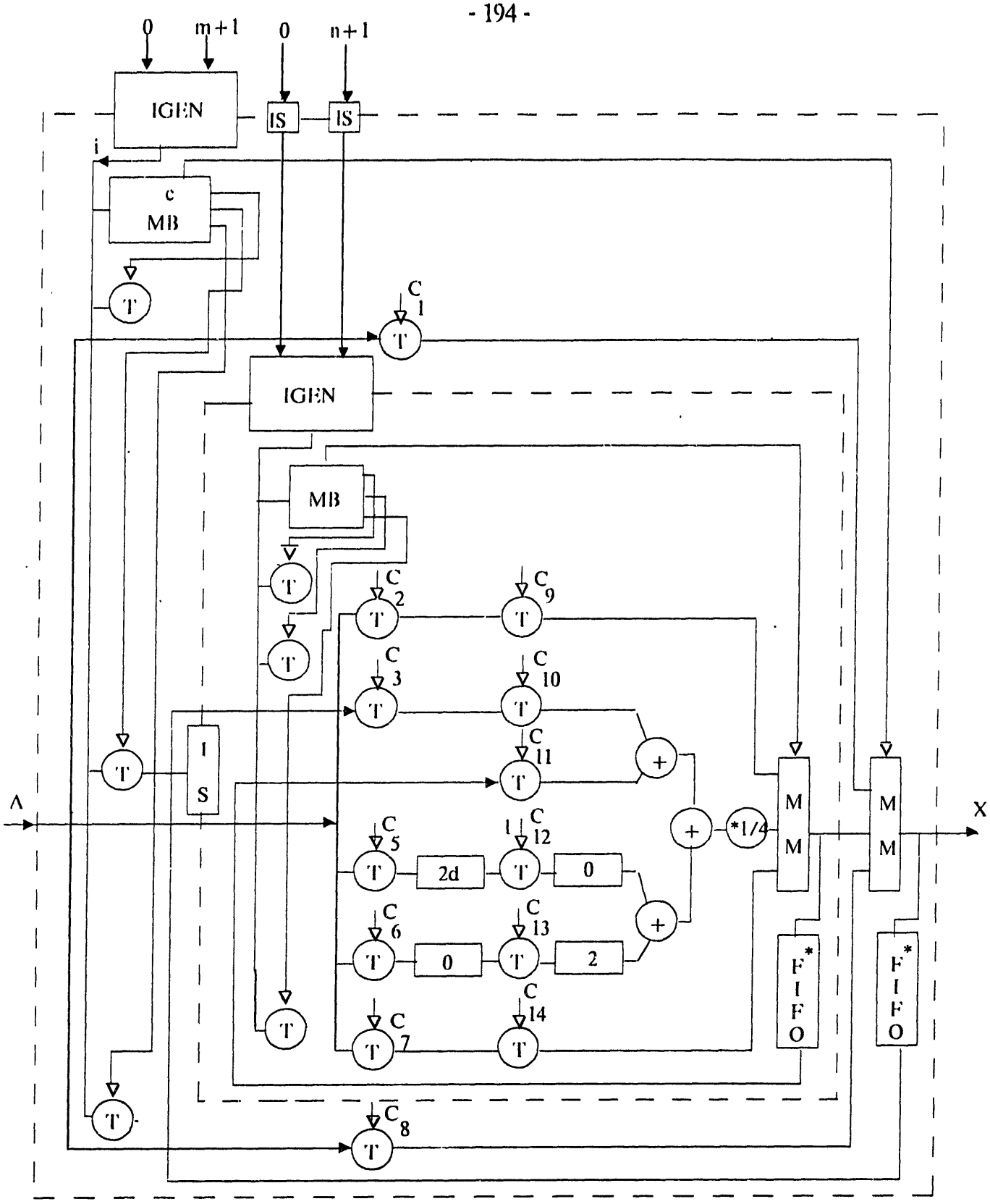


Figure 9.12. A two-level for-construct optimization example -- after removing array actors

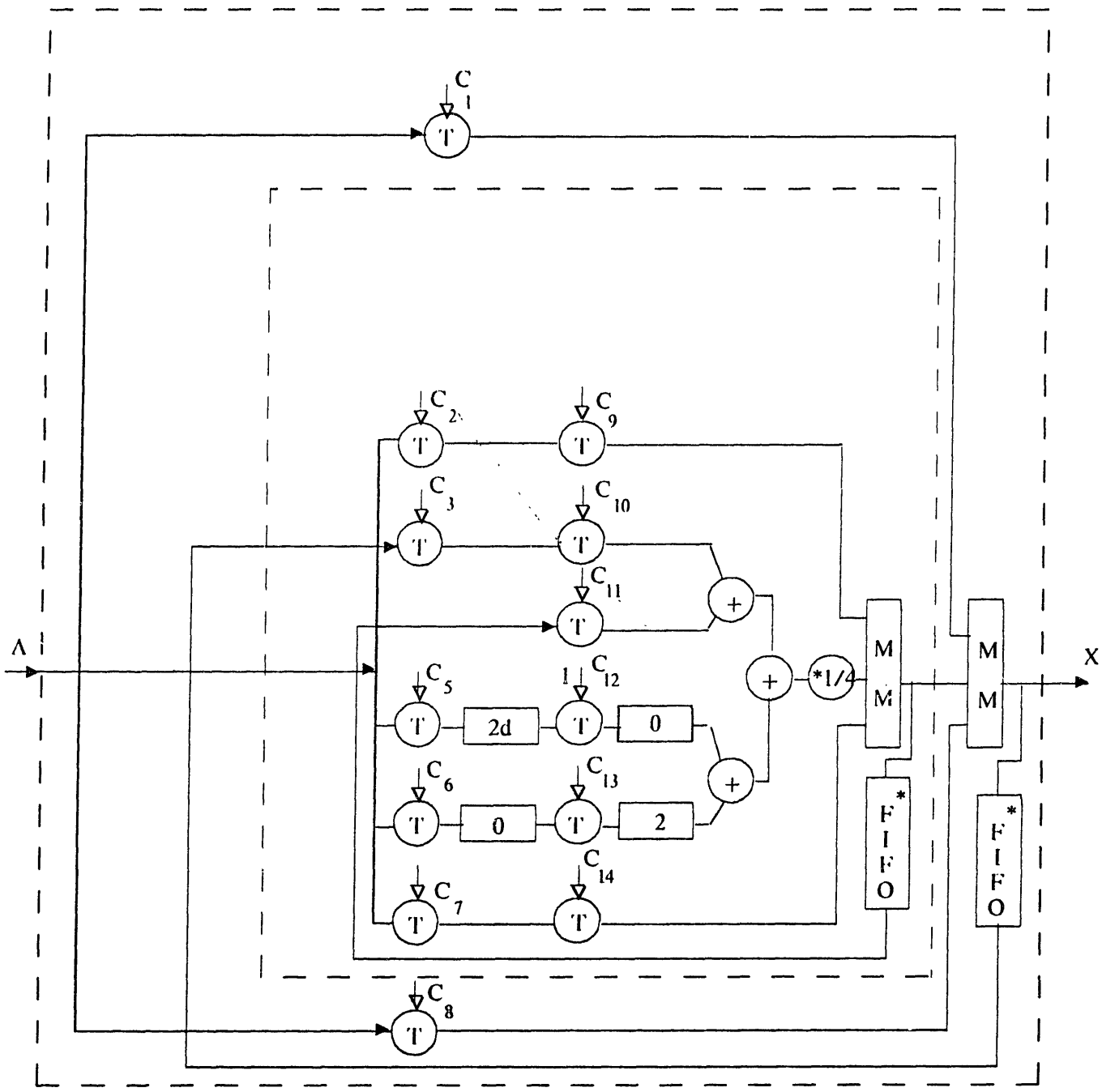


Figure 9.13. A two-level for-construct optimization example -- final result

10. A Survey of Related Optimization Techniques

Chapters 4-9 complete our presentation of the basic pipelined code mapping schemes for PIPVAL expressions, including the two important classes of array construction expressions — primitive **forall** expressions and primitive **for-construct** expressions. These basic mapping rules and optimization procedures provide a basis upon which other transformations and optimization techniques may be incorporated and applied.

As we said earlier, the performance of the innermost level is critical, because it usually constitutes the most computationally intensive part of the program. When the innermost expression is a primitive **forall** expression, the basic mapping scheme described earlier can be applied directly to generate fully pipelined data flow graphs for the innermost expression. If the innermost level consists of a primitive **for-construct** expression, the degree of pipelining that can be achieved (by the direct application of the basic mapping scheme) is often limited by the data dependencies implied by the iteration. In this chapter we survey a few other transformation techniques which, combined with the basic scheme, allow such innermost expressions to be mapped more efficiently in terms of pipelining.

As with conventional language translators, these transformation techniques are usually known as "optimization" techniques, although the word "improvement" may be more appropriate. Our survey concentrates on how these techniques can be combined with our basic pipelined code mapping schemes to improve the performance of the result graphs. Such improvements are often achieved through compromise among different objectives. Use of a particular optimization technique depends on both the nature of the innermost loop and its surrounding expressions in the code block to be mapped and the relation between this code block and other code blocks. No universal scheme exists which is suitable for every situation. Therefore, our discussion will be closely coupled with examples which may illustrate trade-tradeoffs frequently encountered in real applications.

In Section 10.1 we consider the case where the innermost expression computes linear

recurrences. In Section 10.2, we briefly outline how our work relates to loop-unfolding and array-interleaving techniques; in section 10.3, we present techniques which exploit parallelism embedded in outer level expressions (usually **forall** expressions).

10.1 Using Companion Pipelines in Solving Linear Recurrences

Linear recurrences share with arithmetic expressions a role of central importance in scientific numerical computations. Such recurrences are fundamental to the solution of linear equations by Gaussian-elimination; to all matrix manipulations which need an inner product of vectors; and to the solution of differential equations [54,67]. The solving of recurrences may create bottlenecks in a parallel computer because of the sequential constraints implied in the recursive definition. Therefore, it is very important to find fast and efficient solutions to linear recurrences for parallel computers.

The technique surveyed in this section is based on the use of a *companion function* to transform a linear recurrence so that a *companion pipeline* is introduced in the result data flow graph to achieve maximum pipelining. The concept of companion function and the general optimization technique based on companion functions are described in [61,62,63,64]. The application of companion pipelines, for maximum pipelining of data flow graphs is studied in [43,46]. In this survey we assume the readers are familiar with the basic ideas, and our goal is to show how such optimization can be combined with the basic pipelined code mapping schemes. We use first-order and second-order linear recurrences as examples in our discussion.

10.1.1 Mapping Of First-order Linear Recurrences

A first-order linear recurrence is described by the following equations:

$$x_1 = b_1$$
$$x_i = a_i x_{i-1} + b_i, \quad i = 2...n \qquad (10.1)$$

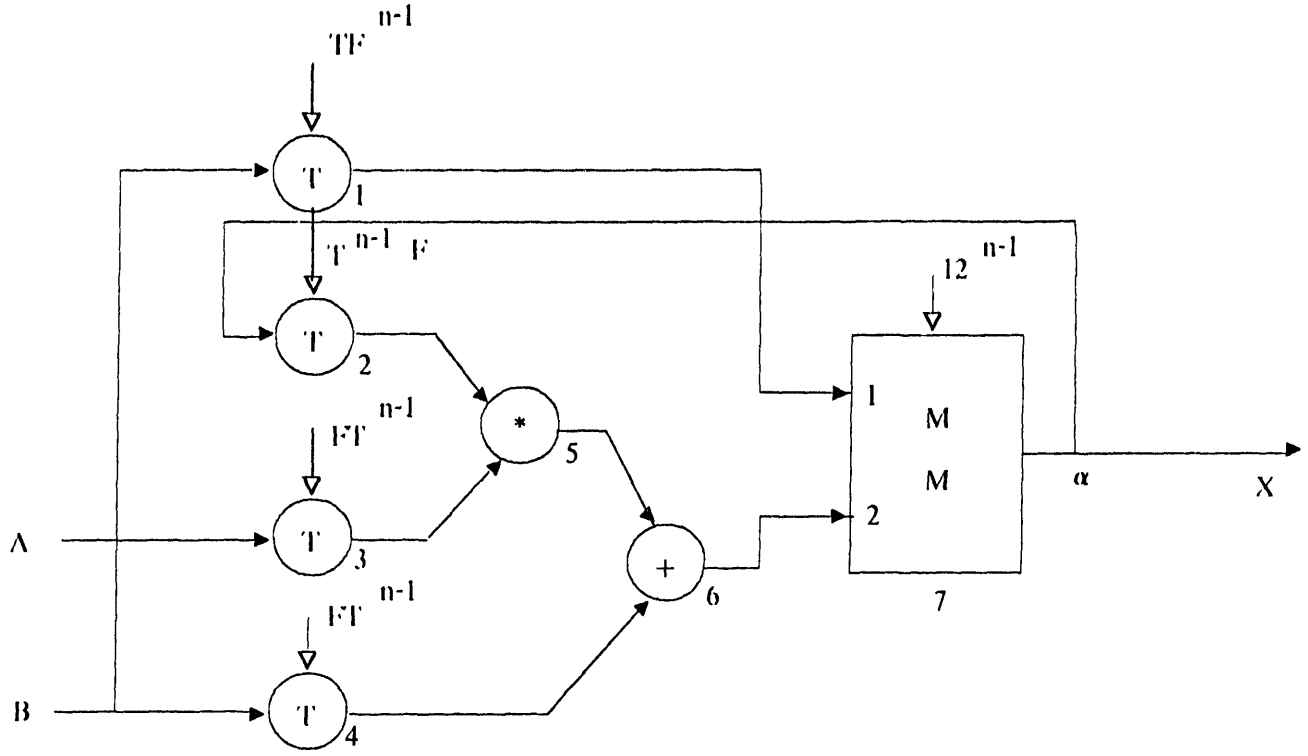


Figure 10.1. The data flow graph of a first-order linear recurrence

It can also be specified by the following **for-construct** expression

```
X =
        for     i from 1 to n
                T from array_empty
        construct
                let x1 = B[1]
                in
                        if i = 1 then x1
                        else
                                A[i]*T[i-1] + B[i]
                        endif
                endlet
        endfor
```

Applying the basic mapping rule and optimization procedure to the code block produces the result data flow graph shown in Figure 10.1. There are 4 T-gate actors numbered 1-4. The upper T-gate actor (labeled 1) selects B[1] for the initial value X[1]. The T-gate actor (labeled 2) selects the first n-1 values of X for feedback. The lower two T-gate actors (labeled 3,4) select the value sequences A[2]...A[n] and B[2]...B[n] respectively. We note that MM will select the initial value x_1 from its input port 1 as its first output X[1]. The remaining n-1 outputs x[2]...x[n] are selected by MM from input port 2.

The loop in Figure 10.1 has a length of four (from α through nodes 2,5,6,7 back to α).[1] Thus, the loop has the capacity to process two elements (e.g., x_i and x_{i-1}) of the result array concurrently when run at the maximally pipelined rate. However, the recurrence constraint of (10.1) prevents x_i and x_{i-1} from being processed concurrently.

A solution based on the use of a companion function is to relax the constraint imposed by the recurrence, i.e., perform a transformation which removes the dependence of x_i on x_{i-1}. Equation (10.1) can easily be rewritten as

$$x_i = a_i^{(1)} x_{i-2} + b_i^{(1)} \qquad i = 3...n \qquad (10.2)$$

where $x_1 = b_1$, $x_2 = a_2 b_1 + b_2$, and

$$a_i^{(1)} = a_i a_{i-1} \qquad (10.3)$$

$$b_i^{(1)} = a_i b_{i-1} + b_i \qquad (10.4)$$

The corresponding **for-construct** expression is shown in Figure 10.2.

Figure 10.3 shows the result graph derived by applying the basic mapping rules and optimization procedures to Figure 10.2. Compared with Figure 10.1, the main loop remains the same except that the first two values of x are now selected from the first two

---

1. Here we assume MM counts as one node.

```
X =
        for     i from 1 to n
                T from array_empty
        construct
                let   x1 = B[1],
                      x2 = A[2]*x1 + B[2],
                      AP = A[i]*A[i-1],
                      BP = A[i]*B[i-1]+B[i]
                in
                      if i = 1 then x1
                      elseif i = 2 then x2
                      else
                            AP*T[i-1] + BP
                      endif
                endlet
        endfor
```

Figure 10.2. A first-order linear recurrence with backup

ports (1,2) of MM, which are provided by subgraph INIT for computing initial values. Another subgraph COMP denotes the so-called *companion pipeline* [43], which computes $a_i^{(1)}$, $b_i^{(1)}$ in (10.4). The structure of COMP and INIT are shown in Figure 10.4.

The companion pipeline can be generated automatically by the basic mapping scheme as long as an adequate source level transformation of the program block is performed. Thus we have illustrated how the transformation technique of introducing companion pipeline can be combined with the basic code mapping scheme to systematically generate efficient data flow graphs.

The reader may find it useful to study the first few steps of the execution as shown in Figure 10.5 (a) ~ (c). Here for simplicity, we show only the tokens carrying element values of the result array and the corresponding feedback values. In the configuration of Figure

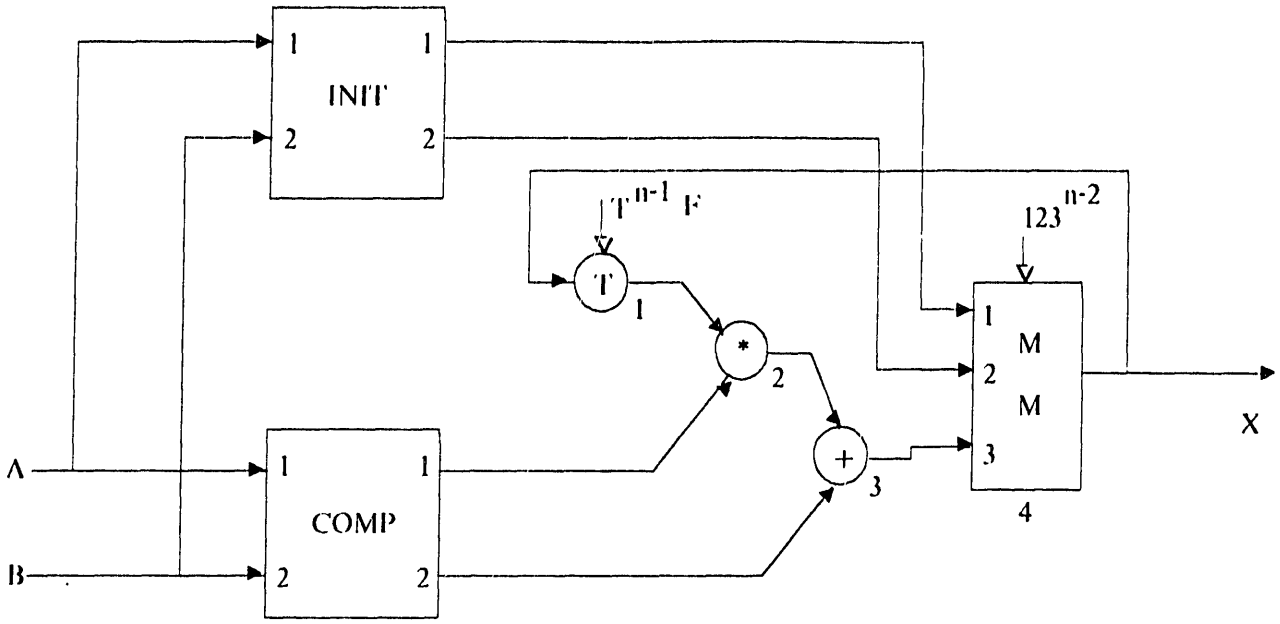


Figure 10.3. A maximally pipelined data flow graph for a FLR

10.5 (c), two alternate set of nodes (node 2,4) are fired, processing x_3, x_4 at the same time. The loop will continuously run in this maximally pipelined fashion, generating the element values of the result array.

10.1.2 A second-order linear recurrence

Consider the fibonacci recurrence as a simple example of a second-order linear recurrence:

$$x_1 = 1$$
$$x_2 = 1$$
$$x_i = x_{i-1} + x_{i-2} \qquad i = 3,4,...n \qquad (10.5)$$

Figure 10.7 shows a corresponding **for-construct** expression which computes (10.5). By directly applying the basic mapping rules and optimization procedure to (10.5), we derive

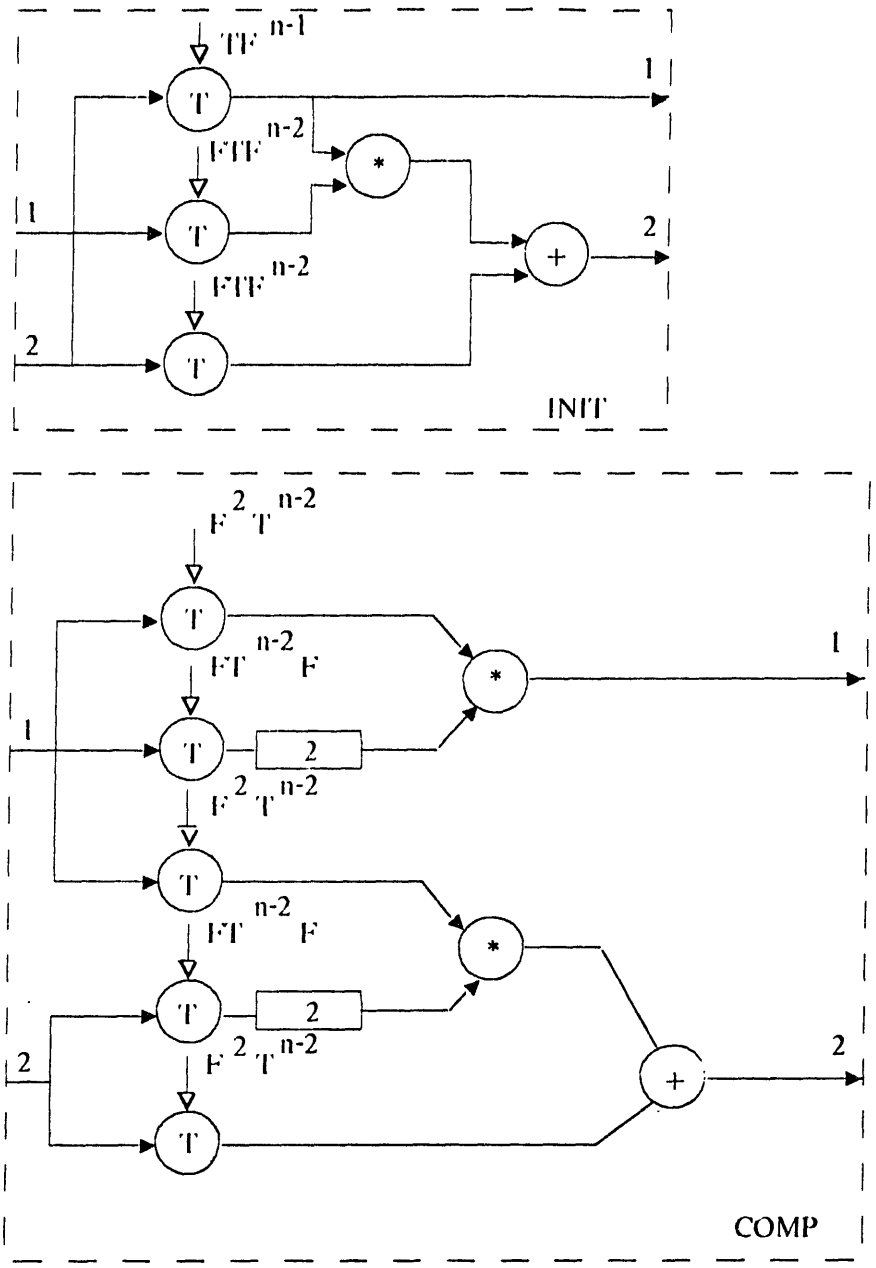


Figure 10.4. The compainion pipeline in Figure 10.3

the data flow graph in Figure 10.6. Note how the two terms x[i-1], x[i-2] of the recurrences are handled naturally by our mapping scheme.

We observe that the loop consisting of node 2-5 has a length of five, and hence has the capacity to process more than two elements of the array at the same time. In order to

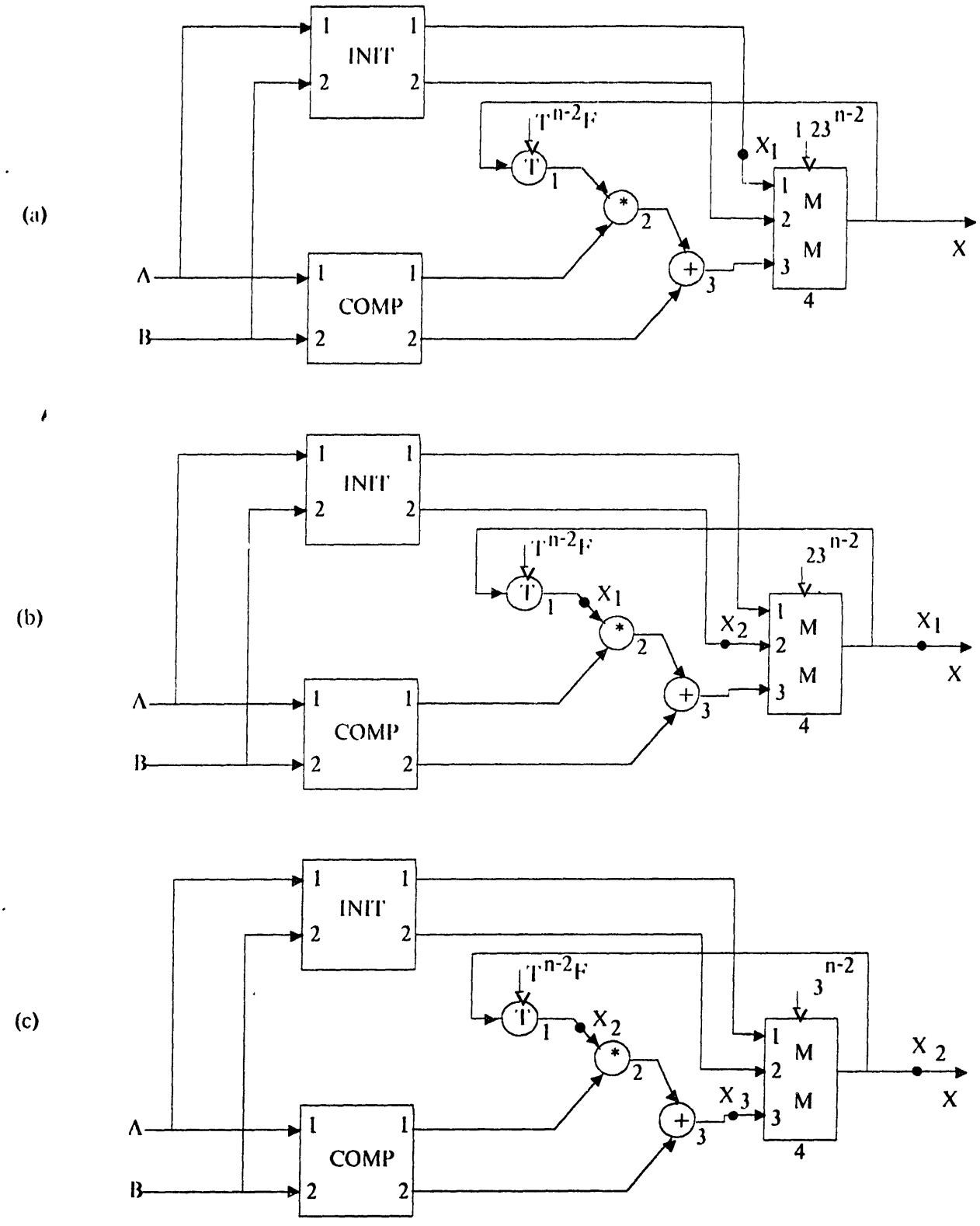


Figure 10.5. Pipelined execution of FLR -- the first few steps

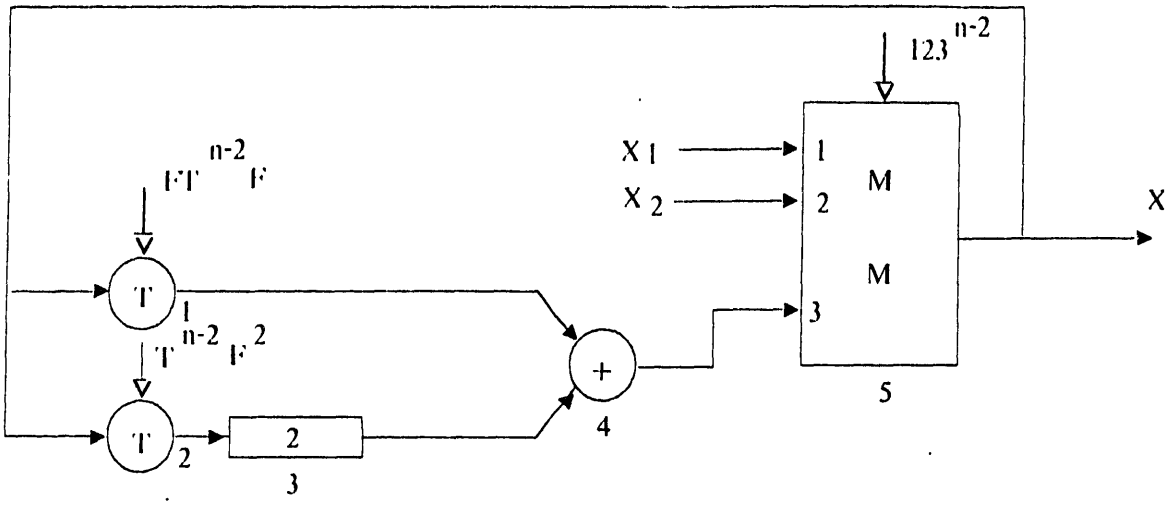


Figure 10.6. A data flow graph for the fibonacci recurrence

```
X =
        for     i from 1 to n
                T from array_empty
        construct
                let     x1 = 1,
                        x2 = 1
                in
                        if i = 1 then x1
                        elseif i = 2 then x2
                        else
                                T[i-1] + T[i-2]
                        endif
                endlet
        endfor
```

Figure 10.7. A for-construct expression for (10.5)

achieve maximum pipelining, the recurrence constraints of (10.5) should be relaxed by the following transformation:

$$x_1 = 1$$

$$x_2 = 1$$

$$x_3 = x_1 + x_2$$

$$x_i = 2x_{i-2} + x_{i-3} \qquad i = 4,5,...n \qquad (10.6)$$

Figure 10.8 shows a **for-construct** expression which computes the transformed fibonacci recurrence according to (10.6). Obviously, it is a primitive **for-construct** expression. Figure 10.9 shows the data flow graph derived by the application of the basic mapping rules and optimization procedures to (10.6). Two FIFOs of total size 3 are placed on the path leading x_{i-3} to the ADD actor: the FIFO of size 2 is for the skewing introduced by the optimization procedure, and the FIFO of size 1 is for the balancing purposes.

It is easy to see that the loop in Figure 10.9 can compute the recurrence in a maximally pipelined fashion. Figure 10.10 (a) - (c) illustrates the first few steps of the

```
X =
        for     i from 1 to n
                T from array_empty
        construct
                let x1 = 1,
                    x2 = 1,
                    x3 = x1+x2
                in
                    if i = 1 then x1
                    elseif i = 2 then x2
                    elseif i = 3 then x3
                    else
                        2*T[i-2] + T[i-3]
                    endif
                endlet
        endfor
```

Figure 10.8. The fibonacci recurrence after transformation

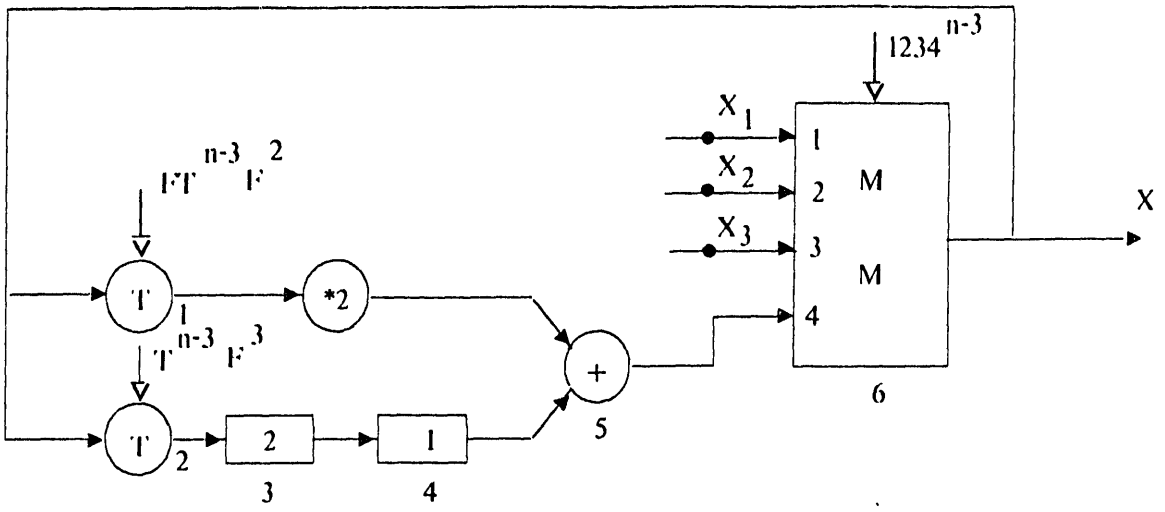


Figure 10.9. The data flow graph for the transformed fibonacci recurrence

computation. Note that the loop in Figure 10.10 (c) can concurrently compute x[4] and x[5] respectively, keeping the loop running at its maximally pipelined rate.

10.1.3 A Discussion

In a large numerical computation program, a linear recurrence may be only part of the whole program. Therefore some input vectors, such as a_i,b_i, may be arrays generated by some preceding code blocks in a maximally pipelined fashion. The result of the recurrence itself may become an input vector for some succeeding computation which may also consume the values in a pipelined fashion. In such cases it may be very inefficient, in terms of space and time, to wait for entire input vectors to be computed before starting the parallel evaluation of the recurrence, as required by the cyclic reduction method for solving linear recurrences [54,87]. In contrast, our scheme evaluates the recurrence in a pipelined fashion, consuming the input and producing the output concurrently. This not only saves the space that would be needed to hold the intermediate result values, but also eliminates substantial data rearrangement overhead. It sustains a relatively constant parallelism during program execution. Furthermore, the machine code size is much smaller and

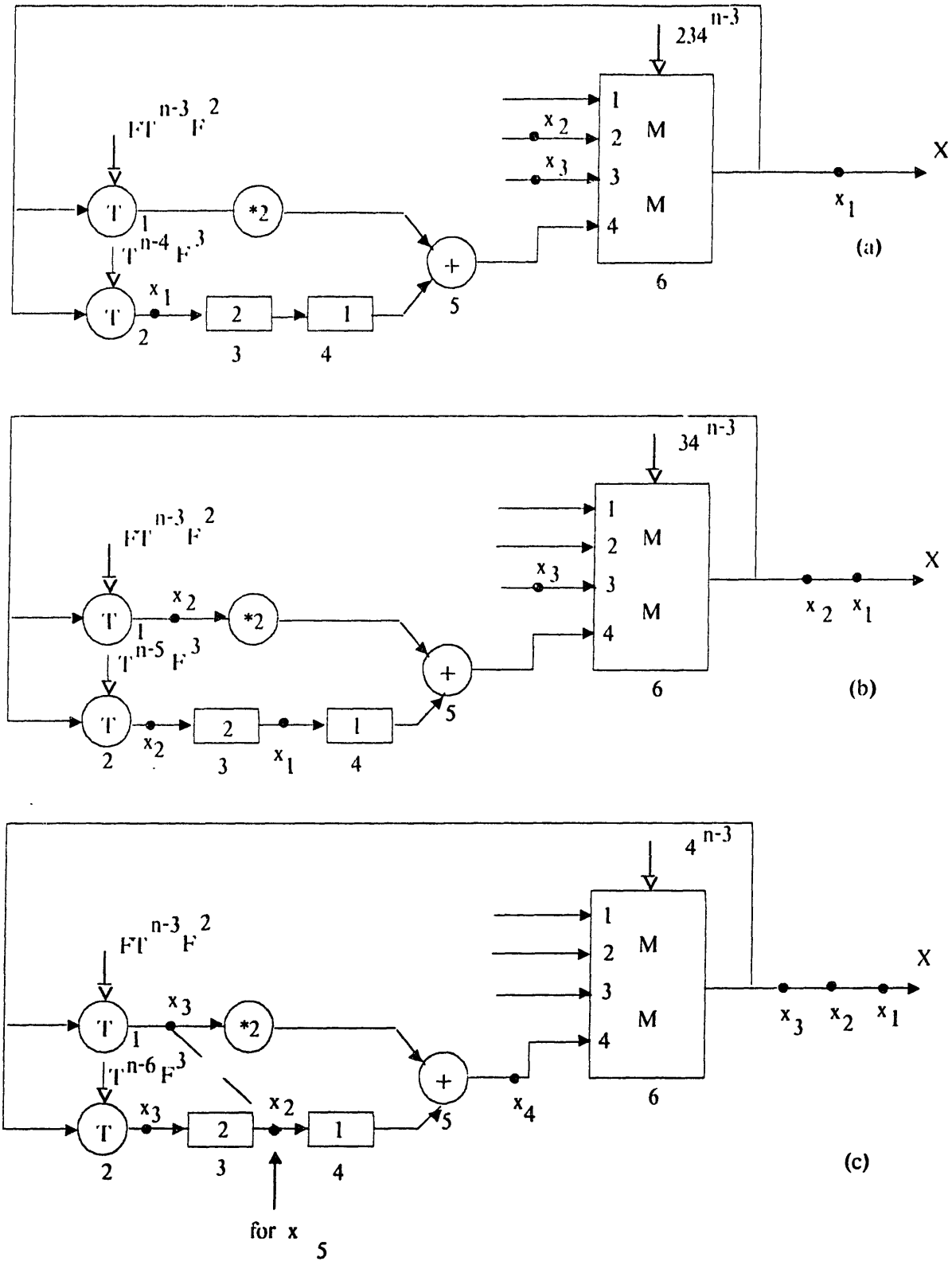


Figure 10.10. Pipelined execution of the transformed fibonacci recurrence

hence more efficient in terms of memory usage.

Finally, note how our mapping scheme handles feedback array references such as T[i-1], T[i-2]..., where T is the internal name of the array being produced. Our optimization procedure removes these array operations just as it does for any other array selection operations.

Based on the above techniques, a maximally pipelined solution scheme for tridiagonal linear equation systems has been studied by the author and the results are reported in [48,49,50].

10.2 Enhancing Pipelining by Loop Unfolding and Interchanging

10.2.1 Loop Unfolding

An important transformation that sometimes can be successfully applied to optimization is the so-called loop unfolding technique. It is essentially a way to "unfold" an iteration into multiple copies of the iteration body. Let us consider the following simple **for-construct** expression

```
X =
    for
        i from 1 to n
        T from array_empty
    construct
        let x1 = f(x0,A[1])        % x0 is constant
        in
            if i = 1 then x1
            else
                f(T[i-1],A[i])
            endif
        endlet
    endfor
```

where f denotes the function computed by some expression. Without loss of generality, we

assume the loop has only one input array A. The simplest unfolding of the above expression is shown below where the function f is duplicated.

```
X =
    for
        i from 1 to n
        T from array_empty
    construct
        let x1 = f(x0,A[1]),
            x2 = f(x1,A[2])
        in
            if i = 1 then x1
            elseif i = 2 then x2
            else
                f(f(T[i-2],A[i-1]),A[i])
            endif
        endlet
    endfor
```

Loop unfolding is a technique frequently used in compilers for conventional computers, where the main goal is to reduce the overhead of loop control, i.e., termination test and exit mechanisms.

Loop unfolding can also be applied to perform optimization for data flow computers. Figure 10.11 (a) and (b) illustrates the effects of unfolding on the above example.[1] The data dependencies between different copies of f may often only appear on a so-called "critical path" encompassing a small part of the computation. Therefore, the execution of a considerable part of the data flow graph in different copies can overlap. This type of application has been extensively studied in [6], where loop unfolding is combined with other techniques such as *array interlacing* to perform code optimization.

In this thesis, we are interested in the loop unfolding technique for a different reason

---

1. The figure is not a complete data flow graph We have omitted the initialization part and the part which merges the element values into the result array.

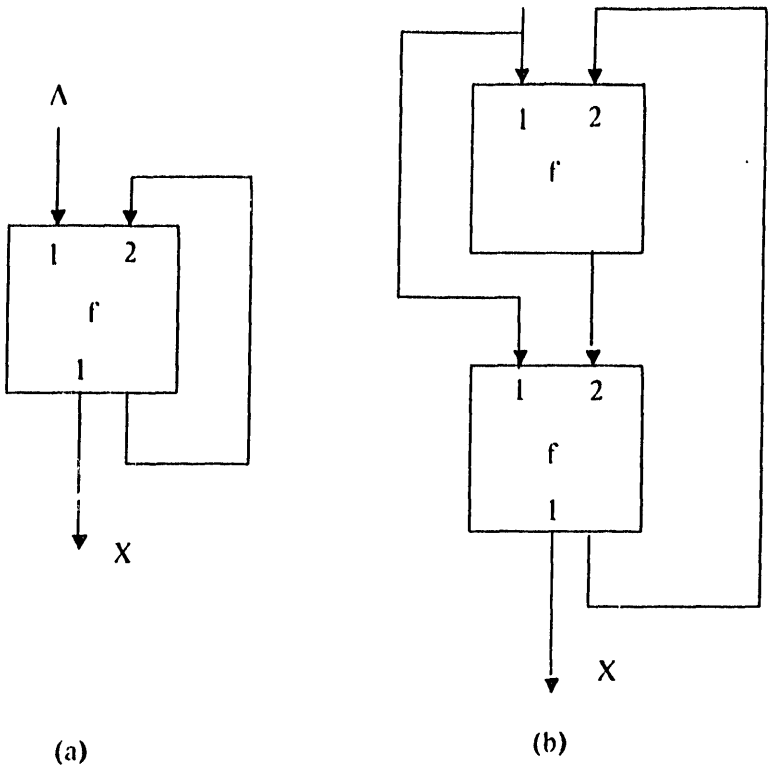


Figure 10.11. Loop unfolding

i.e. to exploit parallelism in terms of efficient pipelining. Assume that the loop in the above example is completely unwound as shown in Figure 10.12, where we omit the part of the graph which handles initial values. Since the cycle is completely removed from the graph, the bottleneck for pipelining is removed. Note, however, that if the expression is only evaluated once, the parallelism provided by the unfolding may not be fully utilized to enhance pipelining. Recall that unfolding the loop does not reduce its critical length.

However, by complete unfolding of the loop, the graph becomes acyclic. Thus, it provides the opportunity for pipelined execution of the graph by different evaluations of the same loop, each being initiated by a separate input vector, e.g. A_1, A_2,...A_m, as shown in Figure 10.13. Here we consider the above completely unfolded graph as a multi-input-multi-output pipeline, with each set of input values corresponding to one row of a two-dimensional input array A, and each set of output values corresponding to one row

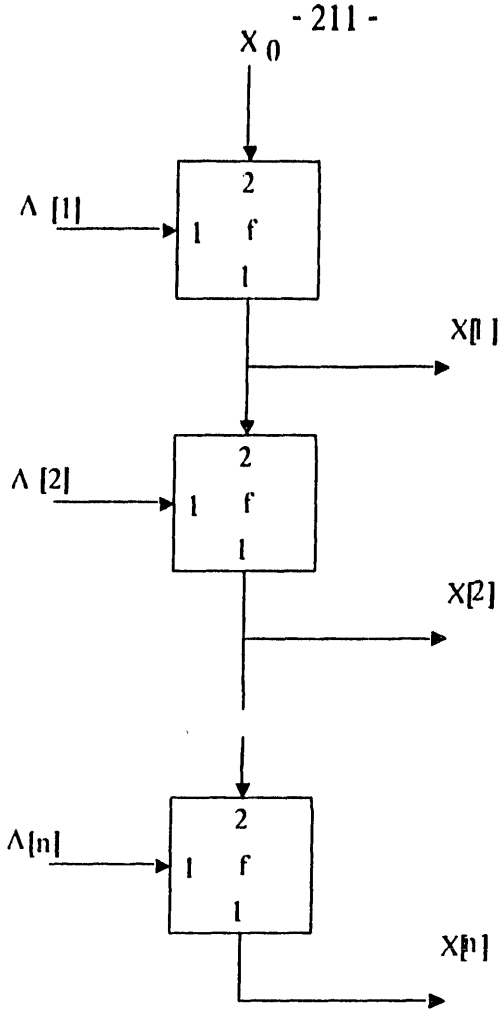


Figure 10.12. A completely unfolded loop

of a two-dimensional result X. Since each graph of f is acyclic, the entire graph can be maximally pipelined.

The above example shows that loop unfolding can be a very useful transformation technique in the pipelined mapping of a **for-construct** expression. It can be applied whether or not the corresponding recurrence has a companion function.

In Section 10.2.3, we present an example that illustrates how the unfolding technique can be successfully combined with other optimization techniques.

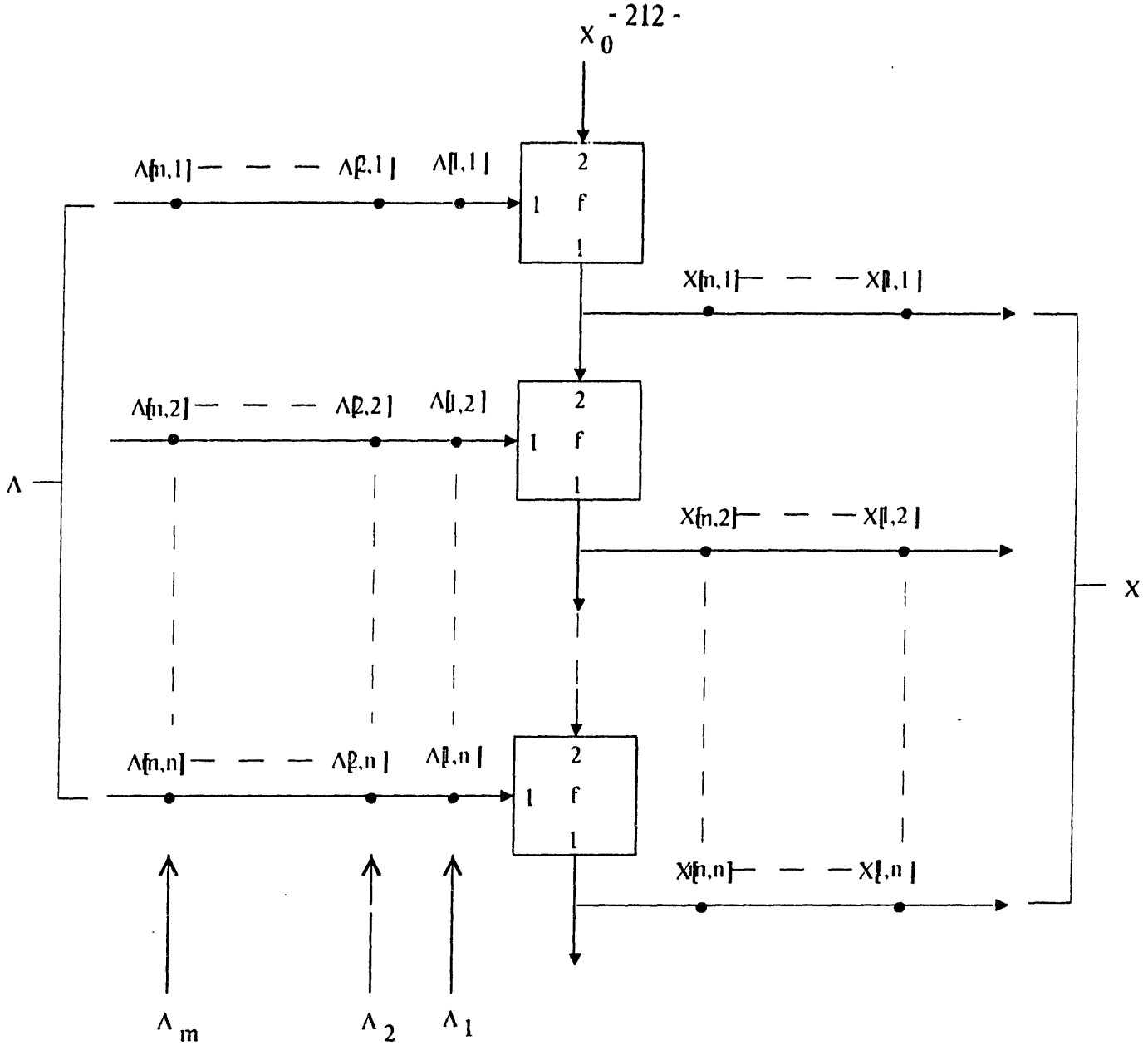


Figure 10.13. Maximum pipelining of a completely unfolded loop

10.2.2 Interchanging the Order of the Innermost Loop

Let us consider the following nested mixed expression :

```
X =
    forall i in [1,m]
    construct
        for   j form 1 to n
              T from array_empty
        construct
              f(A,T,i,j)
        endfor
    endall
```

The outer expression is a **forall** expression with index range [1,m], while the inner expression is a **for-construct** expression. The code block constructs a two-dimensional array X with m rows where no data dependencies exist between elements of different rows. Therefore we can rearrange the computation such that the array is computed in a column-by-column fashion. This can be achieved by interchanging the order of the outer and inner expression as follows:

```
X =
    for   j from 1 to n
          T from array_empty
    construct
          forall i in [1,m]
          construct
                f(A,T,i,j)
          endall
    endfor
```

After the transformation the **forall** expression becomes the innermost expression, and we can generate a pipelined data flow graph using the basic code mapping scheme.

10.2.3 A Matrix Multiplication Example

Matrix multiplication provides an interesting example of the different ways in which data flow graphs can be organized and structured to best exploit parallelism in terms of pipelining. In this section we study a special case of vector multiplication and illustrate how loop unfolding and interchange can be combined in a pipelined solution. This requires the application of both the basic mapping scheme and some special optimization techniques.

The matrix-vector multiplication problem can be expressed as follows:

$$\begin{bmatrix} a_{11} & \cdots & a_{1p} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mp} \end{bmatrix} \times \begin{bmatrix} x_1 \\ \vdots \\ x_p \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \qquad (10.7)$$

or in matrix-vector notation:

$$\mathbf{Ax} = \mathbf{y} \qquad (10.8)$$

where A is an m×p matrix, and x,y, are vectors of size p and m respectively.

The matrix-vector multiplication can be computed by forming the inner product of each row of A with the vector y, as specified by the expression below.

```
Y =
   forall  i in [1,m]
   construct
         for k from 1 to p
               T from array_empty
         construct
               if k = 1 then A[i,k]*X[k]
               else
                     A[i,k]*X[k] + T[k-1]
               endif
         endfor
   endall
```

In this nested mixed **forall** expression, the body expression is a **for-construct** expression which computes one vector-vector inner product as an element for the result array y. Note that the last column of Y is the result vector.[1]

We can interchange the order of the outer and inner expressions: the program after the loop interchange transformation is shown below.

```
YT =
   for    k from 1 to p
          T from array_empty
   construct
          forall i in [1,m]
          construct
                if k = 1 then A[i,k]*X[k]
                else
                      A[i,k]*X[k] + T[k-1,i]
                endif
          endall
   endfor
```

The inner **forall** expression can be mapped by the basic mapping scheme. It computes the

---

1. For convenience, we use **for-construct** as the inner loop expression which makes Y (so is YT below) appeared to be a two-dimensional array. In practice, a **for-iter** expression may be used and the same principle can be applied for its mapping.

m elements of one row of YT in parallel, where an element YT[k,i] is the kth partial sum of the inner product for the i-th row of A and the vector X. Thus, the p-th row of YT is the result vector y. After the loop interchange, the result array is transposed. The inner **forall** expression can still be mapped using the basic mapping scheme, as can the outer **for-construct** expression. However, care should be taken that the elements of A are used one column at a time in pipelined fashion by the inner **forall** expression. For convenience, we can also directly use the transposed array AT for A in the interchanged program, as illustrated below.

```
YT =
   for   k from 1 to p
         T from array_empty
   construct
         forall i in [1,m]
         construct
               if k = 1 then AT[k,i]*X[k]
               else
                     AT[k,i]*X[k] + T[k-1,i]
               endif
         endall
   endfor
```

Now let us completely unfold the outer loop. This can be done according to the principles outlined in Section 10.2.1. The result graph of the mapping is shown in Figure 10.14, with each dashed-line box being one copy of the unfolded body expression. For the purpose of uniformity, the first copy also includes an addition actor with the constant zero as one of its operands. Note that the entire graph is acyclic and is maximally pipelined.

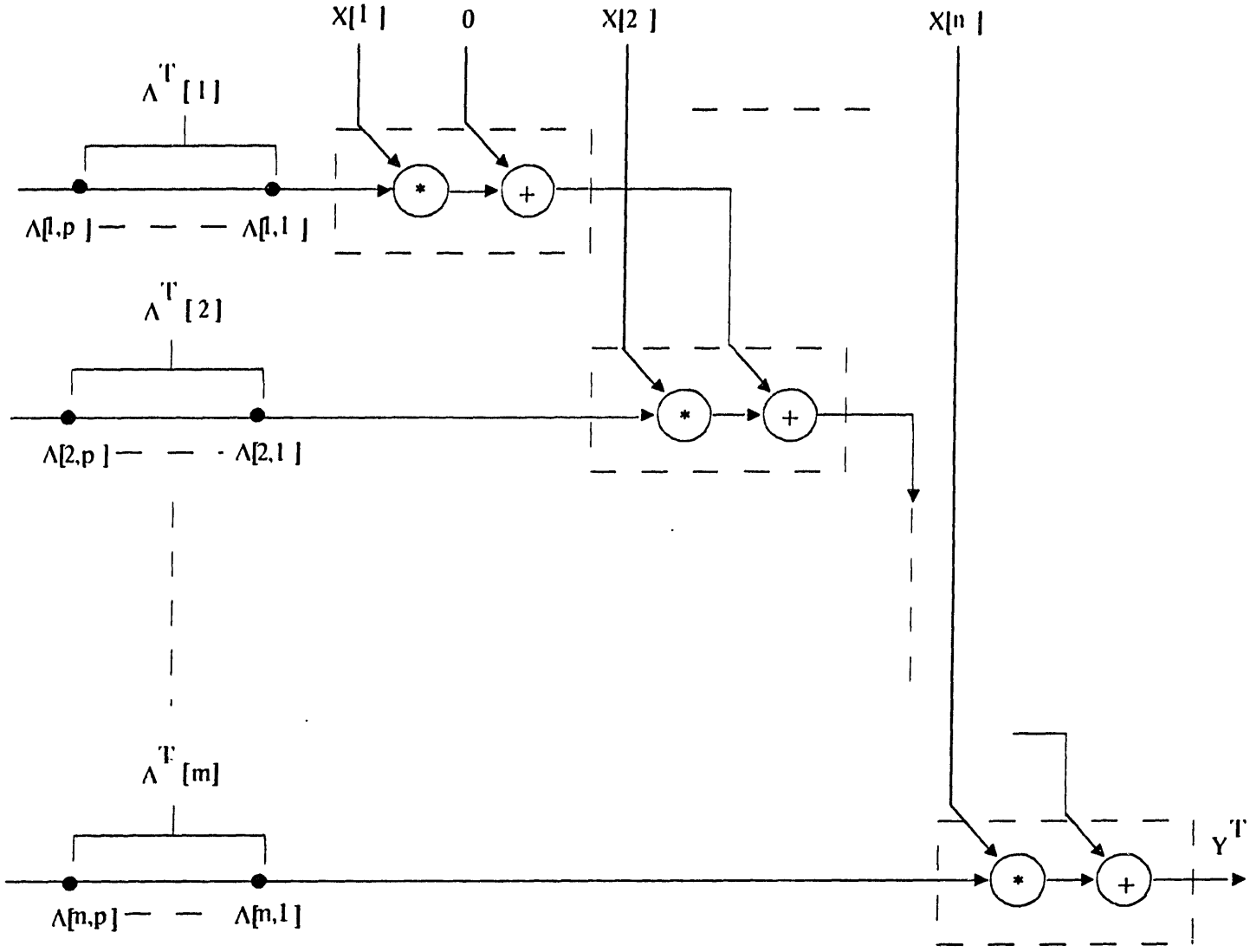


Figure 10.14. Matrix-vector multiplication by loop interchanging and unfolding

10.3 Multiple Pipelines

Let us reconsider the nested mixed expression at the beginning of Section 10.2.2. If the index limits of the outer **forall** expression are known, we can generate m copies of the graphs for the body expression (the inner loop), one for each index i, and let them run in parallel. This mapping scheme is illustrated in Figure 10.15. Each box labeled f in the diagram denotes a graph for an innermost **for-construct** expression corresponding to a particular index i, generated by the basic mapping scheme. Therefore, each box will compute one row of the two-dimensional array x in a pipelined fashion. Using multiple pipelines will increase the parallelism by m fold. Thus, in cases where the graph of an innermost **for-construct** expression consists of a loop with a long critical length, the multiple pipelined solution should be favorably considered.

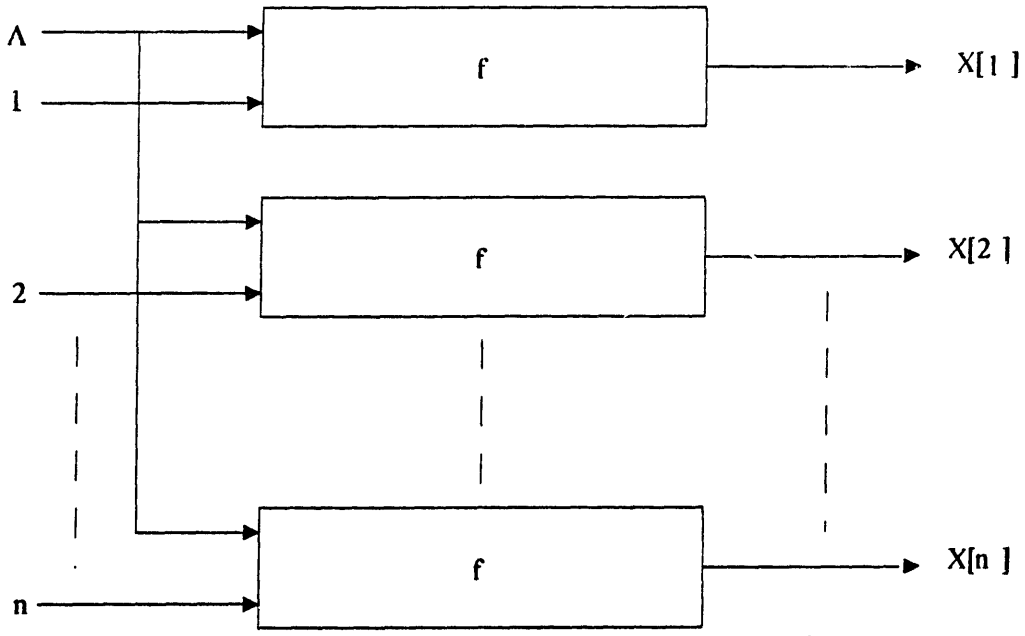


Figure 10.15. A mutiple pipelined mapping scheme

To assemble x[1], x[2],...x[m] coming out of the multiple pipelines into one result array, an extra subgraph may be needed. The structure and function of such a subgraph depend on how the result array is to be expressed and used by the succeeding code blocks.

11. Considerations in Program Structure, Compilation and Machine Design

In this chapter we address certain issues in program structure, compilation techniques and machine design which are important for the efficient support of pipelining on static data flow computers. In Sections 10.1 and 10.2, we outline interesting and challenging problems in program structure analysis as future research topics. In Section 10.3, we discuss briefly some pragmatic aspects in compiler design for data flow computers, presenting an outline of a compiler structure in which the pipelined code mapping scheme can be incorporated. In Section 10.4 to 10.6, we discuss certain issues in machine architecture design. Of course, a comprehensive discussion of data flow computer design is beyond the scope of this thesis. We will concentrate on a few points in the instruction set design that directly relate to our pipelined code mapping scheme.

11.1 An Overview of Program Structure Analysis

To make effective use of the pipelined code mapping scheme developed in this thesis, a compiler needs information concerning overall program structure as well as the structure of each code block. A coherent code mapping strategy will produce a machine code structure that will fully and efficiently utilize the computation power of the machine.

In terms of pipelined code mapping strategy, we assume that the structure of the source program is expressed by a *program block graph* (PBG). A program block graph can be considered as a digraph where a node denotes a program block which defines a new array from some input arrays generated by other nodes. The directed arcs, usually labeled, denote array dependencies, i.e., an arc with label x directed from node C1 to node C2 denotes the fact that an array X defined by C1 is referenced by C2. The array X is called the *input array* of C2, and the *output array* of C1. The structure of a code block is specified as a PIPVAL array creation expression, and we are mainly interested in the few types of code blocks defined in Chapter 4. There are some attributes associated with each code

block in the PBG which are used to specify certain information regarding both the structure of the block itself and the nature of its input/output links. This becomes very valuable in making decisions about the mapping strategy for each individual block.

The major portion of this thesis is devoted to the development of the basic pipelined mapping schemes for the code blocks. The precise notations of the PBG and the set of attributes, are not our major concern. We assume, that a compiler will appropriately analyze the user program and generate a PBG together with specifications of the code blocks.

In terms of the PBG and PIPVAL representation, a program structure analysis should concern with the following:

1. The mapping strategy of each code block in the PBG: should it be mapped directly by the basic pipelined mapping scheme or restructured for optimization (for example the optimization surveyed in Chapter 10)? what optimization techniques would be appropriate, in terms of the time/space tradeoff?
2. The suitable form of the result array produced by a code block: should its representation be flattened — pipelined, parallel or mixed flattened? what is the preferable generation order when pipelined?
3. The suitable form of a link between a pair of nodes in the PBG: Should it be implemented without using array memory? Should it be implemented through FIFOs, and if so, what is the proper size and structure of the FIFOs?
4. A projection of the computational resources needed for code blocks and links in the PBG.

In order to perform a program structure analysis specific information for each code block is needed. We have already conducted extensive analysis of the relation between code mapping schemes and the structure of a code block. Now we summarize the key attributes associated with a code block:

1. The type of the code block: Is it a **forall**, a **for-construct** or a mixed code block? If one of the first two types, is it primitive? If mixed, is the innermost one primitive?
2. The features of a code block: including the number of its levels and the selection order of each input array; Is the body of each level in standard case-1 form? Does the code block have a consistent order for each input array? If so, how does the order relate to the major generation order for the result array of the code block?
3. The parameters of array index ranges: what are the index limits and the subrange limits of index range-partitioning conditional expressions at each level (if applicable)?
4. The computation size of a code block in terms of the number of scalar and array operations: How many of iterations occur in a **for-iter** expression? The fraction of times each arm of the conditional expression will be evaluated.

The global structure of a program is essentially embedded in the PBG. It is important to consider whether the PBG is acyclic and whether the arrays generated by the code blocks have compile-time computable sizes.

If the program itself does not provide all the necessary information, the compiler should allow the user to provide information through an appropriate channel. Successful implementation of a pipelined code mapping scheme relies heavily on available information and program structure analysis based on such information.

In addition to the above information, knowledge about machine architecture is also important in deriving a pipelined mapping strategy from the program structure analysis. It is beyond the scope of this thesis to discuss this in detail.

11.2 Considerations in Analyzing a Certain Class of Programs

In the last section, we outlined the general objectives of program structure analysis and indicated the information necessary to support such analysis. Recall that our pipelined code mapping schemes are designed to work most effectively for a certain class of programs. In this section, the issues of program structure analysis for this class of programs are studied.

We start with discussion of a class of programs with very simple structure, a class that includes the computationally intensive parts of certain numerical application programs. We then extend this class in several ways, and predict what impact these extensions will have. Our goal is to list the issues and problems which must be considered and solved in such analysis, and to predict possible good solutions. Formulation of effective solutions to these problems is beyond the scope of this thesis.

We have observed that the computationally intensive part of a scientific numerical program is usually formed by one or a few clusters of acyclic connected code blocks. Within the same cluster, the arrays produced by all code blocks have the same dimension and size. Such a cluster of blocks often makes up most of the body of some outer loop [31,35]. For the purpose of this discussion, however, we concentrate on the acyclic part of the PBG. This restriction considerably simplifies the problem, yet still covers the most interesting portion of the computation. The conclusions derived from our study can be very useful in the analysis of the entire PBG, with outermost loops included.

We further assume that: (1) all arrays have compile-time manifest constants as their index limits; and (2) each code block has consistent selection orders for each of its input arrays. The above assumptions are typically met by the kernel code of the several benchmark numerical applications studied by the Computational Structures Group at MIT.

11.2.1 A Cluster consisting entirely of Primitive forall Blocks

The simplest situation, when all code blocks in a PBG are primitive **forall** expressions, is shown in Figure 11.1 (a). In this cluster of 5 code blocks (labeled C1 - C5), each node denotes a **forall** expression.

In order to study the issues related to the generation and selection orders for array values, we adopt the following notation. If the result array of a code block is generated in

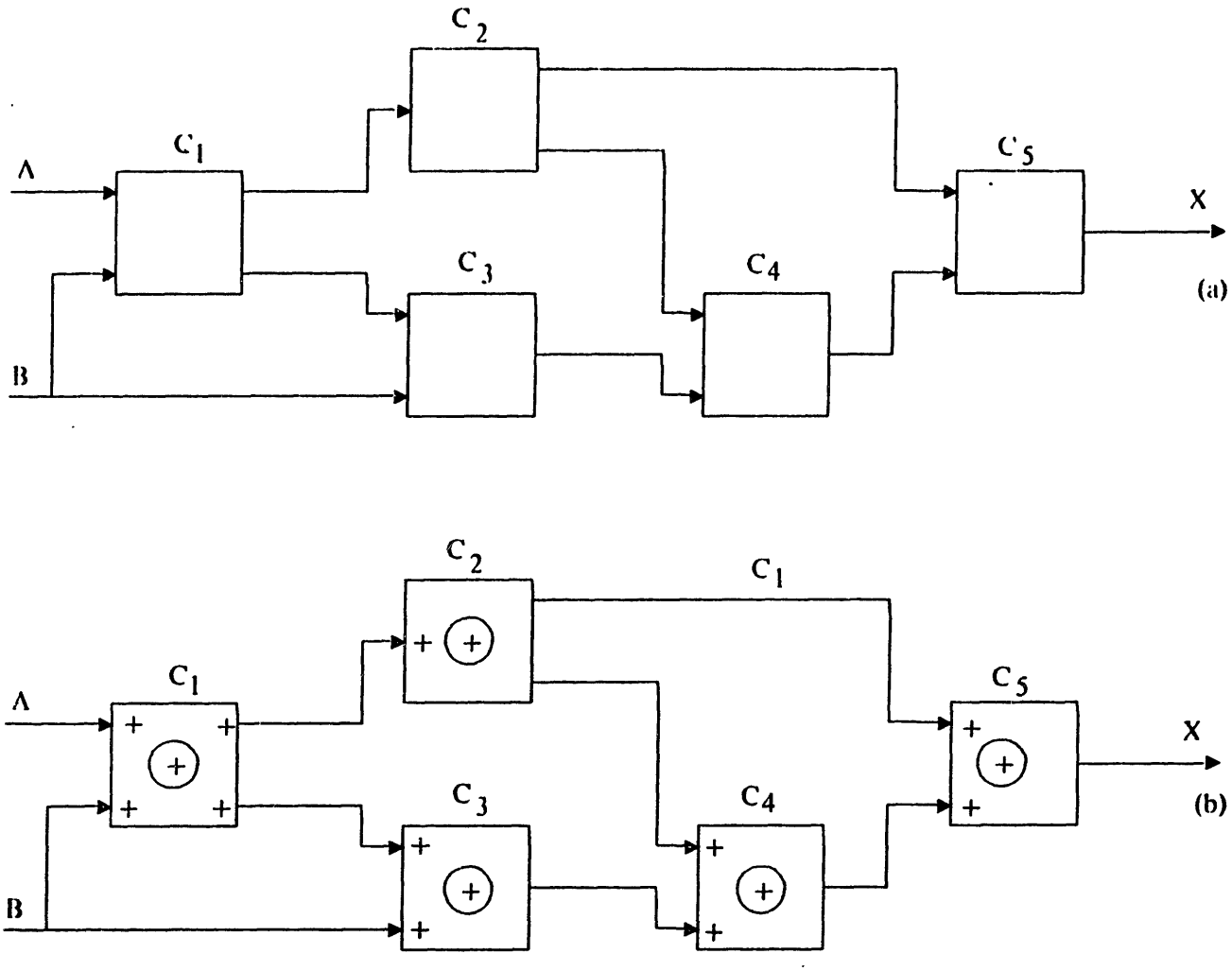


Figure 11.1. An acyclic code block cluster -- example 1

major normal generation order, a "+" sign, enclosed by a circle, is placed inside the box of the code block. A plain "+" sign placed next to an input port in the box of a code block denotes the fact the selection order of the corresponding input array is in major normal order. A similar role is played by the "-" sign except that it denotes a major reverse order.

1. *Situation 1*

Figure 11.1 (b) shows the use of above notation in Figure 11.1 (a), where the selection orders of each input array of a code block are all consistent with the generation order of the result array of the block. This is the first and simplest situation to be considered. It is easy to see that, with the assignment of the potential generation orders indicated in the figure, all arrays are consumed in the same order in which they were generated. Thus, there is no need to store the arrays in array memory. In fact, our pipelined code mapping rules and the optimization procedures can be applied to generate a data flow graph for each code block in which all array operations are replaced by ordinary graph actors.

Since in this example all code blocks are primitive **forall** expressions, the entire code consists of an acyclic data flow graph with ordinary graph actors. Thus, it can be transformed into maximally pipelined data flow graphs. FIFOs are often needed on the arcs linking code blocks, and their sizes can be computed using balancing techniques.

2. *Situation 2*

Figure 11.2 (a) illustrates a situation where the selection orders of an input array of a code block may be different from its generation order. In particular, this happens to both input arrays of code block C3, while it only happens to one input array of C4. Obviously, with the assigned generation orders, three internal arrays have to be stored in array memory as indicated by the "//" signs on the corresponding arcs.

Some improvements can be achieved by proper adjustment of the generation order of code blocks C3 and C4, as shown in Figure 11.2 (b). As a result, one internal array must be

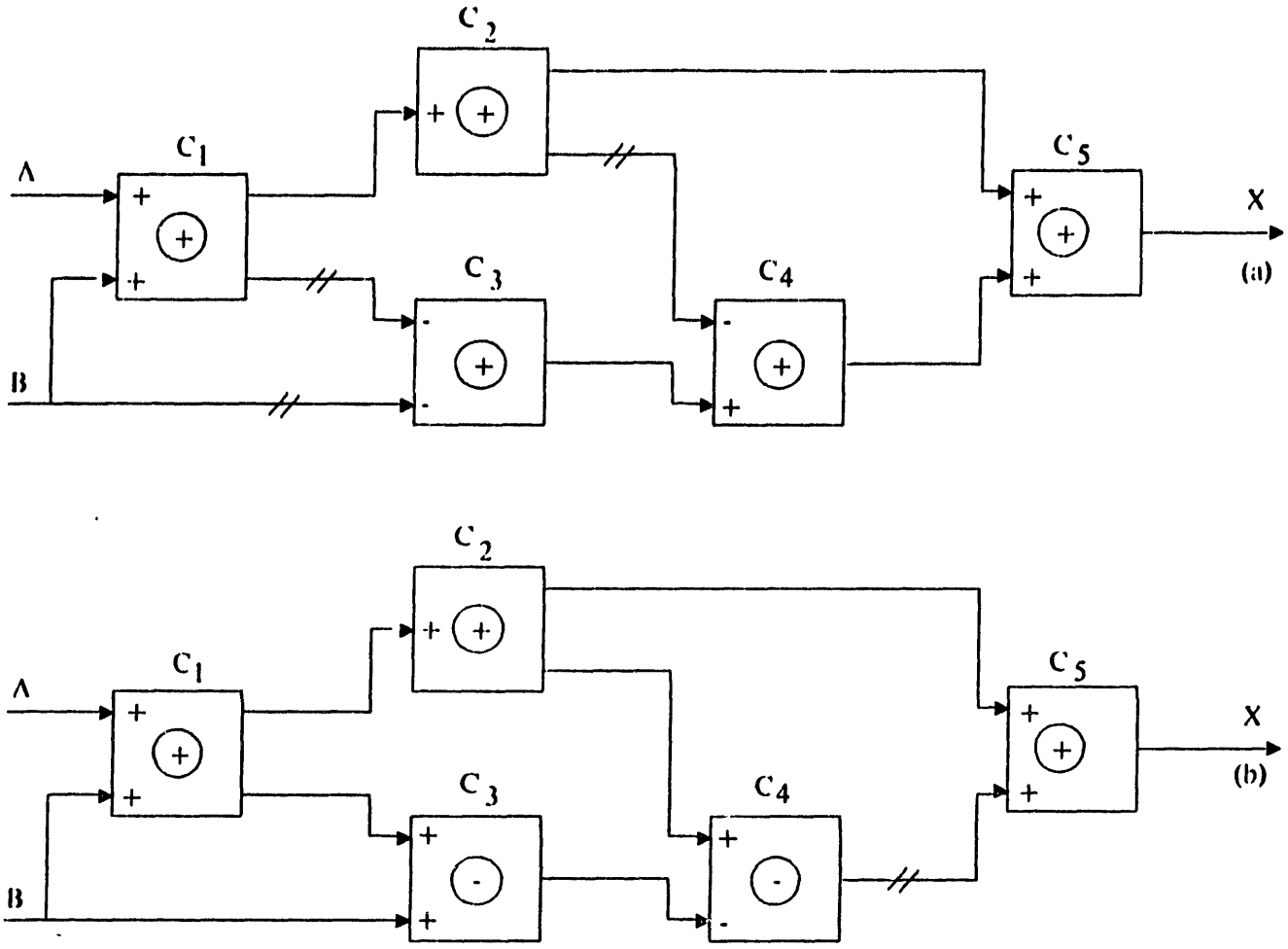


Figure 11.2. An acyclic code block clusters -- example 2

stored in the memory before it can be used.

11.2.2 Some Nodes are Primitive for-construct Expressions

Now we extend the above class of programs such that some nodes may be primitive **for-construct** expressions. In this situation each **for-construct** expression demands a certain generation order for its result array, and thus also a certain selection order for its input arrays. Such order is usually determined by the data dependencies among the elements of

the array to be computed.

Let us consider the example of a PBG shown in Figure 11.3 (a). It is similar to Figure 11.2 (a) except that C4 is a for-construct expression, specially marked by an "*". Given a similar initial assignment of generation orders, there are three internal arrays which must be stored in memory. However, since the generation order of C4 is fixed, we can only adjust C3. The result is shown in Figure 11.3 (b), where two internal arrays need to be stored in memory.

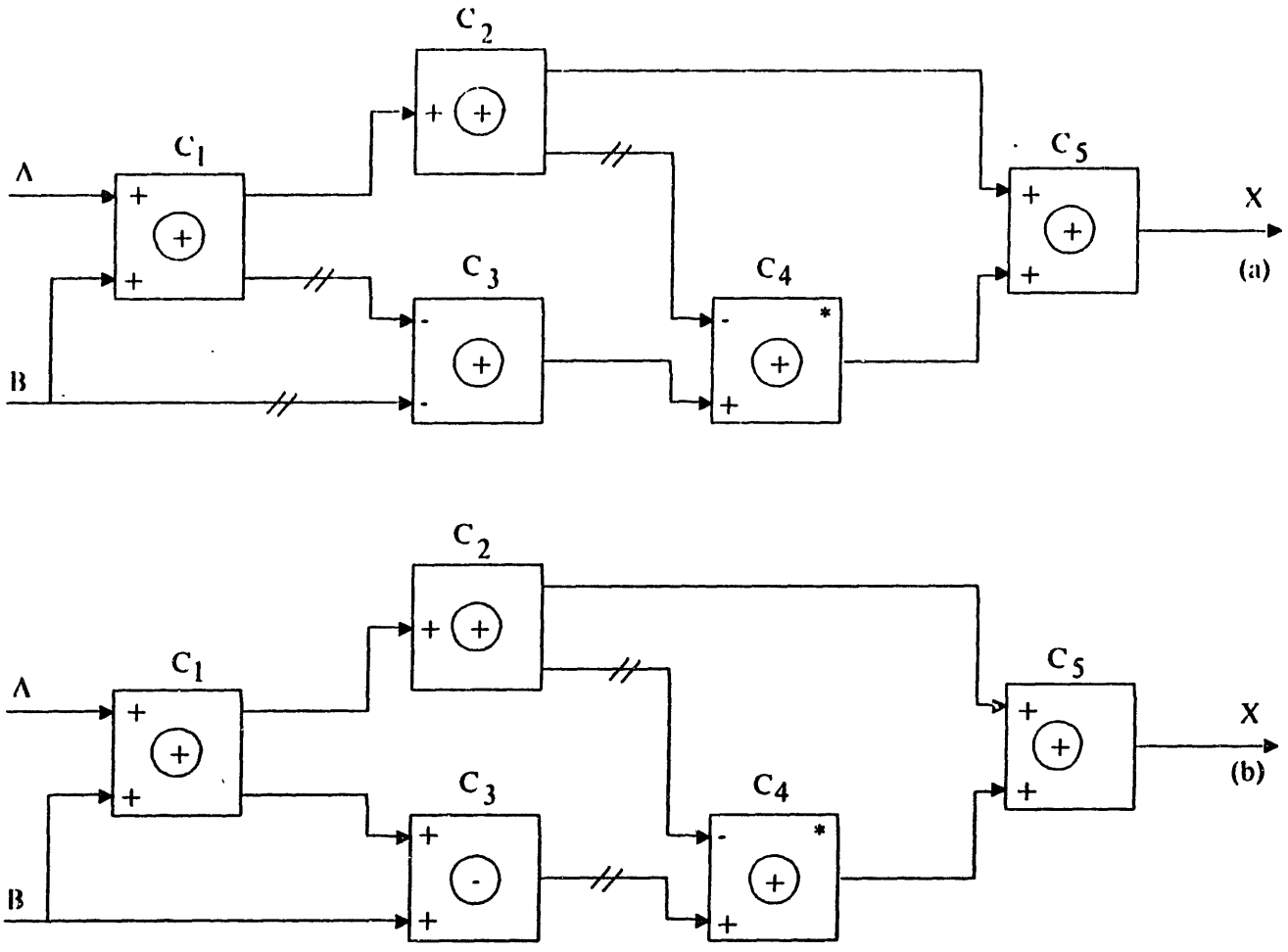


Figure 11.3. An acyclic code block clusters -- example 3

An ideal situation exists when the assignment of generation orders to the code blocks is such that all arrays are generated and consumed in the same order. In such a situation, all array actors can be removed and the entire graph can run in a pipelined fashion without use of any array operations. However it still may not be possible to run the graph in a maximally pipelined fashion due to the cycles introduced in the subgraphs of **for-construct** expressions.

11.2.3 Remarks

The structure of the class of source programs discussed in this section is quite simple; hence an efficient algorithmic approach to perform the above analysis seems quite possible. While it is beyond the scope of this thesis to develop the algorithms, such work would make an interesting topic for future research.

Further extensions of this class of programs might cover the following situations: a PBG with nodes which may have inconsistent selection orders for one of its input arrays; index limits of arrays which are not the same for different code blocks; and an acyclic-connected code block cluster which is enclosed by an outer loop.

Finally, it is interesting to incorporate the special optimization techniques discussed in Chapter 10. These techniques often become important when, after the basic mapping scheme is applied, there exist particular code blocks — usually with **for-construct** expressions as their innermost expression — where the long critical path in their innermost loops seriously degrades the pipelined performance of the entire graph.

11.3 Pragmatic Aspects of Compiler Construction

A primary goal of studying the code mapping scheme is to form a basis on which an effective program transforming compiler can be built. In this section, we briefly address some pragmatic aspects of compiler design for data flow computers. The successful construction of a compiler is a complicated process which involves many different

disciplines. For our purpose, we only discuss a few issues which are important for the implementation of the pipelined code mapping strategy developed in this thesis. We assume that the readers are familiar with the structure of compilers for a conventional computer. Our focus is to outline the important differences which must be made when a data flow compiler is constructed.

In a conventional optimizing compiler for sequential machine, a great deal of optimization effort is spent on local optimization, such as *strength reduction, common subexpression elimination,* etc. Another sort of optimization is performed on sections of code, and is often concerned with speeding up loops. A typical loop improvement is *loop invariant motion* and *code avoidance.* For example, loop invariant movement is to move a computation that produces the same result each time around the loop to a point in the program (often before the entry of the loop) so that it is computed only once. Usually such optimization will improve the speed of the code and reduce its size. These optimizations, although they involve control and data flow analysis of loops, do not address the issue of producing the overall machine code structure that best exploits the parallelism in the program.

Most vectorizing compilers for pipelined vector processors perform sophisticated parallelism detection on user programs. The optimization is focused on the maximal vectorization of each (nested) loop. As outlined in the introduction of this thesis, after the parallelism for each loop is detected, the vectorizing compiler faces the problem of mapping the vector operations onto the von Neumann architecture. The intermediate languages and machine languages reflect the sequential nature of the machine architecture, as well as the imperative model of computing [13]. Central to an imperative model of computing is the concept of a state, which encompasses the program counter, the stored values of registers and storage locations, etc. Two separate pieces of code may share the same locations (variables) and it is difficult characterize parallel execution when several independent loci of execution can have side-effects on each other. Therefore, global

program transformation and optimization are more difficult because the imperative model does not lend itself to easy and clear functional characterization. We have already mentioned the difficulty of scheduling multiple vector operations on multiple hardware pipelines. A compiler may need other sophisticated techniques to perform optimizations between a group of vector operations as in vector chaining and vector register allocation. Such optimization is still "local" since it takes little consideration of the overall data flow and load balancing based on global program structure.

A most important feature of a data flow compiler is it emphasizes the role of overall program structure and the strategy of global optimization of machine code structure. The data flow compiler should include a stage where global program mapping and transformation strategy are determined. More specifically, an important section of the compiler should be devoted to the program structure analysis and the way to implement the pipelined code mapping strategy.

A preliminary view of a data flow compiler is shown in Figure 11.4. The front end performs conventional functions such as syntax analysis, static semantic checking, etc. Besides producing an intermediate form of the source program (which should retain enough structure information about the source program), it also gathers the information about the program structure needed for the analysis and strategy decision processes.

The Analyze and Map modules are the core parts for program transformation and optimization. A discussion of what may be done by the Analyze module is outlined in the previous two sections of this chapter. The mapping schemes defined in Chapter 5-9 form the foundation of the Map module. The optimization techniques for further exploiting parallelism and improving performance of pipelining (such as those discussed in Chapter 10) should also be considered and applied in the Analyze and Map modules. The goal of these two modules is to establish a proper mapping strategy for each component (code block) of the program. The generated machine code should run with high throughput, while achieving balanced utilization of computational resources. The later can only be

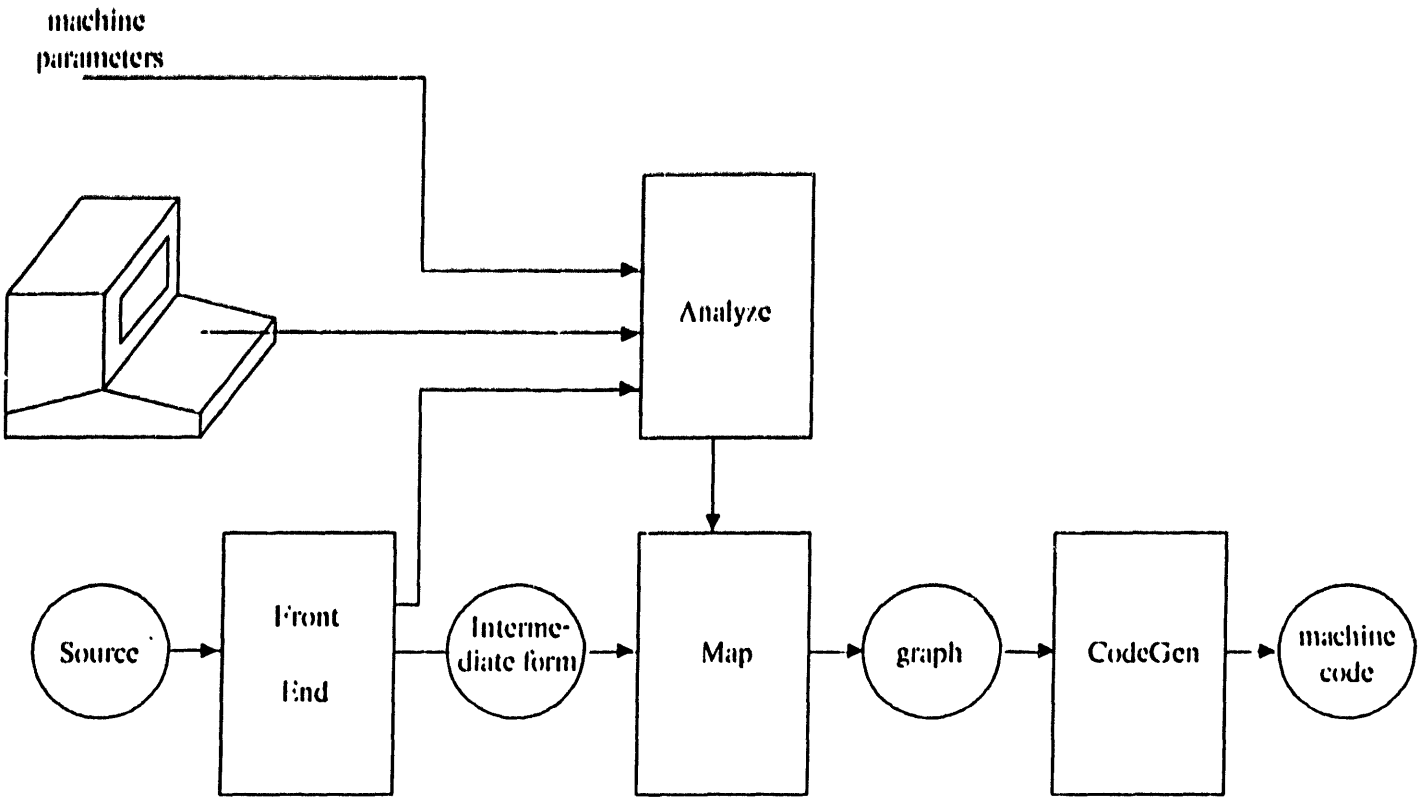


Figure 11.4. The Structure of a Dataflow Compiler

achieved if certain machine parameters are provided to the compiler, which we will not explore in detail.

We should note that the structural description of the source program may not contain all information required for carrying out the necessary analysis. The compiler should allow an interaction channel be established such that the user may supply some additional information, as illustrated in Figure 11.4. Interactive techniques have also been adopted to vectorizing compilers for conventional machines [80]

Some conventional optimization techniques can also be applied in dataflow compilers. For example, optimization techniques similar to common subexpression elimination, constant folding, dead code avoidance, and loop invariant motion should be

considered and implemented in the Analyze and Map modules. The detailed form of such optimization techniques and their position in the modules are beyond the scope of this thesis.

The Analyze and Map modules will produce data flow graphs with basic structure very much like that of the final machine code. These may be processed by another phase of the compiler — Codegen, which will perform some machine dependent optimization and final code generation.

11.4 Considerations in Instruction Set Design

11.4.1 Instructions for Conditionals

In the discussion of a multi-armed conditional expression, we have introduced two graph actors, MB and MM, which lead to a cleaner representation of the corresponding mapping rule. They become very helpful in the formulation of the mapping rules for array construction expressions, the body of which often consists of such a multi-armed conditional expression.

Let us reconsider the data flow graph in Figure 6.10 and Figure 6.11 for the function of MB and its internal structure (see Section 6.4). The core of MB is a B-gate actor which has one input for each test expression (except the last arm), and one boolean-valued output for each arm as well. It also has an output for an encoded value which is to be connected to the corresponding input of MM. The function of MM is to merge the results from the multiple arms under the control of the encoded values from MB.

Of course, MB (mainly B-gate) and MM can be implemented by subgraphs of several graph actors. We propose, however, that the B-gate and MM actors are directly implemented by graph actors which are supported by special machine instructions. It appears that one difficulty may be the number of inputs and outputs for the two actors depending on the number of arms. However, we are most interested in using them in the

mapping of the range-partitioning expressions found in the body of array construction expressions. The number of arms of such a conditional expression is a small constant (for example less than 10) known at compile-time. The machine instructions for a B-gate actor should have the flexibility to allow a small number of operands. It should also include a proper mechanism, such as some field in the instruction representation, to allow a code generator to set the desired numbers of operands. Since each operand value of a B-gate actor is of boolean data type, it should not be difficult for an instruction to hold multiple operands. The machine also should support the data type for the encoded values needed in B-gate and MM actors.

Efficient implementation of the MB and MM actors may result from active research on a newly proposed static data flow architecture [37]. In the new architecture, the control values and data values are handled in separate processing units and no explicit T/F-gate actors and merge actors are needed. Further discussion of architectural improvements is beyond the scope of this thesis.

11.4.2 Control Value Sequences

In a result graph, after successfully applying optimization procedures, there are often a considerable number of T-gate (F-gate) actors controlled by long boolean-value sequences. They are included to replace array SEL actors, and hence to reduce the number of expensive memory operations. Therefore efficient implementation of the boolean pipelines is important to the performance of the entire program.

A situation which is of particular interest to us is when the pattern of a boolean value sequence is known at compile-time. This happens frequently in real applications. In such a situation, a sequence of boolean values can be "generated" at compile-time, and be packed such that little memory space is needed for storing them. The machine may include a special instruction which can have a packed boolean vector as its operand and generate the necessary control value sequence for the corresponding T-gate (F-gate). This can be

done locally to avoid the overhead of sending boolean values over long distances.

11.4.3 The IGEN Instruction

The IGEN actor, a frequently used graph actor throughout this thesis, acts as an index generator which generates a sequence of index values within a specific range. Its function is specified by the subgraph shown in Figure 9.2 (see Chapter 9). The subgraph consists of a loop of four actors. We suggest that the machine instruction set should provide a single instruction to perform the function of the above IGEN subgraph. The implementation details of such an instruction are beyond the scope of this thesis.

11.5 Considerations in Machine Support of Array Operations

Although we are mainly focused on the class of programs where array operations can be removed by the optimization procedures, the machine should also provide efficient support of array operations where they are needed. In this section, we outline additional graph transformation techniques which will assist achieving this goal and mention several instruction set design issues.

11.5.1 Flattening of Arrays

The model of multi-dimensional arrays we use in presenting the basic mapping schemes is the "vector of vector" model, and we have seen its advantages in presenting the basic mapping rules. From the standpoint of machine implementation of array operations, flattening multi-dimensional arrays is more convenient. An important benefit of flattening, among others, is that no array values (descriptors) are stored in the array memory, hence eliminating the associated overhead of manipulating them -- a painful job in an applicative system.

An r-dimensional array selection operation is first mapped into a series of r SEL

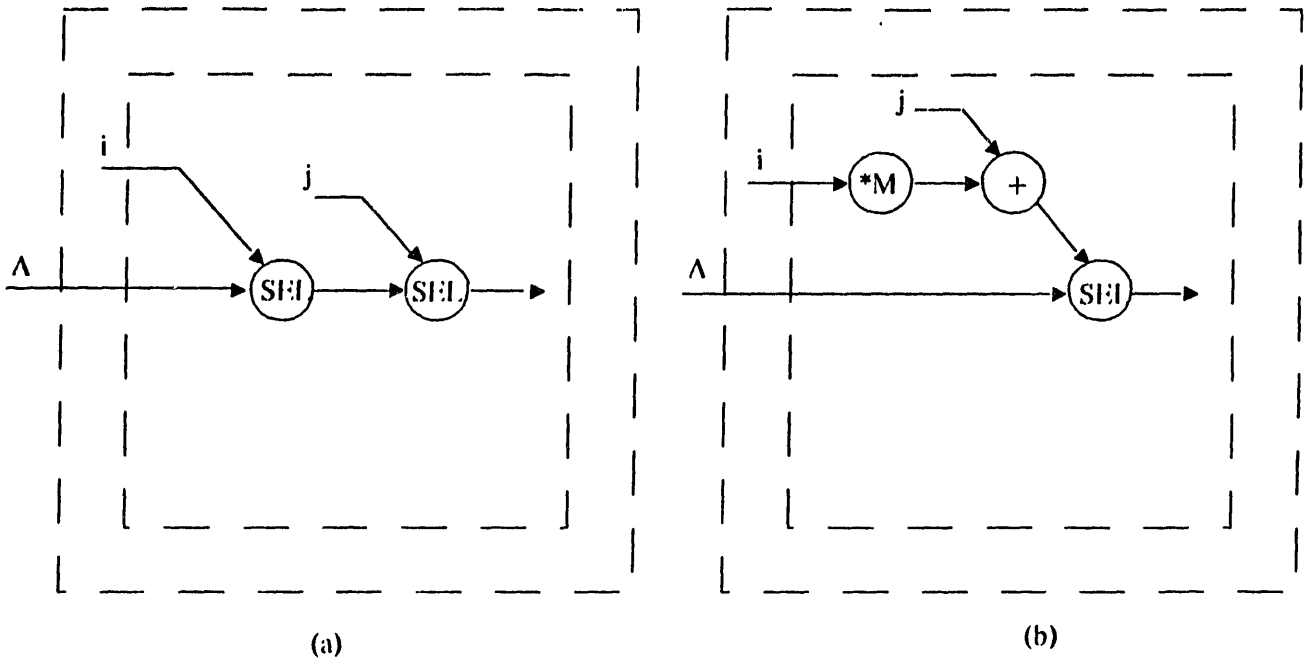


Figure 11.5. The flattening of an array -- a selection operation

actors using the basic mapping scheme. Let us consider a m×n 2-dimensional array A. The graph for A[i,j] is shown in Figure 11.5 (a), where A[i,j] resides in the body of a 2-level nested code block. When A is flattened in the so-called "last index varying most rapidly" order, A[i,j] becomes A[i*m+j], with A becoming a one-dimensional array of m×n elements. The results of flattening are shown in Figure 11.5 (b), where the number of SEL actors is reduced to one.

Next comes the problem of flattening the AGEN actors. Consider again a 2-level nested array creation expressions with the index value names i, j, corresponding to the level-1 and level-2 respectively. A direct application of the basic mapping rule will generate two AGENs in series as shown in Figure 11.6 (a). Recall from Chapter 9 that this implies the use of two append actors. When the result array X is flattened, it becomes one linear array of m×n elements. Hence, only one append actor is required, and the index

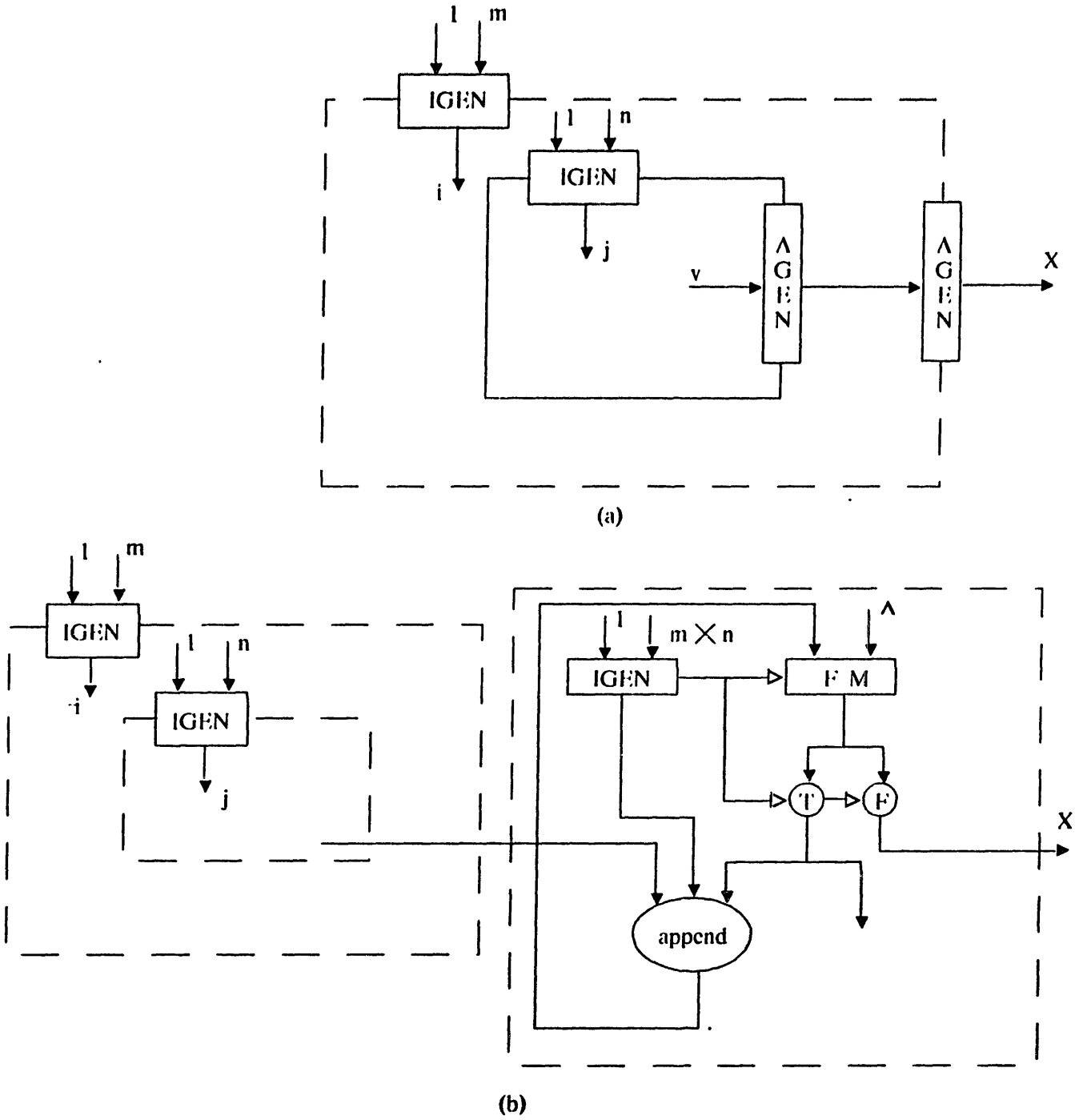


Figure 11.6. The flattening of an array -- an array generation subgraph

value to the append should be adjusted as it is for the flattening of the selection operations. This can be performed by combining the two AGENs. Figure 11.6 (b) shows a result of the flattening where only one array append actor is used. Furthermore, the graph for generating the result array is encapsulated as a separate subgraph (on the right) which is very similar to the structure of an AGEN subgraph in a Figure 9.2 (see Chapter 9). The principle outlined above can also be easily extended to the flattening of a multi-dimensional array.

11.5.2 Flattening And Pipelining

In this section we explore the power of array flattening by combining it with the optimization techniques developed in this thesis.

Let us consider a 2-dimensional primitive **forall (for-construct)** code block which has

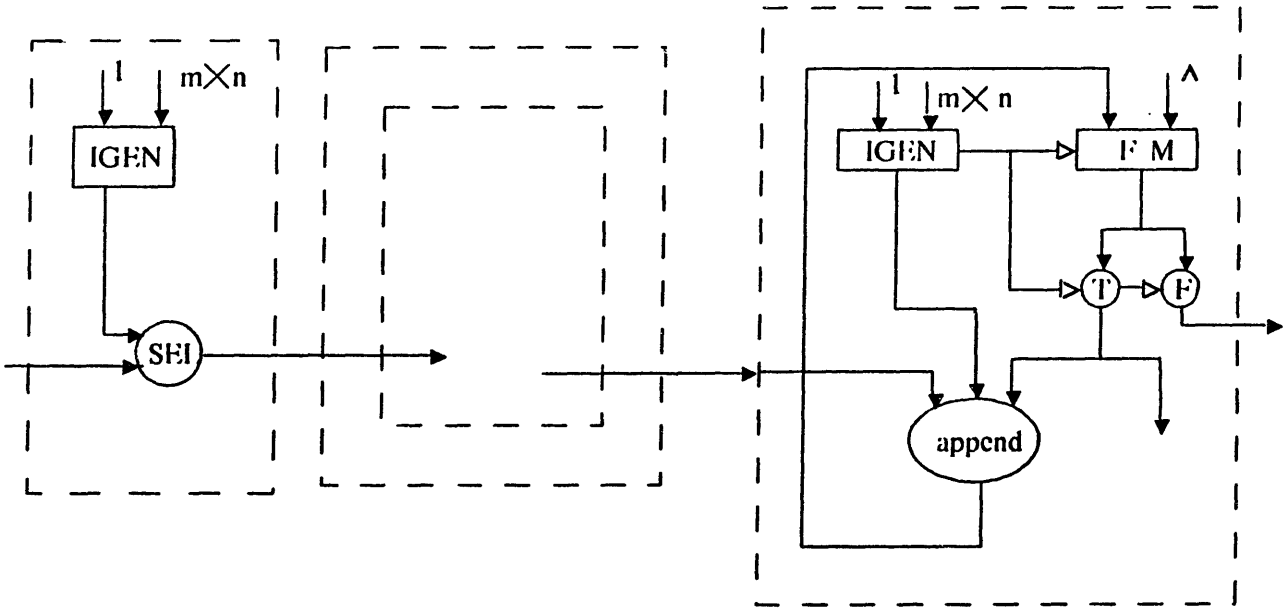


Figure 11.7. Using flattened techniques for pipelining

an input array A and a result array X. Assume that a structure analysis of the PBG has determined that A and X must be stored in AM. If the code block has a considerable number of array selection operations we may still want to perform the optimization procedure to remove the array selection operations. We also may want to remove the two AGENs. The only change is that two additional subgraphs should be provided: one for reading the sequence of elements of array A from the memory, and the other for assembling the sequence of elements of the result array X and putting them in the memory.

The flattening techniques in the last section can be extended to perform the functions of the extra subgraphs described above. The approach is illustrated in Figure 11.7 (a) and (b). The subgraph on the left is for accessing array A, and the subgraph on the right is for assembling the element values and storing them as the flattened result array X.

The advantage of combining the array flattening and the optimization techniques for pipelining are obvious: it may reduce considerably the total number of array operations in the graph, on top of the other benefits for both pipelining and flattening.

These input and output subgraphs for array flattening are similar to the array "unpacker" and "packer" in [6], but used for a different purpose. The target computer should be able to support these operations efficiently.

11.5.3 Array Memory Management in a Target Machine

Supporting array operations in target computers always presents a tradeoff between generality and efficiency considerations. This is true both for conventional computers and data flow computers. Memory mechanisms suitable for data flow computers have been studied by researchers, and the generality/efficiency tradeoffs have been addressed in [3], and most recently in [6].

For the purpose of this discussion, we are not interested in mechanisms for supporting dynamic arrays in the general sense, i.e., arrays having dynamic bounds. In fact we are mostly interested in static memory allocation for arrays.

Through the program structure analysis outlined in the last chapter, the memory usage of the arrays (in the class of programs we are interested in) should be determined: thus a static allocation would be a favored approach. We expect that most of the array actors — SELs and appends — reside in standard subgraphs such as the array packers and unpackers described earlier. Since the arrays are all fully flattened, the SEL and append can be directly implemented by ordinary array memory indexed read/write operations. The program should be partitioned such that each such packer/unpacker subgraph will be accessing the array in some local AM module. Since the size of such an array is known at compile-time, and remains constant during its entire life, the AM memory modules do not need to support the general dynamic allocation and management mechanisms.

If a need does occur, the locality and regularity of array operations in the packer/unpacker subgraphs can be used to implement a memory management scheme with certain dynamic features. For example, we can use a simplified reference count scheme to manage the blocks of storage for arrays (as described in [6]). The key observation is that an array is always generated by a packer subgraph and used by one or several unpacker subgraphs known at compile-time. We use the number of unpackers as global reference count for the array. Both the packer and the unpacker subgraphs are accessing the array in a regular fashion and thus do not change the reference count while they are running. After a packer is done, the array is generated and ready to be used, so the global reference count is set accordingly. After an unpacker is done, the array A is no longer being used by the corresponding code block, hence the reference count can be reduced by 1. Thus, instead of adjusting the reference count each time an array reference is made (as in the conventional reference count scheme), the global reference count is updated only when a packer or unpacker subgraph is terminated. Of course, we may include extra data flow graphs for each array to perform the reference counting. Each AM module should also support dynamic-allocations of memory blocks when an array token is generated. Since the block size is known and remains constant, the memory management scheme should be much

simpler than a general dynamic scheme.

There are also arrays which are implemented through large FIFOs. These arrays may also be stored in the array memory, and we discuss them in the next section.

11.6 FIFO Implementation

FIFOs are used extensively in the data flow graph of a code block or on links between code blocks as required by balancing or array skewing. Therefore, efficient support of FIFOs should be considered in the instruction set design for a static data flow computer. The function of a FIFO with fixed size, known as a *fix-sized* FIFO, is equivalent to a chain of k identity actors. FIFOs may indeed be implemented by ID actors, but it is quite expensive, in particular for long FIFOs. Data flow implementation of FIFOs has also been studied and discussed earlier, for example, in [43,35].

We propose that the instruction set of a static data flow machine should provide adequate mechanisms to directly implement the FIFO function. The instruction set may include a dedicated FIFO instruction to support the *fix-sized* FIFO. For short FIFOs, the memory space can be directly allocated in the program memory in the PE. If the operands (data) are stored in a separate memory, the FIFO may use the space in that memory as well. For long FIFOs used between code blocks, we should consider employing array memory as the primary FIFO storage space. In such cases, a FIFO usually uses blocks of storage — for example one or several rows of an array. Such a scheme was described in [43].

The flexible-sized FIFO is more or less like a *ring buffer* in the conventional computer. As long as a size limit is known (the case we are interested in), there should be no difficulty in implementing such a FIFO. For reasons of efficiency, both the PE and the AM modules should provide mechanisms which can efficiently manage the FIFO storage when executing a flexible-sized FIFO instruction. We leave the details of such mechanisms to the designer of the target data flow computers.

12. Conclutions

12.1 A Summary of the Thesis Work

We have developed a pipelined code mapping scheme which may produce machine code structure such that the potential parallelism in the user programs may be effectively exploited by static data flow computers. Our code mapping scheme is particularly suitable for a class of programs frequently found as the kernel of certain scientific numerical application programs. Strong regularity of array operations is a major feature of such programs.

The functional language Val is our choice as a high level language to express user programs. In particular, we define a subset of Val — PIPVAL — to represent the source programs to be handled by the basic mapping rules. An important feature of PIPVAL is that two array creation constructs are provided by the language to express the construction of arrays — the **forall** and **for-construct**. Using these two constructs, array creation operations with the desired regularity can be expressed without using Val array append operations. Other array regularities are also utilized by the code mapping scheme, including the regularity in the form of array selection indexing scheme.

Pipelined code mapping schemes are developed for PIPVAL expressions. The focus is on the mapping rules of array creation constructs, especially the primitive **forall** and **for-construct** expressions. Optimization procedures are presented which can utilize the regularity in array operations as found in these expressions and transform the data flow graphs such that array operations may be effectively removed or replaced by ordinary machine operations. In addition, certain related optimization techniques are discussed which can be used together with the basic mapping scheme to improve the performance of pipelining.

The code mapping scheme developed in the thesis is based on the power of a sufficiently large data flow computer which can effectively exploit the parallelism by means

of the pipelined execution of machine level data flow programs. Certain issues in the architecture design, in particular the instruction design, which are important to support pipelining, are addressed.

12.2 Suggestions for Future Research Topics

We suggest the following research topics which are important in extending the results of this thesis. They include the areas of language design issues, mapping algorithm issues, and issues in compiler construction and machine design.

The generality of user programs which can be effectively handled by pipelined code mapping schemes is one area of substantial room for future research interests. The optimization techniques for array operations depend heavily on their regularities. In terms of array selection operations such as A[exp], what will be the impact if we allow exp to be a more complicated expression than an affine function of the index values? In terms of array construction expressions, what will be the effect if we relax the restriction that the bounds of array must be compile time constants? What changes should be made in the optimization procedures to handle such cases efficiently? How about the situation when the code block does not have a consistent selection order in terms of an input array?

In terms of compiler construction, a number of areas remain to be studied. As indicated in Chapter 11, this includes the development of algorithms which can effectively perform the allocation of the arrays which cannot be implemented simply by FIFO, so array memory is needed. The solution should not only minimize the array memory storage usage but also keep the locality as well as the simplicity of array accessing mechanisms. Although the emphasis should be placed on the kernel of the computation where regularity usually makes the algorithm simple, the compilation scheme should also take into consideration the complexity for other parts of the problems. For example, we may consider the code blocks which do not belong to the several classes studied in the thesis, or program block graphs which are not acyclic.

As indicated in Chapter 10, there are other optimization techniques which can be combined with the pipelined code mapping scheme to improve performance of the resulting object code. How can they be incorporated adequately and effectively in a compiler? One suggestion is that the compiler should provide a tool for performance estimation of the object program to be generated, thus a user may be prompted with some statistics of how well the basic pipelined code mapping scheme will perform. If the result is not satisfactory, he may direct the compiler to perform other transformations using additional optimization techniques. Ideally, the user may also be informed of potential bottleneck program modules. How hard is it to build such a performance analysis tool and how accurate can the performance estimation be? What machine and program parameters will it need? What form of interface should exist between this tool and other parts of the compiler? This is both an interesting and challenging task a compiler implementor must face.

Many architecture design issues should be further studied. An ideal static data flow machine model was adopted for this thesis to provide a simple and clean framework for formal analysis. However, to apply the code mapping scheme successfully, we face pragmatic and problematic issues in real data flow computers. First, parallelism of the machine will have a strong impact on the balancing strategy for data flow programs. We should also consider the variation of execution time between instructions due to the fact that they may perform different operations or due to the processing load fluctuation of the machine and interconnection network. The author believes that the balancing and optimization should achieve the ultimate goal of keeping the processors usefully busy. Therefore the above factors should certainly be considered in both the architecture design as well as compiler implementation. Other issues in instruction set design and machine architecture support, such as those that have been outlined in the previous chapter, should be fully investigated.

While we assume that the machine should have enough parallelism to support high

concurrency for the pipelined execution of data flow programs, no assumption is made about how the code is to be partitioned and allocated to the processing elements. To what extent does the locality of allocation will affect overall performance of programs? It is difficult to evaluate a solution strategy without understanding the nature of the communication cost in terms of the interconnection network architecture, the behavior of the programs and the technology of the hardware modules. Much work remains to be done in these areas.

13. References

[1] Abu-Sufah, Kuck, D. and Lawrie, D., "Automatic Program Transformation for Virtual Memory Computers", Proc. of the 1979 National 1st Computer Conference, June, 1979.

[2] Ackerman, W. B., "A Structure Memory for Data Flow Computers", Technical Report 186, Laboratory for Computer Science, MIT, Cambridge, MA, Aug. 1977.

[3] Ackerman, W. B., "A Structure Processing Facility for Data Flow Computers", Proceeding for the 1978 International Conference of Parallel Processing, Aug. 1978.

[4] 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.

[5] Ackerman, W. B., "Data Flow Languages", AFIPS Proceedings, Vol. 48: Proceedings of the 1979 National Computer Conference, AFIPS, 1979.

[6] Ackerman, W. B, "Efficient Implementation of Applicative Languages", Technical Report 323, Laboratory for Computer Science, MIT, Cambridge, MA, March, 1984.

[7] Adams, D. A., "A Computation Model with Data Flow Sequencing", TR CS-117, School of Humanities and Science, Stanford Univ., Stanford, CA, Dec. 1968.

[8] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., "The Design and Analysis of Computer ALgorithms", Addison-Wesley Publishing Co., 1974.

[9] Agerwala, T. and Arvind, "Special Issue on Data Flow Systems", IEEE Computer, Vol 15. No. 2, Feb. 1982.

[10] Arvind, K. P., Gostelow, K. P. and Plouffe W., "An Asynchronous Programming Language and Computing Machine", Tech. Rep. 114a, Information and Computer Science, University of California, Dec. 1978.

[11] Arvind and Iannucci, R. A., "A Critique of Multiprocessing von Neumman Style" Proceeding of the 10th International Symposium on Computer Architecture", June. 1983.

[12] Arvind et. al. "The Tagged Token Dataflow Architecture", (preliminary version), Lab. for Computer Science, MIT., Cambridge, MA. Aug. 1983.

[13] Backus, J. "Can Programming Be Liberated from the Von Neumann Style? A Function Style and Its Algebra of Programs" CACM, Vol. 21, No. 8, Aug. 1978.

[14] Barnes, G.H. et al, "The ILLIAC IV Computer", IEEE Transaction on Computers, C-17, 8, August, 1968

[15] Bazaraa, M. S. and Jarvis, J. J., "Linear Programming and Network Flows", John Wiley & SOns, Inc., 1977.

[16] Boughton, G. A., "Routing Networks in Packet Communication Architecture", Dept. of Electrical Engineering and Computer Science, S.M. Thesis, June 1978.

[17] Boughton, G. A., "Routing Network in Packet Communication Architecture", Dept. of Electrical Engineering and Computer Science, Ph.D dissertation, LCS TR-341, June 1985.

[18] Bouknight, W. J., Denenberg, S. A., McIntyre, Randall, J. M., Sameh, A. H. and Slotnik, D. L., "The Illiac IV System", Proceeding of the IEEE, Vol. 60, No. 4., April 1972.

[19] Brock, J. D., "Operational Semantics of a Data Flow Language", MIT/LCS/TM-120, Dec. 1978.

[20] Brock, J. D., "Consistent Semantics for a Data Flow Language", Ninth International Symposium on Mathmatical Fundations of Computer Science, Poland, Sept. 1980.

[21] Brock, J D., "A Formal Model of Non-Determinate Dataflow Computation", TR-309, Laboratory for Computer Science, MIT, Cambridge, MA, Aug. 1983.

[22] Control Data Corp., "CDC Cyber 200/Model 205 Technical Description.", St. Paul. Minn. Nov. 1980.

[23] Dantzig, G., "Applicationof the Simplex Methode to a Transportation Problem", in Activity Analysis of Production and Allocation, Ed. T.C. Koopmans, John Wiley & Sons, New York, 1951.

[24] Dennis, J. B. and Fossen, J. B., "Introduction to Data Flow Schemas", Computation Structure Group Memo 81-1, Laboratory for Computer Science, MIT, Cambridge, MA., Sept. 1973.

[25] Dennis, J. B., "First Version of a Data Flow Procedure Language", Lecture Notes in Computer Science, 19, Springer-Verlag, N. Y., 1974, pp 362-376.

[26] 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.

[27] Dennis, J. B., "Packet Communication Architecture", Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing.

[28] Dennis, J. B., "Data Flow Supercomputers", Computers, Vol 13, Nov. 11, Nov. 1980.

[29] Dennis, J. B., Boughton, G. A. and Leung, C. K. C., "Building Blocks for Data Flow Prototypes", Proceeding of the 7th Symposium on Computer Architecture, May 1980.

[30] 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., March, 1982.

[31] Dennis, J. B., "High Speed Data Flow Computer Architecture for the Solution of Navier-Stokes Equations" Computation Structure Group Memo 225, Laboratory for Computer Science, MIT, Cambridge, MA.

[32] Dennis, J. B. and Gao, G. R. "Maximum Pipelining of Array Operations on Static Data Flow Machine", Proceeding of the 1983 International Conference on Parallel Processing, Aug 23-26, 1983.

[33] Dennis, J. B., Willie, Y-P. L. and Ackerman, W. B., "The MIT Data Flow Engineering Model", Proceedings of the IFIP 9th World Computer Congress, Paris, France, Sept. 1983.

[34] Dennis, J. B., "Data Flow for Supercomputers" Proceeding of 1984 Compcon., March, 1984.

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

[36] Dennis, J. B., "Data Flow Models of Computation", Notes from lectures at the International Summer School on Control Flow and Data Flow: *Concepts of Distributed Programming*, Marktoberdorf, Germany, Aug. 1984.

[37] Dennis, J. B., "Data Flow Computation — A Case Study", submitted for publication, Lab. for Computer Science, MIT, Cambridge, MA, Jan. 1986.

[38] Dias, D. M. and Jump, J. R., "Analysis and Simulation of Buffered Delta Networks", IEEE Trans. on Computers, C-30, No. 4., April, 1981.

[39] Dias, D. M. and Jump, J. R. "Packet Switching Interconnection Networks for Modular Systems", Compute, Vol. 14, No. 12, Dec. 1981.

[40] Dijkstra, E. W., "A Note on Two Problems in Connexion with Graphs", Numerische Mathematik 1, 269-271, 1959

[41] Evans, D. J., "Parallel S.O.R. Iterative Methods", Parallel Computing, Vol 1, No. 1, Aug. 1984.

[42] Ford, L. R. and Fulkerson, D. R., "Flow in Networks", Princeton University Press, Princeton, N. J. 1962.

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

[44] Gao, G. R. "Homogeneous Approach of Mapping Data Flow Programs", Proceeding of the 1983 International Conference on Parallel Processing, Aug, 1984.

[45] Gao, G. R. and Theobald, K. "An Enable Memory Controller Chip for A Static Data Flow SuperComputer", CSG Design Note 18, Lab. for Computer Science, M.I.T., Cambridge, MA, Jan. 1985

[46] Gao, G. R. "Maximum Pipelining of Linear Recurrence on Static Data Flow Computers", Computation Structure Group Note 49, Lab. for Computer Science, Aug. 1985.

[47] Gao, G. R, "Massive Fine-Grain Parallelism in Array Computation — a Data Flow Solution", Proceedings of Future Directions of Supercomputer Architecture and Software, Charleston, SC., May, 1986.

[48] Gao, G. R, "A Pipelined Code Mapping Scheme for Solving Tridiagonal Linear System Equations", Proceeding of IFIP Highly Parallel Computer Conference, Nice France, March, 1986.

[49] Gao, G. R., "A Maximally Pipelined Tridiagonal Linear Equation Solver" (revised), To appear on the IEEE Journal of Parallel and Distributed Computing, 1986.

[50] Gao, G. R, "A Stability Classification Method and Its Application to Pipelined Solution of Linear Recurrences", to appear on the Journal of Parallel Computing, North Holland, 1986.

[51] Gajski, D. D., Padua, D. A, Kuck, D. J., "A Second Opinion on Data Flow Machines and Languages", IEEE Computers, Jan 1982.

[52] Gajski, D. D., Kuck, D. J., Lawrie, D. and Sameh, A., "Cedar — a Large Scale Multiprocessor" Proceeding of 1983 ICPP Conference, Aug. 1983.

[53] Gurd, J. R, Kirkham, C. C. and Watson, I., "The Manchester Prototype Dataflow Computer", CACM, Vol. 28, No. 1, Jan. 1985.

[54] Hockney, R. W. and Jesshope, C. R., "Parallel Computers", Adam Hilger Ltd, Bristol.

[55] Hwang, K. and Briggs, F. A., "Computer Architecture and Parallel Processing", McGraw-Hill Book Company, 1984.

[56] Hwang, K. and Su, S. P., "Multitask Scheduling in Vector Supercomputers", TR-EE 83-52, School of Electrical Engineering, Purdue University, Dec. 1983.

[57] Jensen, P. A. and Barnes, J. W, "Network Flow Programming", John Wiley & Cons, Inc., 1980.

[58] Khachian, L. G., Doklady Akademii Nauk USSR, Vol. 244, No. 5, 1979.

[59] Karp, R. M. and Milner, R. E, "Properties of a Model for Parallel Computation: Determinancy, Termination, Queueing", SIAM Journal of Applied Math., Vol 14, Nov. 1966.

[60] Kellr, R. M., Lindstrom, G. and Patil, S. S, "A Loosely-Coupled Applicative Multi-processing System", Proc. of the 1979 National Computer Conference, AFIPS Conference Proceeding 48, June 1979, 613-622.

[61] 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.

[62] kogge, P. M. "Maximum Rate Pipelined Solutions to Recurrence Problems," in Proceedings, 1st Computer Architecture Symposium., Dec. 1973, pp. 71 - 76.

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

[64] Kogge, P. M. "The Architecture of Pipelined Computers", McGraw-Hill Book Company, New York, 1981.

[65] Knuth, D. E, "The Art of Computer Programming", Second Edition, Vol. I: Fundamental Algorithms, 1968.

[66] Kuck, D. J., Muraoka, Y. and Chen, S. C., "On The Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speed-Up", IEEE Trans. on Computers, Vol C-21, No. 12, Dec. 1972.

[67] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", Computing Surveys, Vol. 9., No. 1., March 1977.

[68] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B. and Wolfe, W. "Analysis and Transformation of Programs for Parallel Computation", Proc. of the Fourth International Computer Software & Application Conference", Oct. 1980.

[69] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B. and Wolfe, W. "Dependence Graphs and Compiler Optimizations", Proceedings of the 8th ACM Symposium on Principles of Programming Languages.

[70] Kruskal, C. P. and Snir, M., "The Performance of Multistage Interconnection Networks", IEEE Trans. on Computers, C-32, No. 12, Dec. 1983. Kung, H. T., "Why Systolic Architecture", Computer, Vol. 15, No. 1, Jan. 1982.

[71] Lawler, E., "Combinatorial Optimization Networks and Matroids", Holt, Rinehart and Winston, 1976.

[72] Leiserson, C. E., "Area Efficient VLSI Computation", Ph.D. Dissertation, Dept. of Computer Science, CMU-CS-82-108, Oct. 1981.

[73] Leiserson, C. E., "Optimizing Synchronous Systems", TM-215, Laboratory for Computer Science, MIT, March, 1982.

[74] Leiserson, C. E. and Saxe, J. B., "Optimizing Synchronous Circuitry By Retiming" (Preliminary Version)

[75] Leighton, T., "Parallel Computation Using Meshes of Trees" (Extended Abstract), Math. Dept. and Laboratory of Computer Science, M.I.T., Cambridge, MA, Sept. 1981.

[76] McGraw, J. R., "The Val Language: Description and Analysis", ACM Transaction on Programming Languages and Systems, Vol. 4., No. 1, Jan. 1982.

[77] Milne, G. and Milner, R., "Concurrent Processes and Their Syntax", Journal of the ACM, Vol. 26, No. 2, April. 1979.

[78] Milner, R., "Flowgraphs and Flow Algebras", Journal of ACM, Vol. 26, No. 4., Oct. 1979.

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

[80] Miura, K. and Uchida, K., "FACOM Vector Processor VP-100/VP-200", Proc. Nato Advanced Research WOrkshop on High Speed Computing, Julich, W. Germany, Springer-Verlag, 1983.

[81] Padua, D. A., Kuck, D. J. and Lawrie, D. H. "High Speed Multiprocessors and Compiling Techniques", IEEE Trans. on Computers, Vol. C-29, No. 9, Sept. 1980.

[82] Patil,S. S, "Closure Properties of Interconnections of Determinate Systems", Record of the Project MAC Conference on Concurrent Systems and Parallel computation, 1970.

[83] Moder, J. J. and Phillips, C. R, "Project Management with CPM and PERT", Van Nostrand Reinfold Co., New York, 1970.

[84] Adams, G. B., Brown, R. L. and Denning, P. J., "Report on an Evaluation Study of Data Flow Computation", TR 85.2, Research Institute of Advanced Computer Science, NASA Ames Research Center, April, 1985.

[85] Rodriguez, J. E., "A Graph Model for Parallel Computation", Ph. D. thesis, MIT/LCS/TR-64, Laboratory for Computer Science, MIT, Cambridge, MA, 1969.

[86] Russel, R. M., "The Cray-1 Computer System", CACM, Vol. 21, N0. 1, Jan., 1978.

[87] Stone, H., "Parallel Tridiagnal Equation Solvers", ACM Trans. on Math. Software, Vol. 1, 1975.

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

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

[90] Todd, K. W. "Function Sharing in a Static Data Flow Machine", Proceedings of the 1982 International Conference of Parallel Processing, Aug. 24-27, 1982.

[91] Weng, K. S., "An Abstract Implementation for a Generalized Data Flow Language", TR-228, Laboratory for Computer Science, MIT, Cambridge, MA, Feb. 1980.

[92] Watson, I., and Gurd, J. R., "A Practical Dataflow Computer", COMPUTER, Feb. 1982.

[93] Valiant, L. G. and Brebner, G. J., "Universal Schemes for Parallel Communication", Proc. of the 13th Annual ACM Symposium on Theory of Computing, May 1981.