Compile-Time Scheduling of Digital Signal Processing Data Flow Graphs onto Homogeneous Multiprocessor Systems

Ali Shatnawi

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy at

Concordia University

Montréal, Québec, Canada

April 1996

ISBN 0-612-18441-2

ABSTRACT

Compile-Time Scheduling of Digital Signal Processing Data Flow Graphs onto Homogeneous Multiprocessor Systems

Ali Shatnawi, Ph.D.
Conacordia Unviersity, 1996

The data flow graph (DFG) has proven to be an efficient model for the class of problems not involving data-dependent decision-making operations, such as those in most digital signal processing (DSP) applications. It has the inherent property of exposing the parallelism in the algorithms as it does not impose unnecessary constraints other than those imposed by the data precedency among the operations.

This thesis is concerned with finding a compile-time (static) schedule for DFGs representing DSP algorithms onto multiprocessor systems. It concentrates mainly on producing a rate-optimal schedule that achieves the minimum iteration period known as the iteration period bound. The problem of optimizing the I/O delay is also considered. Regardless of the optimality criteria used, minimizing the number of processors is one of the main concerns.

The problem of scheduling consists of two phases: (1) the time scheduling phase where the tasks of the DFG are scheduled in the time domain such that the precedency constraints are satisfied and the iteration period bound achieved assuming an infinite number of resources, and (2) the processor assignment phase where the time schedule is mapped onto a matrix in the time-processor discrete space.

A combinatorial theory is developed to produce a rate- and delay-optimal time schedule for a fully specified DFG. A rate-optimal time schedule is obtained analytically by a cyclic to acyclic transformation through a sequence of critical-circuit contractions. It is also shown that it is always possible to achieve both the rate and the delay optimality simultaneously, and a technique to ensure it is presented. An algorithm is then presented to map the time-schedule to a processor assignment. Further, a technique is proposed to reduce the number of processors, which often results in achieving the processor bound. Several examples are considered to illustrate the efficiency of the technique proposed. Finally, it is shown that the overall order of time complexity is lower than those of the existing techniques for most applications.

To My Late Mother

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude and heartfelt thanks to my thesis supervisors Drs. M. O. Ahmad and M. N. S. Swamy for their interest, guidance and constructive criticism throughout this work. Their advice, encouragement and close support have been invaluable.

I gratefully acknowledge the support of Jordan University of Science and Technology through a scholarship. This work was also supported in part by NSERC and FCAR through grants awarded to my thesis supervisors.

It is my great pleasure to thank my friends and colleagues at the Centre for Signal Processing and Communications for the many fruitful discussions which we have had during the course of this work.

Special thanks go to my family for their constant support and prayers, without which this work would not be possible. I owe special thanks to my wife, Faten, for her patience, encouragement and love. Her great effort in taking care (the lion share) of our daughter, Inas, has contributed a lot in giving me the peace of mind to complete this work.

» وَمَآ أُوتِيتُم مِّنَ العِلمِ إِلَّا قَلِيلًا «

الإسراء آية ٨٥

" ... of knowledge it is only a little that is communicated to you. (O men!)"

Holy Quran, Chapter 17, Verse 85.

» وَقُل رَّبِّ زِدنِي عِلمًا «

طه آية ١١٤

"... And say, "O my Lord! advance me in knowledge."

Holy Quran, Chapter 20, Verse 114.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

For real-time applications which are computationally intensive, parallel processing solution is unavoidable, since physical constraints will eventually place a limitation on the performance of a single processor. Furthermore, due to the rapid advances in VLSI design and technology, and consequent decrease in the chip cost and size, parallel processing is becoming more attractive. Parallel processing and pipelining as two approaches utilizing concurrency benefit from the advancement in VLSI technology; however, they lead to more complicated programs. Increased parallelism in architectures means that algorithm designers have to be more concerned with vectorizing, pipelining, or finding systolic implementations for their algorithms. Hence, in order to utilize these sophisticated hardware resources, efficient software tools are required. As a matter of fact, the capabilities of the existing software are much lower than those of the available hardware. In other words, parallel processing systems which are already designed are under-utilized by the available software. This under-utilization may be attributed to the following two reasons. First, researchers have been focusing their attention on designing general purpose architectures by either optimizing for the worst case conditions, or for the most frequent conditions: neither of these methods necessarily leads to optimizing an entire program for a given task. Second, the models used to represent the algorithms lack the feature of exposing their inherent parallelism. Adapting algorithms to parallel implementations is traditionally performed on a case-by-case basis [1]-[5]. Automated techniques for finding parallelism in algorithms have not made significant progress. Despite the fact that parallel processing affords an opportunity to improve computational performance, it requires more implementation effort.

The primary goal of parallel machines such as multiprocessors, vector machines, array processors, etc, is to exploit parallelism of algorithms. Along with introducing these machines, compilers have been developed to extract parallelism from programs written in conventional languages. Moreover, some languages such as Parallel C and Concurrent Pascal have been developed to deal with parallel systems. Researchers have realized the need for a radical departure from the conventional programming methodologies in order to utilize parallel resources efficiently. The most successful compilers either address only the modes of parallelism in pipelined and superscalar machines or restrict themselves to domain-specific areas such as signal processing.

A conventional way of representing concurrent tasks of an algorithm is accomplished by breaking down the algorithm into subtasks, and this technique is well known since the invention of the computer. This is achieved by breaking a program into subroutines; but, this technique is not very suitable for parallel processing implementations in view of the fact that such programs are written with sequential execution in mind. However, if the breakdown of the tasks of a program is done as a consequence of the programming model, we could expect more efficient utilization of concurrent resources.

1.1 Programming Models

A programming model is a set of program abstractions which provides the user a transparent view of the computer hardware/software system. Parallel programming models are those models which deal with programming of multiprocessors or multicomputers [6].

1.1.1 Parallel programming models

In general, parallel programming models can be categorized into the following five groups [6]:

1. *Shared-variable model:* This model is mainly used in tightly coupled multiprocessors. It involves handling the issues of cache coherency, memory consistency, atomicity of memory operations, memory access protection, etc.

2. *Message passing model:* This model is used in a multicomputer system with interconnection network where messages between processing nodes are rooted. In this programming model, the program is distributed or duplicated over the processing nodes.

3. *Data-parallel model:* This model is suitable for single instruction multiple data (SIMD) computers, where one program is executed on several data sets. It could also be used in synchronized multiple instruction multiple data (MIMD) computers, where all processors execute the same program on different data.

4. *Object oriented model:* In this model, concurrency is exploited between objects which use message passing as inter-object communication. Objects, which are dynamically created and manipulated, are program modules that contain the working data along with the executable code.

5. *Logic and functional programming models:* Logic programming are models suitable for knowledge processing and artificial intelligence.

The programming model of our interest is the functional programming model. Functional programming is a model that describes a program by a set of functions, each having a set of inputs and outputs. Executing a function may not produce "side effects"; therefore, there is no notion of storage, assignment and branching. "Side effect" refers to the case where a program module writes to a memory location which may be used by other modules. In other words, a model is free of side effects if the modules of any program can only communicate through *a priori* specified communication paths. By having no side effects, functional programming models give a better chance for exploiting parallelism. Furthermore, parallelism is enhanced by the fact that regardless of the order of producing its operands, a function produces the same results. Data flow and single assignment languages are examples of functional languages.

1.1.2 The data flow model

The basic premise behind data flow is that a node representing a task can be executed whenever its inputs are available. Data flow was originally thought of as a new architectural paradigm rather than a programming model for multiprocessors [7, 8]. Such architectures contain specialized hardware that execute their functions upon the arrival of operands on their inputs. In data flow architectures, data driven semantics is obviously implemented at a low level, namely the machine level. In contrast, data flow as a programming model permits to describe algorithms in such a way that implementations on a general purpose multiprocessor or a pipelined architecture becomes relatively simple.

The principal strength of data flow graphs is that they do not over-specify an algorithm by imposing unnecessary sequencing constraints between operators. Instead, they specify a partial order, where sequencing constraints are imposed only by data precedences. Since the representation does not over-constrain the order of operations, a scheduler has the freedom it needs to adequately exploit deep pipelines, to maximize the re-use of limited hardware resources, or to exploit parallel processing units.

A data flow language as a functional language is a model describing a set of functions performing some tasks with inter-relations represented by the flow of operands. This model of programming is suitable for representation by graphs. A graph which describes the data flow model is called a data flow graph (DFG). A DFG is a directed graph represented by the pair (V,E), where V is a set of nodes representing operations and E a set of edges representing communication paths.

Representing the data flow model by a graph stems from the following two reasons:

1. The concept of data flow is that the availability of operands enables the execution of a node; therefore, references to operands can be thought of as edges, which when loaded, fire the corresponding node.

2. Data flow graphs can be easily composed into larger ones. This can be accomplished by merging the output edges of one graph with the input edges of another according to the governing program.

The data items which are assumed to flow among the nodes through the specified directed edges are called tokens. An edge may be thought of as a FIFO (first in first out) queue which stores incoming operands and feeds them in sequence to the corresponding node every time it fires. A scalar node is one that has no input edges and provides the same output value every time it is requested by another node. In this thesis, we will assume that the data flow model has no scalar nodes, instead we consider the scalar operands as a parameter embedded in the operation of a node itself. An input node is a queue that stores an input stream, while an output node is one which stores an output stream. The input and output nodes do not have the notion of firing, but they store the operands on arrival.

The data flow model has been used for many years [9]-[30] and has proved to be successful in exhibiting the parallelism of the algorithms. It does not impose unnecessary constraints other than those imposed by the data precedency among operations of an algorithm; consequently, the maximum concurrency is theoretically achievable. Moreover, the data flow model provides the user an opportunity to break down a program into tasks so that utilizing the resources becomes more feasible. Another appealing characteristic of the DFG model is that it permits several permissible firing sequences, that is, all firing orders of the actors of a given graph produces the same result as long as the data precedency is preserved.

General data flow graphs

General DFGs are those which can represent any application on a general purpose computer system. Before giving details, we define the following terms. A synchronous actor is one, which upon firing, consumes a fixed amount (number) of data items from each of its input lines, and produces a fixed amount of data items on each of its output lines. A typical synchronous actor is depicted in Figure 1.1 where m and n are, respectively, the number of consumed and produced tokens upon the firing of the actor **a**.

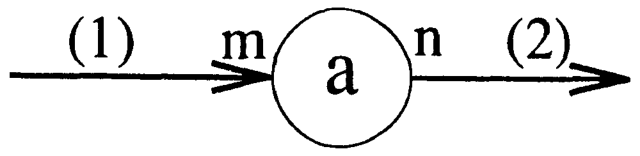


Figure 1.1: A synchronous actor with one input edge and one output edge.

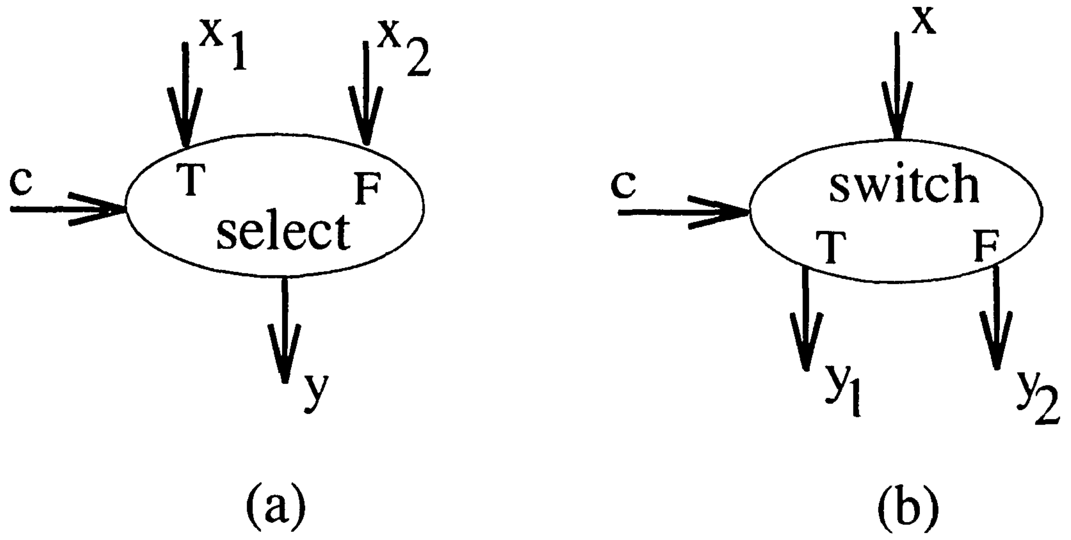


Figure 1.2: Asynchronous actors (a) select and (b) switch.

In contrast, an asynchronous actor consumes and/or produces a variable number of data items on at least one of its input or output lines based on some data-dependent control. Asynchronous actors are also called data-dependent decision making operators, because the behavior of the actor is based on a decision affected by some input data. Several models of actors have been adopted in the literature to represent data dependent flow. For the purpose of demonstration we will use the model adopted in [31]. This model consists of only two asynchronous actors namely *switch* and *select*. The *select* actor shown in Figure 1.2(a) consumes one input from either x_1 or x_2, based on the consumed input from line C, and outputs it on line y. In other words, (a) it consumes C and (b) if C is *true*, then it consumes x_1 and sets $y = x_1$; otherwise it consumes x_2 and sets $y = x_2$. The second asynchronous actor, *switch*, shown in Figure 1.2(b) consumes x and outputs it to either y_1 or y_2 depending on the boolean value consumed at C, that is, $y_1 = x$ if C is *true*; otherwise $y_2 = x$.

To illustrate how to use these actors to represent data-dependent flow, we give the following two examples written in C-like code.

Example 1. (Conditional)

```
Input x;
yl=A(x,c)
If (c)
        y2=G(y1);
else
        y2=H(y1);
y=y2;
```

Example 2. (Loop)

```
Input x;
I=1;
while (I ≤ N)
{
        F(x);
        I++;
}
G(x);
```

The conditional example is depicted in Figure 1.3 using data flow representation. The boolean output C of actor A controls the flow of y_1 which is consumed by either actor H or actor G. S1, and S2 are, respectively, the control lines of the *switch* and the *select* actors. Both S1 and S2 will get the same boolean value as C; therefore, the select actor will select the output of the function (G or H) corresponding to the input y_1 of the same instance.

The data flow representation of the loop example is shown in Figure 1.4. The small diamond node is an initial condition on the edge: it is marked with T to denote a boolean input initialized by a token of value *true*. The large diamond

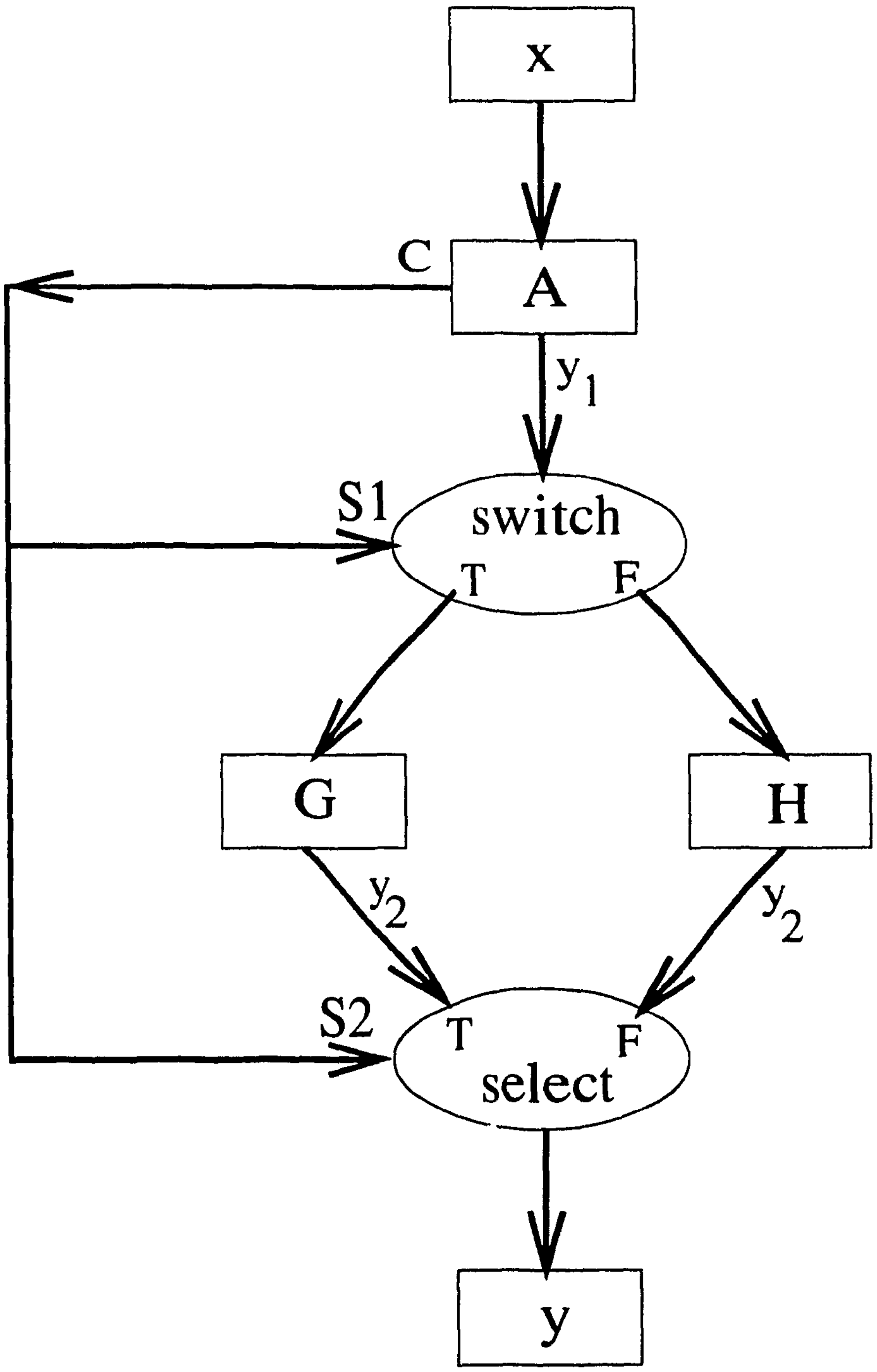


Figure 1.3: An asynchronous DFG showing a conditional example.

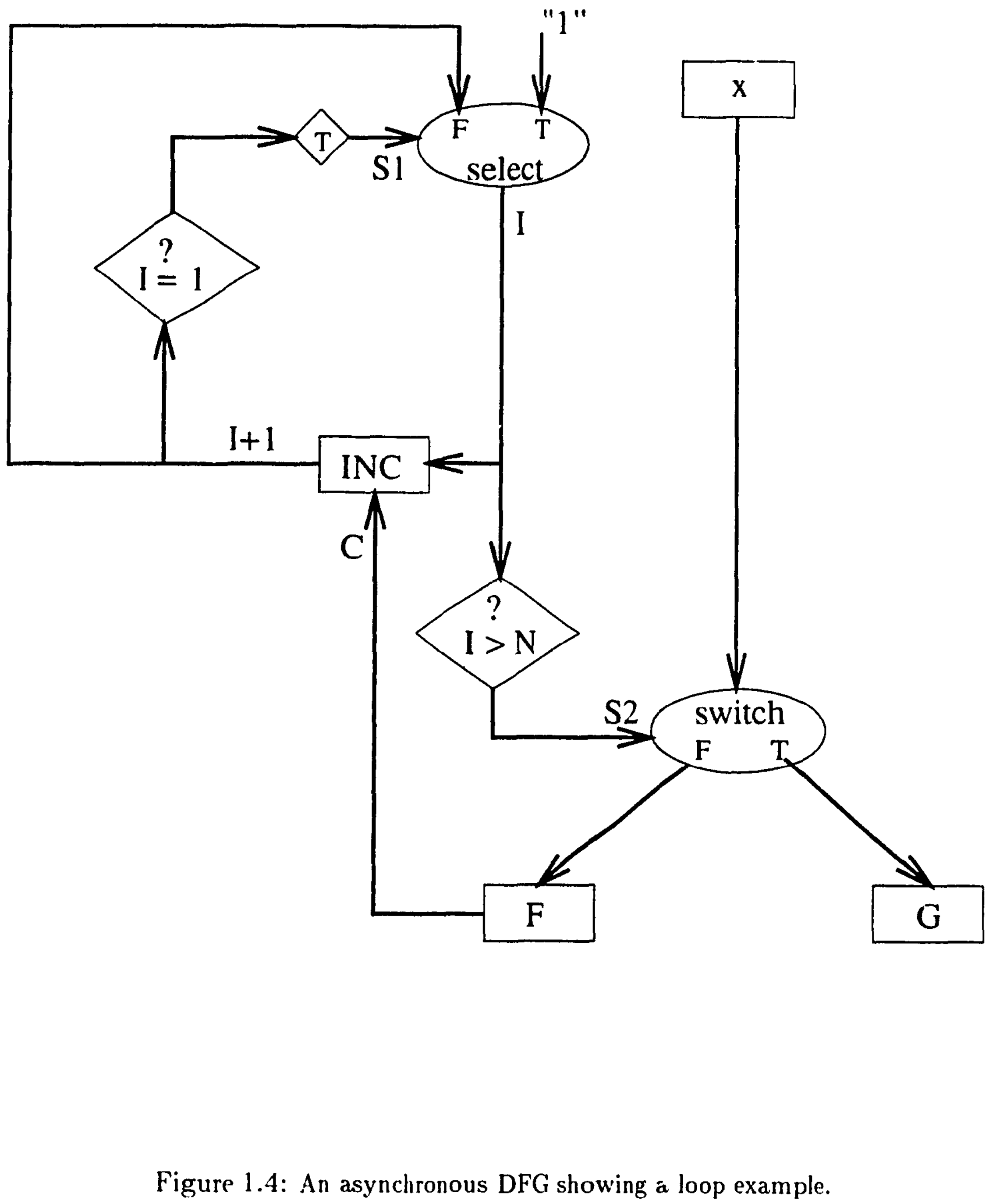


Figure 1.4: An asynchronous DFG showing a loop example.

actors are comparison actors which output boolean tokens depending on the input value relative to the indicated condition. Initially the *select* actor will select the value 1 for **I** because the consumed control on S1 has the value *true*. As long as **I** is less than or equal to N, S2 will get a *false* value, and hence function F will consume the input x and execute. Executing function F, a dummy output is transferred as a control to the **INC** actor which then fires. The $(I = 1)$ comparator will always generate a *false* value, that is, after the select actor consumes the value 1 for I, it will keep selecting the updated value of I. Once S2 gets a *true* token as I exceeds N, the switch will direct the input x to function G which then executes. Since F does not execute, no dummy control will reach **INC** and hence the loop action will halt.

General data flow graphs having data-dependent decision-making operations may not be scheduled at compile time, since the behavior of the actors is dependent upon some input data which may not be known at compile time. In this thesis, we are concerned with the problem of compile time scheduling, and hence, this type of graph will not be considered.

Synchronous data flow graphs

A graph whose nodes are all synchronous is called a synchronous data flow (SDF) graph. In other words, the amount of consumed tokens from each input and produced token on each output are known at compile time for a node in a SDF graph. A SDF graph is said to be consistent if in the long run the number of tokens consumed from an edge matches that produced on the same edge. An inconsistent SDF graph may require infinite resources to execute. Consider the inconsistent data flow graph shown in Figure 1.5. Upon the firing of actor a, one token will be produced

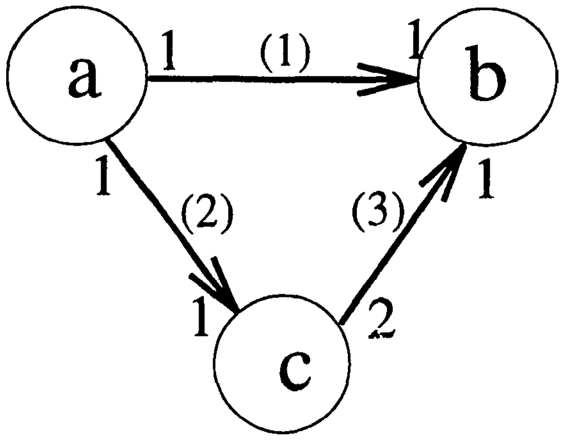


Figure 1.5: An inconsistent SDF graph.

on each of the edges (1) and (2). Hence, actor c will execute because it requires only one operand and as a result it will produce two tokens on edge (3). One of them along with the token on edge (1) will be consumed as actor b fi nd one token will remain on edge(3). Consequently, for every sequence of firing actors a, b and c, one token will stack on edge (3) which means that the graph can not execute with finite memory on edge (3). This kind of a graph is inconsistent.

For large graphs, a systematic way of determining the consistency is required. Lee et al. have proved in [32] that the problem of checking the consistency of a SDF graph can be reduced to a problem of finding the rank of a matrix, called the topology matrix, denoted by ζ. In this matrix the rows represent edges while the columns represent nodes. The matrix entry $\zeta(i,j)$ represents the number of tokens produced on edge i when node j fires. The entry $\zeta(i,j)$ is assigned a negative value if the tokens are consumed by node j. It has been proved in [32] that if the rank of the matrix ζ is equal to $(N-1)$, where N is the number of nodes, then the SDF graph corresponding to the topology matrix ζ is consistent, otherwise it is inconsistent. As an example, consider the SDF graph shown in Figure 1.6. The topology matrix ζ

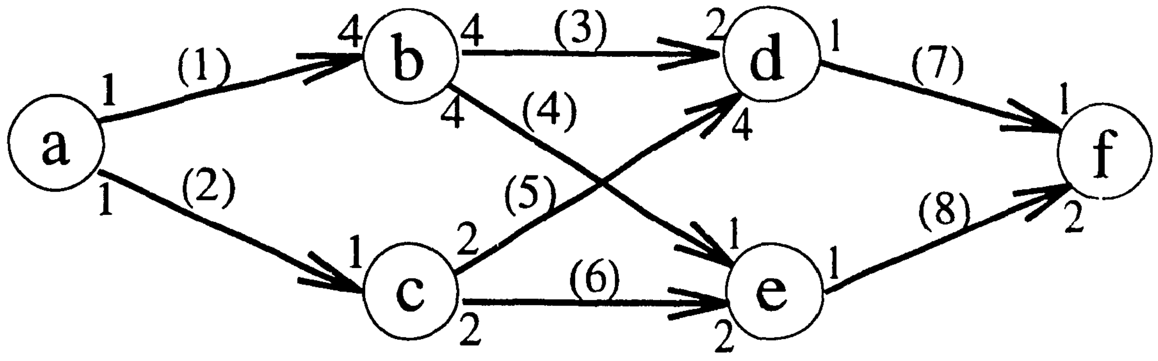


Figure 1.6: A consistent SDF graph.

for this graph is

$$\zeta = \begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{cccccc|l} a & b & c & d & e & f & \\ 1 & -4 & 0 & 0 & 0 & 0 & edge\ 1 \\ 1 & 0 & -1 & 0 & 0 & 0 & edge\ 2 \\ 0 & 4 & 0 & -2 & 0 & 0 & edge\ 3 \\ 0 & 4 & 0 & 0 & -1 & 0 & edge\ 4 \\ 0 & 0 & 2 & -4 & 0 & 0 & edge\ 5 \\ 0 & 0 & 2 & 0 & -2 & 0 & edge\ 6 \\ 0 & 0 & 0 & 1 & 0 & -1 & edge\ 7 \\ 0 & 0 & 0 & 0 & 1 & -2 & edge\ 8 \end{array}$$

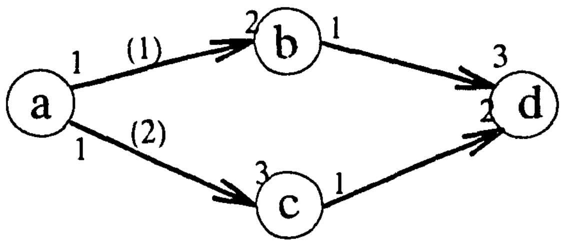
The rank of ζ is five which is equal to $(N-1)$; therefore, the graph corresponding to ζ is consistent.

In addition to using the topology matrix to determine the consistency, it can be used to determine the relative firing frequency of the nodes. Let q be an N-dimensional vector representing the relative firing frequency of the nodes in the SDF graph, then q can be obtained by solving the linear system $\zeta q = 0$. If ζ is of rank N, then a nontrivial q does not exist. For the example of Figure 1.6, the smallest integer solution for q is $q = [4\ 1\ 4\ 2\ 4\ 2]'$ which means that if node a executes 4 times, the other nodes b, c, d, e and f will, respectively, execute once, 4 times, twice, 4 times and twice. The graph will return to its initial state after each iteration specified by the above frequencies

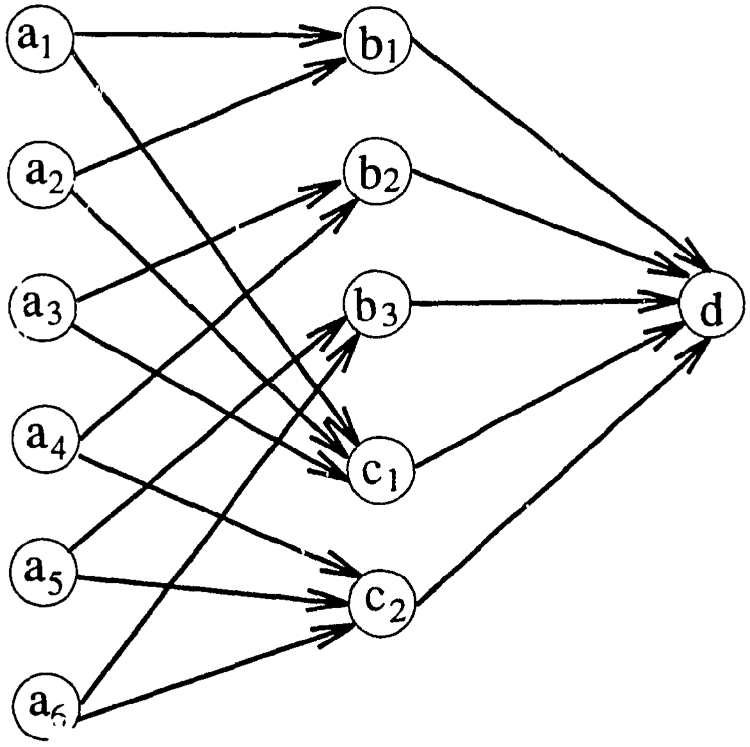
Homogeneous data flow graphs

A special case of SDF graphs is the homogeneous SDF graph in which every node consumes one token from each of its input edges and produces one token on each of its outputs edges. In this thesis, we are interested in this type of graph. A schedule of a homogeneous SDF graph contains one instance for each node, that is, q is a vector of ones. Unlike a general SDF graph, a homogeneous SDF can be scheduled for different optimality criteria. The reason behind the difficulty of optimal scheduling of a general SDF graph is that the instances of a node, with frequency other than unity, may need to be scheduled non-sequentially and on different processors to achieve the optimality criteria.

Fortunately, any consistent SDF graph can be transformed into a homogeneous SDF graph by replacing each node by q_i nodes and establishing the edge set based on the flow of operands among the different instances of each node [33]. We explain this transformation by an example. Consider the SDF graph shown in Figure 1.7(a). If the nodes are ordered as a, b, c and d, then $q = [6\ 3\ 2\ 1]$. Since actor b requires two tokens from edge (1), while actor a produces only one token on edge (1), an instance of b requires the outputs of two instances of a. Using the same reasoning for the rest of the instances of each node, the resultant homogeneous SDF graph will be as shown in Figure 1.7(b). In the rest of this thesis we will only consider homogeneous SDF graphs, and refer to them, for simplicity, as DFGs.



(a)



(b)

Figure 1.7: (a)A non-homogeneous SDF graph, and (b) its corresponding homogeneous SDF graph.

DFGs representing DSP applications

Regardless of the model used to represent the algorithms, designing a general purpose parallel architecture that efficiently exploits their inherent parallelism is still a dream. If the domain of applications is not very wide, then utilizing the concurrency, reducing the hardware cost, and producing efficient software tools will all become relatively simple. In this thesis, we are interested in the class of digital signal processing (DSP) applications (filtering, adaptive filtering, decimation and interpolation, computation of autocorrelation, power spectrum estimation, transformations, etc.) that are computationally intensive, periodic, and involves no data-dependent decision-making. These characteristics are, as a matter of fact, very common in DSP algorithms, and hence it is possible to apply deterministic scheduling techniques at compile time for such applications. A program is said to be periodic if it is applied to an infinite input stream(s) producing an infinite output stream(s). The periodic nature of the applications gives another degree of parallelism: parallelism between iterations (an iteration is defined as the execution of the entire program to consume one input from each input line and produce one output on each output line), where operations between successive iterations can be overlapped. When the program involves no data-dependent decision-making operations, runtime overhead can be avoided, which means that scheduling can be completely accomplished at compile time. As a result of this automation, the hardware cost is substantially reduced and the performance improved.

Representing DSP programs by block diagrams is more natural than representing them by Fortran, although familiarity with Fortran may lead some to oppose this. The data flow graph, which belongs to the class of block representation languages, has proven to be an efficient model [32]-[41] for the representation of DSP

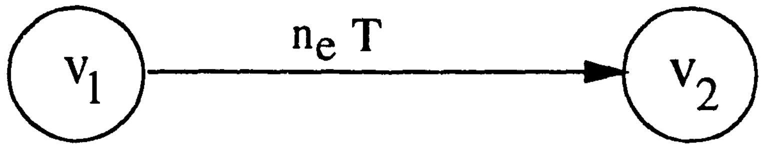


Figure 1.8: A graph showing an edge with an ideal delay of n_e and its two end nodes.

applications described above. In this model, a node has an associated positive integer number that represents the computational delay of its corresponding operation. The complexity of the operations is referred to as the granularity level. Operations might be atomic or compound. Atomic (indivisible) operations represent the basic machine operations like the addition and the multiplication. Compound or non-atomic operations might be macros, code segments, functions, iterations, FFT units, digital filters, etc. If all the operations in a data flow model are atomic then it is a fine grain model; otherwise, it is course grain model.

An edge has an associated nonnegative integer that represents its ideal delay. An edge with no ideal delay appearing in the graph is assumed to be of zero delay. If v_1 and v_2 are, respectively, the initial and terminal nodes of an edge with n_e ideal delays (see Figure 1.8), then the firing of node v_2 at iteration i is dependent upon the availability of the output of v_1 at iteration $(i - n_e)$.

The DFG considered in this thesis is assumed to be a proper graph, that is, there is a path to every node in the graph from an input node, and there is a path from every node in the graph to an output node. Throughout the thesis, without loss of generality, we assume that a DFG has one input node and one output node, since it can be easily transformed to become so even if the given graph has more than one input node or output node. For a graph having more than one input node, an artificial input with zero computational delay is added to the node set of the graph and a zero-ideal-delay edge is added from this node to each input node in the given graph. In a similar manner, a graph having more than one output node is altered. We further assume that the edges in the DFG model have no communication delays as though all the nodes are assigned to a single processor. This assumption may sound too restrictive, but is really not so since the communication delays may be encapsulated with the computational delays of the nodes.

1.2 Graph Theory Background

In this section certain definitions from graph theory that are needed are now given [42].

A directed graph $G = (V, E)$ is uniquely represented by its node set $V(G)$ and its edge set $E(G)$. An edge $e = (v_I, v_T)$ is said to be incident out of its initial node v_I, and incident into its terminal node v_T. The initial and terminal nodes of an edge are said to be its end nodes. An edge is also said to be incident on its end nodes. Every node has an in-degree and an out-degree. The in-degree of a node is the number of directed edges incident into it while its out-degree is the number of directed edges incident out of it. Consider a subgraph $G' = (V', E')$ where $V' \subset V$. Then G' is said to be a node-induced subgraph onto the node set V' if $E' \subset E$ such that edge $(v_i, v_j) \in E' \iff v_i, v_j \in V'$. Let us use the operator $\oplus$ to denote any set operation. The operation $G_1 \oplus G_2$ is assumed to be equivalent to the two operations: $V(G_1) \oplus V(G_2)$, and $E(G_1) \oplus E(G_2)$. Hence, the equation $G_1 \oplus G_2 = G_3$ implies that $V(G_1) \oplus V(G_2) = V(G_3)$, and $E(G_1) \oplus E(G_2) = E(G_3)$.

A path p is a finite alternative sequence of distinct nodes and distinct edges beginning and ending with nodes. The path length, len[p], of the path $p = v_1 \rightarrow v_2$ is defined as the minimum elapsed time between consuming the input operand(s) at its initial node v_1 and producing an output at its terminal node v_2 [43], that is,

$$len[p] = \sum_{\text{node } v_i \in V(p)} d^c_{v_i} - T \sum_{\text{edge } e_i \in E(p)} n_{e_i}, \tag{1.1}$$

where $d^c_{v_i}$ is the computational delay of node v_i, n_{e_i} the number of ideal delays of edge e_i, and T the iteration period. If in an iteration, the firing of the terminal node of a path p does not depend on the output of its initial node, then len[p] may be negative. We define len[p[, len]p] and len]p[as the path lengths of the path p when the computational delays of its terminal node, initial node, and end nodes, respectively, are excluded. An elementary path whose initial and terminal nodes are identical is called a circuit. A DFG which contains at least one circuit is said to be a cyclic graph, otherwise it is acyclic.

1.3 Thesis Objective

The primary objective of this thesis is to seek a compile-time static schedule for completely specified data flow graphs on a parallel system. Since completely specified programs are very common in DSP applications, the scheduling technique is mainly concerned with DSP algorithms. The theory developed in this thesis (see also [44, 45, 46]) is amenable to a general DFG as long as the computational delays of all the nodes are known and there are no data-dependent decision-making operations.

The thesis is concerned with three different optimality criteria: rate-optimality, delay-optimality, and processor-optimality. These optimality criteria will be discussed thoroughly in the next three chapters. In Chapter 2, we first define the rate-optimality criterion, and review related work. Then, a theory for finding a rate-optimal time schedule, without consideration to processor allocation, is presented. In Chapter 3, we present a theory and techniques necessary to modify the schedule developed in Chapter 2 so that delay-optimality is achieved. Chapter 4 deals with the mapping of the time schedule to a processor assignment with an objective to minimize the number of processors needed. In Chapter 5, performance analyses of the different algorithms presented in the preceding chapters are carried out, and a comparison with some related work presented. Chapter 6 concludes the thesis by highlighting the contributions of this study and making some suggestions for possible further work.

Chapter 2

Rate-Optimal Time Scheduling

The scheduling problem can be described by the 4-tuple (PS, TS, PQ, FQ), where PS is the set of processors, TS the set of tasks constituting a given program, PQ the sequence of processors to which the tasks are to be assigned, and FQ is the sequence of firing times of the tasks. Further, let $TQ[P_k]$ be the ordered sequence of tasks which are allocated to processor P_k, that is, if the task s_i precedes s_j in $TQ[P_k]$, then processor P_k will execute the task s_i before executing s_j. For example, let $PS = \{P_1, P_2\}$, $TS = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$, $PQ = \{P_1, P_1, P_2, P_1, P_2, P_2, P_2\}$ and $FQ = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. This implies that the tasks s_1, s_2 and s_4 are allocated to the processor P_1, and the tasks s_3, s_5, s_6 and s_7 to the processor P_2. Further, the task s_1 fires at the time instant t_1, s_2 at t_2, an so on. If $t_1 < t_2 < t_4$, $TQ[P_1] = \{s_1, s_2, s_4\}$; and if $t_3 < t_5 < t_6 < t_7$, then $TQ[P_2] = \{s_3, s_5, s_6, s_7\}$. Hence, scheduling can be defined as the process of assigning tasks to processors, ordering their execution and specifying the times when they start execution. Whether PQ, FQ or TQ are determined at compile time or at run time, scheduling can be classified into the following categories [47, 34].

1. *Fully dynamic:* An operation is assigned to a free processor at run time when all its operands are ready. that is, both PQ and FQ are determined at run time.

2. *Static Assignment:* Operations are assigned to processors at compile time. The availability of operands determines which operations to fire. That is, only PQ is determined at compile time.

3. *Self timed:* The tasks are assigned to the processor at compile time and their order of firing is also known before hand. The availability of operands determine whether to proceed with execution or to pause. That is PQ and TQ are determined at compile time but not FQ.

4. *Fully static:* The allocation of tasks to processors and the exact times of their execution are specified at compile time. That is. both PQ and FQ are determined at compile time.

For a periodic algorithm, the scheduling parameters are determined for one iteration and the complete schedule is constructed by a periodic repetition of a single iteration schedule. It is to be recalled that a periodic algorithm is applied to a stream of inputs to produce a stream of outputs by repeating the tasks of the algorithm on each and every input. In addition to the above scheduling schemes, periodic programs can be scheduled in a cyclo-static manner. Cyclo-static scheduling is similar to the fully static one except that in the former the tasks are periodic in the processor space. That is, a task may be allocated to different processors in different iterations according to some periodic vector that determines the processor displacement from iteration to iteration for each task [43, 48, 49, 50].

In general, for problems where scheduling at compile time is unattainable, many solutions have been proposed to solve the problem of data dependency at run time. Not surprisingly, most systems that execute operations in parallel have data-driven semantic models. As an example, a general purpose super-scalar pipeline processor uses the reservation-station technique to resolve data dependency among instructions. A pre-specified number of instructions are decoded without checking their data dependency and sent to reservation stations. Each instruction in a reservation station waits for its operands to be ready. An instruction having its operands ready waits for the appropriate functional unit to be free. The data dependency among the instructions waiting in the reservation stations is resolved by register renaming and/or scoreboarding techniques. More details about these techniques can be found in [6] and [51]. Obviously, an algorithm which has to be scheduled during run time (fully or partially) requires more hardware and/or software to dynamically resolve the issue of data dependency to ensure its correct execution. However, the overhead due to such dynamic behavior may be avoided for some algorithms if the scheduling can be completely accomplished at compile time. An algorithm can be fully scheduled at compile time if it has the following properties.

1. It has no loop for which the number of repetitions is not known before hand.

2. It has no branch instruction whose behavior (branch/proceed) depends on some value to be determined at run time.

3. It is a pure code, that is, it may not modify itself during run time.

4. Each operator in the algorithm has a fixed number of operands, that is, in an iteration, the number of consumed inputs as well as the number of produced outputs are fixed and known at compile time.

In a fully static schedule, also known as a completely specified static schedule, a task has two parameters which are specified at compile time: the firing time of its execution and the processor which will execute it. In this chapter, we focus on the first parameter, while the second is dealt with in Chapter 4. It is noted that the terms algorithm and DFG are used interchangeably. A schedule that gives the firing time instances of the tasks of a given algorithm without information about processor allocation is referred to as a time schedule. Hence, the principal objective of this chapter is to obtain a valid time schedule for an algorithm represented by a DFG.

The data dependency in algorithms represented by DFGs is inherently exposed by the programming model. Therefore, our task is not to extract the data dependency between actors but instead to schedule them in a way that achieves some optimality criteria without sacrificing the integrity of precedency among tasks.

In this chapter, we will first define the scheduling problem of a DFG on a multiprocessor and explain the implications of the different scheduling parameters. Also, more specifications about the DFGs of interest will be given. In Section 2.2, the rate optimality criterion will be discussed. The theory developed in this work to time schedule DFGs on multiprocessors will be the subject of Section 2.4.

2.1 The Scheduling Problem

A poor schedule for any system in general, and for a parallel one in particular, may violate the timing requirements or lead to an inefficient utilization of the resources. In the scheduling problem, therefore, we seek a schedule that respects the time requirements and at the same time utilizes, optimally or near-optimally, the resources of a given system. The problem of allocating tasks to processors and specifying at compile time their firing times has been proved to be NP hard, that is, a problem that is not solvable by deterministic algorithms in a polynomial time [52, 53]. For such a problem, a heuristic solution is necessary.

The scheduling problem consists of a fully specified DFG representing a periodic algorithm, a set of identical processors and some optimality criteria. In the literature, a node in a fully specified DFG is assumed to have a single atomic operation. In this thesis, we relax the definition of a fully specified DFG by allowing nodes of multiple operations as long as the total computational delay of each such node is known for the underlying architecture. For the solution time to be possible at compile, the flow graph which describes the problem has to be deterministic and fully specified. For example, conditionals or decision nodes (nodes whose outcomes affect the flow of control) leading to non-deterministic problems may not be solved until execution time, thus precluding compile-time solutions. The processors are assumed to be identical to give flexibility for the tasks to run on any processor as long as their precedence requirements are satisfied. A schedule may be preemptive or non-preemptive. It is preemptive if the interruption of the tasks is permitted before being completed; otherwise it is non-preemptive. In general, a preemptive schedule is more efficient than a non-preemptive one unless the task switching cost (in terms of switching time and memory space) is relatively high [52]. In cases where task switching is costly, preempting is usually avoided.

The scheduling problem can be considered to consist of two phases, time scheduling and processor assignment. Time scheduling is concerned with specifying the firing time of each node in a single iteration assuming an infinite number of processors, while processor assignment is a mapping procedure of the nodes in the time schedule to a processor-time space. In this chapter, we present a new theory along with an algorithm for finding a rate-optimal time schedule for a completely specified DFG.

2.2 Rate Optimality

In most real-time applications, the throughput is very critical to be maximized. It is to be recalled that the throughput, a performance measure, refers to the number of outputs per time unit. A schedule for a given algorithm is said to be *rate-optimal* if and only if it achieves the maximum throughput, equivalently, minimizes the time between producing successive outputs. The time between successive outputs is referred to as the iteration period. Hence, a rate-optimal schedule should achieve the minimum iteration period, which is usually referred to as the iteration bound. For an acyclic DFG, the iteration bound is determined by the computational delay of the longest operation. This is so because, if enough computational resources are available, a pipeline with a number of stages equal to the number of operations can definitely achieve an iteration period equal to the computational delay of the longest operation. The minimum iteration period, however, could be less than the computational delay of the longest operation if more than one processor can share the execution of that operation.

For a cyclic DFG, the iteration bound is not only constrained by the hardware resources, but also by the topology of the graph. If the hardware resources are unlimited, the iteration period bound as constrained by the topology of the given graph is given by

$$T_0 = \max_{C \in circuits} \left\lceil \frac{D_C}{N_C} \right\rceil , \tag{2.1}$$

where

$$D_C = \sum_{v_j \in V(C)} d^c_{v_j},$$

and N_C is the total number of delays in the circuit C [54], which is given by

$$N_C = \sum_{e_i \in E(C)} n_{e_i}.$$

Although this bound has been given in the literature as an axiom, we like to provide the reader with some reasoning for this formula. Let A be a node in a circuit C. Obviously, the output token of node A at iteration i (A_i) will propagate through the nodes of C (changes as per the functions of these nodes) and come back to A as an input needed for the computation of the $(i + N_C)th$ iteration. The minimum time required for node A to provide its output at the $(i + N_C)th$ iteration since outputting A_i is D_C. Therefore, at most N_C outputs can be produced by node A in D_C time units, and hence the above formula.

When there are some unbreakable operations with computational delays greater than the iteration period, the rate-optimal schedule is not achievable using fully-static scheduling. This is due to the fact that in fully static scheduling, an operation is assigned to a single processor and hence, has to be executed within the time limit determined by the iteration period. In this case, rate-optimality can be achieved using cyclo-static scheduling or scheduling via unfolding. Consequently, if the schedule is required to be fully static, the iteration bound has to be modified as

$$T_0' = \max\left(T_0, \max_{v_i \in V(G)} d^c_{v_i}\right). \tag{2.2}$$

A circuit C is called *critical* if its loop bound (D_C/N_C) is equal to the iteration period bound. A non-critical circuit has a spare time called the *slack time*. The slack time of a circuit can be thought of as the total time during which none of the circuit operations is executed within one iteration period T_0, and is given by

$$ST(C) = T_0 N_C - D_C. \tag{2.3}$$

It is clear from (2.3) and (1.1) that

$$ST(C) = -len[C]. \tag{2.4}$$

If T_0 is a non-integer, then unfolding is necessary to achieve the bound. A DFG is said to be k-unfolded if it has been unfolded k times, and k is called the unfolding factor. The k-unfolded DFG has an iteration period equal to (kT_0), where T_0 is the iteration period of the original DFG [55]. The unfolding factor can be expressed as

$$k = \frac{N_C}{GCD(N_C, D_C)},$$

where GCD denotes the greatest common divisor, and N_C and D_C are, respectively, the ideal and the computational delays of any critical circuit [56]. For the fully-static schedule to exist, the iteration bound is given by (2.2), where T_0 is replaced by kT_0.

2.3 Previous Work

In the literature different techniques have been developed to employ parallelism in the implementation of algorithms represented by DFGs [54].

1. *Parallelism in a single iteration (intra-iteration parallelism):* In these methods [57], a given cyclic DFG is first converted to an acyclic one which corresponds to the precedence graph of a single iteration. This conversion can be carried out by replacing each delay element by an input and an output node. Scheduling acyclic data flow graphs has been well studied in the literature [58, 59]. In general, techniques that first convert the cyclic graph to an

acyclic one prior to scheduling generate non-overlapped schedules, where the schedule for one iteration is periodically repeated. Since these methods do not utilize inter-iteration parallelism, they are, in general, incapable of producing schedulers with maximum throughput or optimal number of processors. Hence, these methods are not suitable for iterative algorithms in real time applications in which the throughput is a critical issue.

The parallelism in the resultant acyclic graph can then be exploited to minimize the schedule length by any of the following scheduling methods [54]:

(a) Precedence graph method [60].

(b) Precedence matrix method [61].

(c) Critical path method [59, 62].

(d) The method of Nouta and Simula based on the petri nets [63, 64].

The repetitive nature of the DSP algorithms is totally ignored (not utilized), if this technique is used for their scheduling.

2. *Parallelism within a block of iterations* [65, 66]: Before converting the given graph to an acyclic one, it is unfolded by some factor n. This kind of transformation helps capture more parallelism as concurrency can be utilized among the operations of n iterations. Although these methods do better job than those from the preceding category, they do not, in general, produce rate-optimal schedules for high sampling rate in real-time applications.

   This technique is known for implementations of DSP algorithms. Fore example, high speed filtering can be accomplished by processing sequences of input samples instead of one sample. In [67], high speed implementation for finite convolution is done using Fourier transform. This method is efficient only if the transform is applied to filters with high orders. Hence blocking is necessary for filters with low order.

Blocking will definitely improve the utilization of inter-iteration parallelism. However, the maximum utilization may not be achieved as the blocking factor is finite. Hence a virtual infinite unfolding of the graph is required for the maximum utilization of this kind of parallelism.

3. *Pipelining the feed-forward part of the DFG:* Pipelining is efficient when the program consists of repeated processes. The basic principle behind pipelining is that the computation for the iteration of a given task can be started before the computations of earlier iterations have been completed. Pipelining can be viewed as additional delays inserted into the DFG; as a consequence, the I/O delay increases. These delays can be added to the feed-forward part of the network, but obviously may not be added to the circuits of the graph since they may alter the data dependency between iterations. However, pipelining the tasks of a circuit in a graph is possible for certain values of iteration period. This is because a task is not only pipelined with tasks from the same iteration but also with some tasks from other iterations.

4. *Periodic parallelism (inter-iteration parallelism):* This parallelism is a result of utilizing the parallelism between successive iterations in an infinite or a sufficiently long schedule [68, 43, 55, 69]. Schedulers that utilize this parallelism, which can be either fully-static or cyclo-static, generate overlapped schedules, where operations from different iterations are overlapped. When maximum parallelism is utilized, the resulting schedule will be rate optimal for which the

iteration period T is equal to the iteration bound T_0. Some of the important rate-optimal scheduling techniques existing in the literature will be discussed later in this section.

As a matter of fact, the maximum inter-iteration parallelism can be accomplished by using the technique of direct blocking. It has been shown in [55] that the blocking-factor can be chosen so that the unfolded graph is a perfect-rate one, which can always be scheduled rate-optimally. Each circuit in a perfect-rate graph has only one ideal delay unit, say associated with edge e. Hence, the nodes of a circuit can be sequentially scheduled by starting from the terminal node until reaching the initial node of the edge e. Scheduling the acyclic version of the resulting perfect-rate graph will hence produce a rate-optimal schedule.

In the following paragraphs, we will discuss some schedulers that produce rate-optimal schedules. A simple technique to produce a rate- and delay-optimal schedule for an acyclic graph is the critical path method. In this method, the longest distance is computed between the input node of the graph and every other node. If the input node is scheduled at time 0, the distance between the input node and any other node represents its scheduling time. For the longest distance calculations, the weight assigned to a node is its computational delay, and the weight assigned to an edge is the negative of the product of its ideal delay and the iteration period. The basic problem of this technique is the fact that it does not utilize the inherent flexibility of the nodes to minimize the number of required processors.

The maximum spanning tree (MST) technique proposed in [54] and [70] is concerned with producing rate-optimal schedules for IIR filter networks. In this technique, the graph is transformed into an equivalent one by removing the ideal delays from the branches of the MST and inserting non-negative shimming delays such that the resulting delay associated with each link is equal to the total delay of the corresponding fundamental loop in the original DFG. In the resulting DFG, the integrity of time precedency is preserved among the nodes. Then, the MST is scheduled according to the precedency constraints represented by the edges, and the timing constraints represented by the computational and shimming delays. This algorithm produces a valid rate-optimal time schedule. However, other optimality criteria are not addressed as well as no processor optimization is taken into consideration, leading, in general, to solutions with non-optimal number of processors.

In [55], the DFG is first converted to a perfect-rate graph by unfolding it with an unfolding factor that is equal to the least common multiple of the ideal delays in all the circuits in the graph. Each circuit in the resulting graph will contain one unit of ideal delay. Since the computational delay of any circuit will be less than or equal to the iteration period, as will be shown later, the DFG can always be scheduled rate-optimally. It can be easily seen that this approach has a serious problem of increasing the size of a given program. As an example, a DFG for a 7th-order IIR filter may possibly need to be unfolded by a factor of 420 in order for it to be converted to a perfect-rate graph. Further, this method does not try to optimize the number of processors.

The cyclo-static method [43, 48, 49, 50] consists of a depth-first search of cyclo-static solutions by fixing both the iteration period and the number of processors. Unlike static solutions, cyclo-static solutions have the property that the tasks assigned to a processor vary from iteration to iteration. However, the allocation of the tasks on the processors is periodic. That is, the tasks allocated to a processor at iteration n will be the same as those allocated to it at iteration $n + n_p$, where n_p is the cyclo-scheduling period. The main strength of this method is that no unfolding is required even when there are some operations with computational delays greater than the iteration period. This is due to the fact that an operation may be assigned to more than one processor in consecutive iterations. The basic problem with this method is that it does not guarantee a schedule, since it is not always possible to obtain a solution when both the iteration period and the number of processors are fixed. An example of the cyclo-static solution is the scheduling heuristic proposed in [43]. This heuristic narrows the search for a solution by imposing all constraints on the required solution: maximum throughput, minimum delay, and minimum number of processors. The algorithm is expected to output a schedule matrix in the time-processor space, where columns represent time slots and rows represent processors. The resulting matrix not only specifies which tasks are allocated to each processor, but also gives the information as to which iteration the occurrence of each node belongs. Along with the scheduling matrix, the algorithm generates a cycling vector that determines the relative permutation of the processors in successive iterations; hence, the cyclo-static schedule is completely specified. In this heuristic, the flexibility of all the nodes are first computed by considering the slack times of the circuits and the delays of the I/O paths, and next the nodes are scheduled in the order of increasing flexibility.

The most efficient static scheduling technique developed thus far is the range-chart technique [68, 71]. In this technique, first, a node is chosen as the reference node, and the flexibility of every other node is calculated. The flexibility of the nodes in a DFG is represented in a range chart form, which indicates the range within which each node can be executed. The node with the minimum range or flexibility is first chosen to be scheduled, and the range chart updated. The process of choosing a node and updating the chart is repeated until all the nodes are scheduled.

It may be noted that the heuristics proposed in the literature handle scheduling as a single problem. In contrast, we divide the scheduling problem into two phases. The first phase, time scheduling, is solved analytically using combinatorial concepts. The second phase is the mapping of the time schedule to a processor assignment.

2.4 Time Scheduling Theory

The scheduling problem addressed in this thesis is concerned with finding a rate-optimal schedule for a cyclic DFG. In a cyclic DFG, the critical circuits determine the iteration period bound. If C is a critical circuit, it is clear from (2.4) that len[C]=0, since ST(C)=0 from (2.3) and (2.1).

We will first show that any cyclic, fully specified DFG can be converted through a sequence of transformations to an acyclic one. Then we will use one of the known algorithms to schedule the resulting acyclic graph as a starting step in scheduling the whole graph. The transformation is accomplished by contracting each circuit in the graph to a single node.

Lemma 1 *The time schedule of the nodes of a critical circuit is uniquely specified if and only if one of its nodes is time-scheduled.*

Proof:

Necessity: Obvious.

Sufficiency: Assume that $v_0 \in V(C)$ is a node in the critical circuit C with a fixed time schedule. For any node $v_c \in V(C)$, assume that its earliest firing time is EFT(v_c) and its latest firing time LFT(v_c). Without loss of generality, assume that

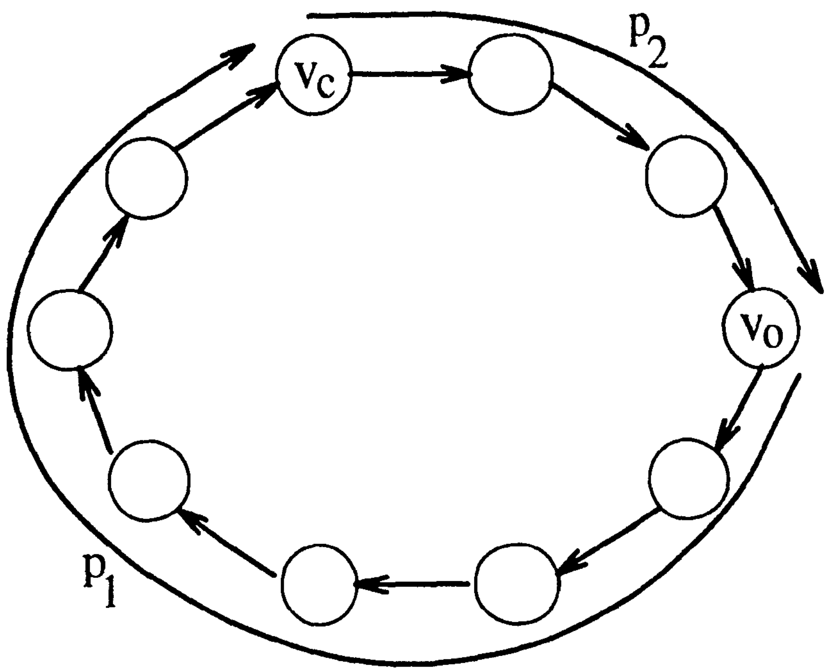


Figure 2.1: A critical circuit.

v_0 is fired at time zero.

Assume p_1 is the directed path $(v_0 \rightarrow v_c) \subset C$ and p_2 the directed path $(v_c \rightarrow v_0) \subset C$ as shown in Figure 2.1. Since C is critical,

$$len[p_1[+ len[p_2[= 0. \qquad (2.5)$$

The precedence constraints require that

$$FT(v_0) + len[p_1[\leq EFT(v_c) \qquad (2.6)$$

and

$$LFT(v_c) + len[p_2[\leq FT(v_0) \qquad (2.7)$$

where $FT(v_0) = 0$ is the firing time of v_0. Adding (2.6) to (2.7) gives

$$LFT(v_c) + len[p_1[+ len[p_2[\leq EFT(v_c)$$

Using (2.5), this inequality reduces to

$$LFT(v_c) \leq EFT(v_c)$$

However, $LFT(v_c) < EFT(v_c)$ is not feasible. Therefore,

$$LFT(v_c) = EFT(v_c),$$

and hence the proof. ■

Lemma 2 *Let $v_x \notin V(C)$ and $v_c \in V(C)$ be connected nodes. If the scheduling time of v_x is fixed relative to the scheduling time of v_c, then it is also fixed relative to all other nodes in C.*

Proof:

Since the scheduling of v_x is fixed relative to the scheduling of v_c, we can establish that

$$FT[v_x] - FT[v_c] = B_1, \tag{2.8}$$

where B_1 is a constant. Further, as per Lemma 1, the scheduling time of v_c is fixed relative to all other nodes in C; hence

$$FT[v_c] - FT[v_{cx}] = B_2, \tag{2.9}$$

where B_2 is also a constant and v_{cx} is any node in C. Summing up (2.8) and (2.9), we get

$$FT[v_x] - F[v_{cx}] = B_1 + B_2 = Constant,$$

and hence the proof. ■

Theorem 1 *In a DFG, any critical circuit can be contracted to one of its nodes without altering the scheduling problem.*

Proof:

Assume that the critical circuit C is contracted into one of its nodes v_0. According to Lemma 1, the rest of the nodes in C have fixed scheduling times as long as v_0 does; therefore, if the node v_0 is scheduled correctly, the rest of the nodes in C will automatically get scheduled according to their displacement relative to v_0. Hence removing such nodes will not affect the scheduling problem as far the scheduling of the circuit nodes is concerned.
Let v_x be any other node not in C. If v_x is not connected to C, then it can be scheduled independently of v_0. If it is connected to some node in C, then according to Lemma 2, the scheduling time of v_x relative to this node can be represented by its scheduling time relative to v_0. Therefore, the circuit nodes other than v_0 are not necessary to be present in the DFG as far as the scheduling of the rest of the nodes in the DFG is concerned, and hence the theorem. ■

So far, we have shown that a critical circuit can be replaced by a single node and yet the scheduling problem is not altered. This kind of a replacement will be referred to as critical circuit contraction, or simply circuit contraction. In order to contract a circuit to a single node, it is required to make up for the data dependency between the circuit nodes which will be removed due to the circuit contraction and the other graph nodes. According to Theorem 1 this is always possible. Let the circuit C be a critical circuit to be contracted to a node v_0. The data dependency between a node v_x not in C and any node v_c in C due to the existence of some path between v_x and v_c, can be compensated for by introducing a path to (form) v_x from (to) v_0 having the same length. One way of introducing such path is by introducing pseudo nodes whose assigned delays compensate for the missing subpaths due to the circuit contraction. Using the above method of compensating for the missing subpaths by introducing pseudo nodes, we now give an algorithm for the contraction of a critical circuit C in a given graph G to form a new graph G'. The single node to which the circuit C is contracted will be referred to as a "supernode".

Contraction Algorithm:

1. Let G' be the node-induced subgraph on its node set $V(G') = [V(G) - V(C)] \cup \{v_0\}$, where $v_0 \in V(C)$ is the node to which the circuit C is to be contracted.

2. For each node $v_c \in [V(C) - \{v_0\}]$ with an out-degree greater than one, perform the following operations.

   (a) Create a pseudo node v_c' with a computational delay equal to len$]p]$, where p is the directed path in C from node v_0 to node v_c, i.e, $p = (v_0 \rightarrow v_c) \subset C$.

   (b) Set $V(G') = V(G') \cup \{v_c'\}$.

   (c) Move the tail of each edge $e \in E(G)$, incident out of the node v_c to some node not in C, to be incident out of the node v_c'. (The terminal node of the edge and the ideal delay associated with it remain unchanged.) Set $E(G') = E(G') \cup \{e\}$.

   (d) Set $E(G') = E(G') \cup \{(v_0, v_c')\}$, where (v_0, v_c') is a delay-free edge.

3. For each node $v_c \in [V(C) - \{v_0\}]$ with an in-degree greater than one, perform the following operations.

   (a) Create a pseudo node v_c'' with a computational delay equal to len$[p[$, where p is the directed path in C from node v_c to node v_0, i.e, $p = (v_c \rightarrow v_0) \subset C$.

   (b) Set $V(G') = V(G') \cup \{v_c''\}$.

(c) Move the head of each edge $e \in E(G)$, incident into v_c from some node not in C, to be incident into $\{v_c''\}$. (The initial node of the edge and the ideal delay associated with it remain unchanged.) Set $E(G') = E(G') \cup \{e\}$.

(d) Set $E(G') = E(G') \cup \{(v_c'', v_0)\}$, where (v_c'', v_0) is a delay-free edge.

We will now study the effect of applying the Contraction Algorithm to contract a critical circuit on the slack times of other circuits and the lengths of the I/O paths. In fact, we are going to show that neither the slack time of a circuit, nor the length of an I/O path is effected by contracting a critical circuit to a single node using the Contraction Algorithm. Recall that, as per (2.4), the slack time of a circuit is nothing but the negative of its path length. Hence, to show that neither the slack time of a circuit nor the length of an I/O path is altered by the contraction of a critical circuit, it is enough to show that the length of any path in the graph does not change due to such contraction. It is to be noted that a path may change its node and edge sets due to a circuit contraction. Therefore, when we say that the length of the path does not change, we relate the original path with the corresponding path in the graph after contraction. Let G' be the graph that results after contracting a critical circuit C from G. For every path $p \in G$ whose end nodes are not in C, there is a corresponding path $p' \in G'$ such that $len[p] = len[p']$. As a matter of fact the path p' is not necessarily elementary, that is, it may contain a circuit as will be shown soon.

Definition 1 *A path p_h^C is said to be a hop-path with respect to a circuit C if and only if it has no common edges with C, and its node set is disjoint from the node set of C except for its initial and terminal nodes, that is, $E(p_h^C) \cap E(C) = \Phi$, and $V(p_h^C) \cap V(C) = \{v_I, v_T\}$, where v_I and v_T are the initial and terminal nodes of p_h^C.*

Lemma 3 *Let G' be a graph resulting after contracting circuit C from a given graph G, and let p_h^C be a hop-path with respect to C. G' will contain a circuit of the form $v_0, v_I', (p_h^C - \{v_I, v_T\}), v_T'', v_0$, where v_I' and v_T'' are the pseudo nodes generated by the Contraction Algorithm as the nodes v_I and v_T are removed, and v_0 is the supernode to which the circuit C has been contracted. Further, this circuit is the path in G' corresponding to p_h^C.*

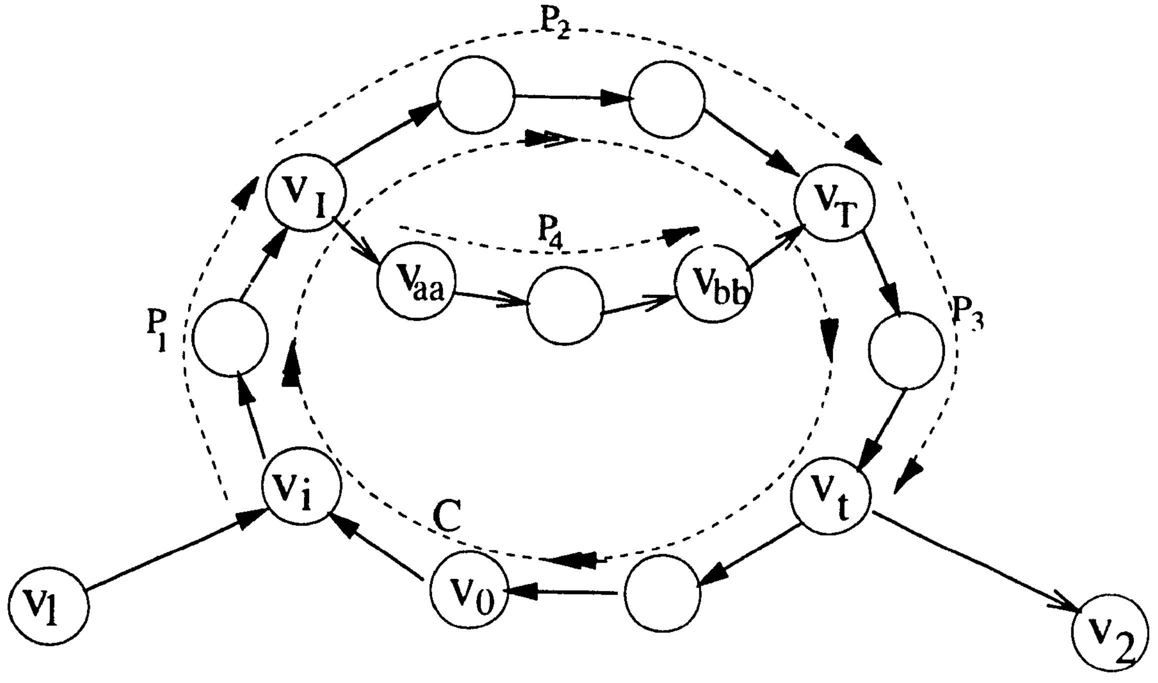
Proof:

Let p_h^C be a hop-path with respect to a circuit C. As seen from Figure 2.2(a), p_h^C can be expressed as $p_h^C = v_I, p_4, v_T$. Contracting the circuit C (see Figure 2.2(b)) will result in the pseudo node v_I' with a delay-free edge coming from v_0 and an edge going to the initial node of p_4. It will also result in a pseudo node v_T'' and an edge incoming from the terminal node of p_4 and an edge going to the node v_0. p_4 along with these four edges constitute the circuit $v_0, v_I', (p_h^C - \{v_I, v_T\}), v_T'', v_0$, and hence the first part of the lemma. The second part of the lemma is obvious as the node v_I is replaced by the pseudo node v_I' and an incoming edge from v_0, and the node v_T by the pseudo node v_T'' and an outgoing edge to v_0. ■

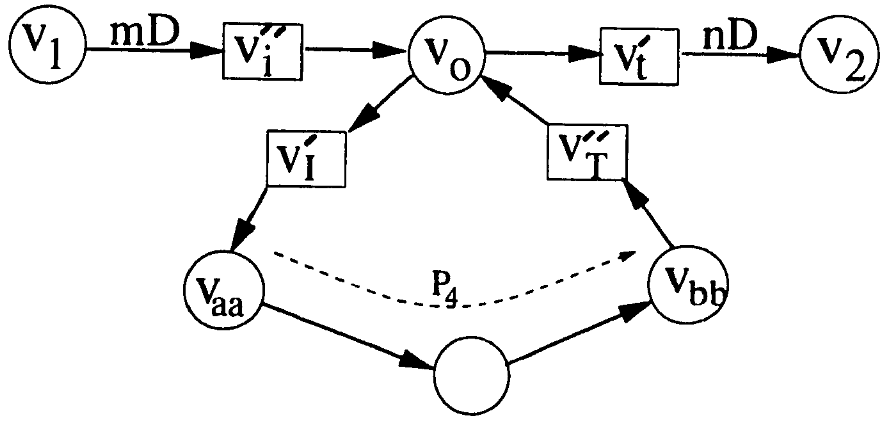
Theorem 2 *Let G' be the graph resulting after contracting a circuit C from the given graph G, and p an elementary path in G. If $p_h^C \subset p$, where p_h^C is a hop-path with respect to C, then there would be no elementary path in G' corresponding to p.*

Proof:

Let p' in G' be the path corresponding to the path p in G. Since $p_h^C \subset p$, $p_h'^C \subset p'$, where $p_h'^C$ is the path corresponding to p_h^C in G'. The path $p_h'^C$, by Lemma 3, is a circuit. Therefore, p' contains a circuit and hence the proof. ■



(a)



(b)

Figure 2.2: (a) A subgraph before and (b) after the contraction of the shown circuit.

Corollary 1 *A circuit C_0 that contains a hop-path with respect to a circuit C in G may not have a corresponding circuit in the graph G' resulting after contracting the circuit C.*

The above theorem showed that some paths have no corresponding paths in the resulting graph after contracting a circuit. The following theorem, however, will show that a path which does not meet the criterion in the above theorem, will have a corresponding elementary path in the graph resulting after a circuit contraction with exactly the same path length.

Theorem 3 *Let v_1 and v_2 be any two nodes such that $v_1, v_2 \in G - C$, where G is the DFG before contracting the circuit C. Then for each path $p_0 = (v_1 \rightarrow v_2) \subset G$, such that p_0 does not contain a hop-path with respect to C, there is a corresponding path $p_0' = (v_1 \rightarrow v_2) \subset G'$ with the same path length, that is, $len[p_0] = len[p_0']$, where G' is the graph resulting after the contraction.*

Proof:

Take any path p_0 whose initial node is v_1 and terminal node v_2. If p_0 has no common subpath with C, then the proof is evident as p_0 will remain unchanged in G'. Assume that p_0 has some common subpaths with C. If there are multiple such disjoint paths, then the path p_0 will contain at least one hop-path, and hence its corresponding path in G' is not elementary as stated by Theorem 2. If there is only one common subpath, say p, assume that its initial node is v_i and terminal node v_t. Without loss of generality, we may assume that v_1 is an immediate precedent to v_i and v_2 is an immediate successor to v_t, that is, $(v_1, v_i), (v_t, v_2) \in E(G)$. It is enough to show that the length of the path $p_0' = v_1, (v_1, v_i''), v_i'', (v_i'', v_0), v_0, (v_0, v_t'), v_t', (v_t', v_2), v_2$ in the graph after contracting circuit C to node v_0 is the same as the length of the path $p_0 = v_1, (v_1, v_t), p, (v_t, v_2), v_2$ in the graph before the contraction, where v_t'' and v_t' are pseudo nodes resulting as per the Contraction Algorithm. The lengths len]p_0[and len]p_0'[are given by

$$len]p_0[= \overline{n}_{(v_1,v_t)} + len[p] + \overline{n}_{(v_t,v_2)},$$

and

$$\begin{aligned} len]p_0'[&= \overline{n}_{(v_1,v_t'')} + d^c_{v_t''} + \overline{n}_{(v_t'',v_0)} + d^c_{v_0} \\ &+ \overline{n}_{(v_0,v_t')} + d^c_{v_t'} + \overline{n}_{(v_t',v_2)}, \end{aligned}$$

where $\overline{n}_{(v_1,v_t)} = -n_{(v_1,v_t)}T$.

But $\overline{n}_{(v_1,v_t'')} = \overline{n}_{(v_1,v_t)}$ and $\overline{n}_{(v_t',v_2)} = \overline{n}_{(v_t,v_2)}$. Moreover, $\overline{n}_{(v_0,v_t')} = \overline{n}_{(v_t'',v_0)} = 0$. Therefore,

$$len]p_0[= \overline{n}_{(v_1,v_t)} + d^c_{v_t''} + d^c_{v_0} + d^c_{v_t'} + \overline{n}_{(v_t,v_2)}.$$

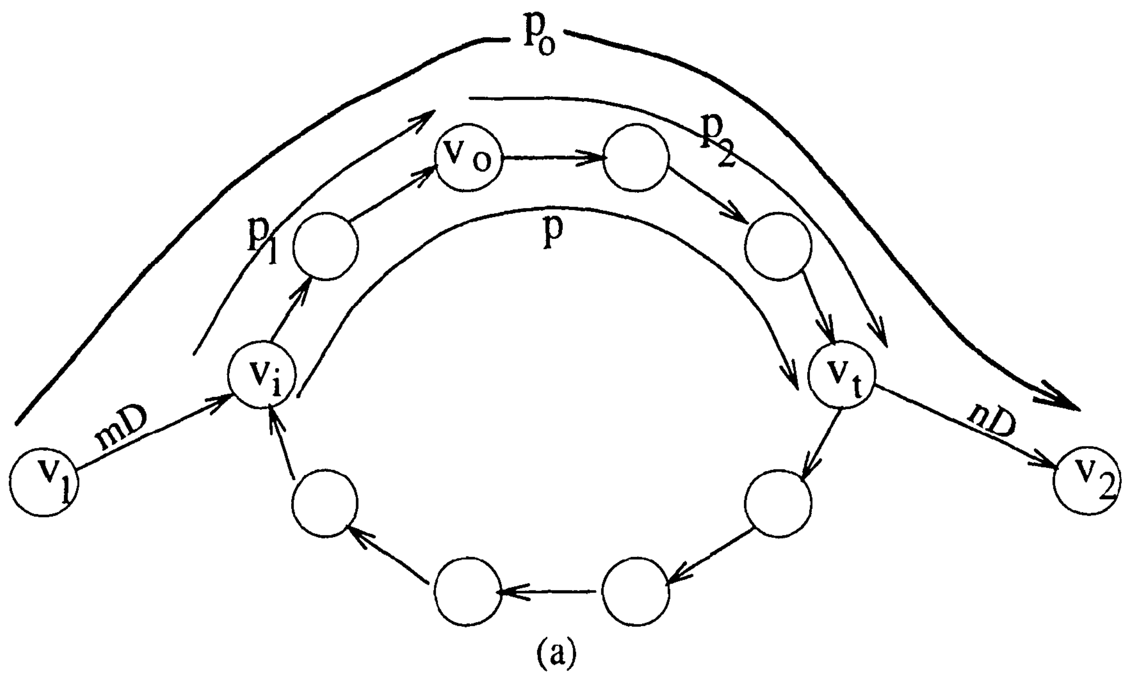
Case 1 ($v_0 \in p$): Let p_1 and p_2 be, respectively, the paths $v_t \to v_0$ and $v_0 \to v_t$ (see Figure 2.3). According to the Contraction Algorithm, the pseudo node v_t'' has a computational delay equal to len[p_1[, and the pseudo node v_t' has a computational delay equal to len]p_2], hence

$$len]p_0'[= \overline{n}_{(v_1,v_t)} + len[p_1[+ d^c_{v_0} + len]p_2] + \overline{n}_{(v_t,v_2)}. \quad (2.10)$$

Using $len[p_1[+ d^c_{v_0} + len]p_2] = len[p]$, (2.10) becomes

$$len]p_0'[= \overline{n}_{(v_1,v_t)} + len[p] + \overline{n}_{(v_t,v_2)} = len]p_0[.$$

Case 2 ($v_0 \notin p$): Let p_1 and p_2 be, respectively, the paths $v_0 \to v_t$ and $v_t \to v_0$ (see (Figure 2.4)). According to the Contraction Algorithm, the pseudo node v_t''



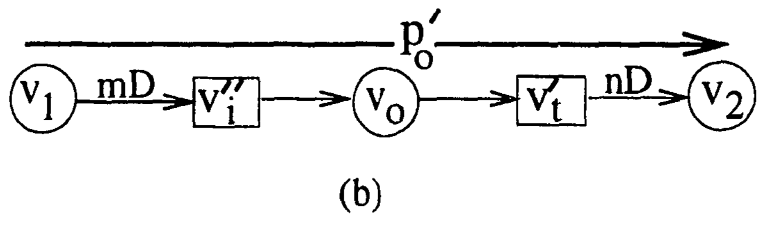
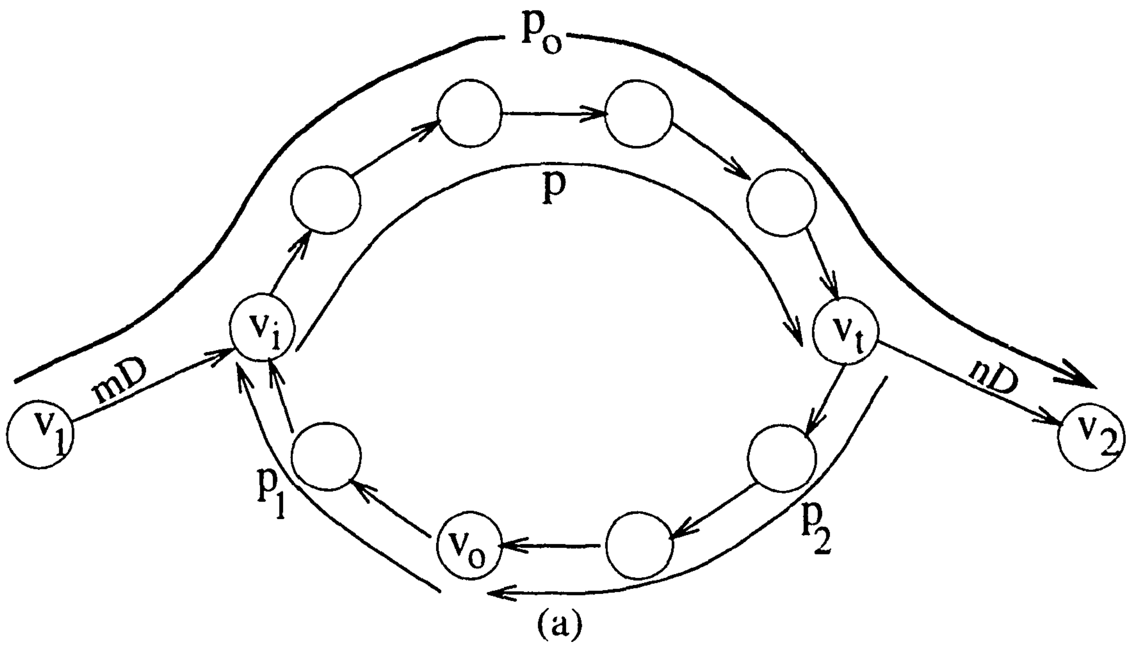


Figure 2.3: (a) A subgraph before and (b) after the contraction of the shown circuit.



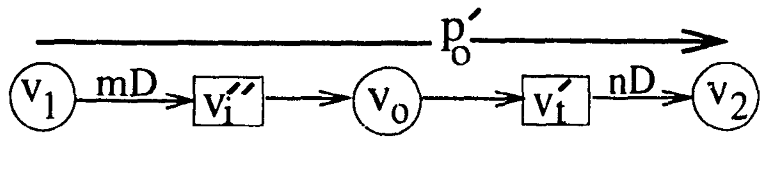


Figure 2.4: (a) A subgraph before and (b) after the contraction of the shown circuit.

has a computational delay equal to $len[p] + len]p_2[$, and the pseudo node v_2' has a computational delay equal to $len]p_1[+len[p]$. Hence,

$$\begin{aligned} len]p_0'[&= \overline{n}_{(v_1,v_t)} + len[p] + len]p_2[+ d^c_{v_0} \\ &+ len]p_1[+ len[p] + \overline{n}_{(v_t,v_2)} \end{aligned} \tag{2.11}$$

Using $len[p] + len]p_1[+ len]p_2[+ d^c_{v_0} = 0$, (2.11) becomes

$$len]p_0'[= \overline{n}_{(v_1,v_t)} + len[p] + \overline{n}_{(v_t,v_2)} = len]p_0[.$$

Hence, whether $v_0 \in p$ or not, len$[p_0']$=len$[p_0]$. ■

Corollary 2 *Let C_0 be a circuit in G, and C_0' its corresponding circuit in G', the graph resulting after contracting a critical circuit C. If C_0 contains no hop-path with respect to C then $ST(C_0') = ST(C_0)$.*

Proof:

A circuit is nothing but a closed path and hence the proof is obvious as per the above theorem. ■

So far, we have shown that a path which has a common subpath with a circuit C to be contracted, will definitely have a corresponding path having the same path length. Further, a path which has more than one subpath common with C (equivalently, it has a hop-path) will have no corresponding path in G', the graph resulting after the contraction. Hence, a path in G either has a corresponding path in G' or has no corresponding path at all. Let C_x be a circuit other than C, the circuit to be contracted. As per Corollary 1, if C_x contains a hop-path with respect to C, then C_x will have no corresponding circuit in G'. Therefore, we have to study the effect of the disappearance of some circuits due to circuit contraction on the scheduling problem. The scheduling problem will only be altered if the flexibility of a node due to the slack times of the existing circuits is higher than the flexibility that node would have if some other circuits have not disappeared. The following theorem will show that this case will never happen.

Theorem 4 *Let G' be a graph resulting after contracting circuit C from a given graph G, and C_x be any other circuit. In G' there is a circuit C_y containing the node set $V(C_x) - V(C)$ such that $ST(C_y) \leq ST(C_x)$.*

Proof:

If $C_x \cap C = \Phi$, the proof is evident and $C_y = C_x$.

If $C_x \cap C \neq \Phi$:

Case 1: *C_x does not contain a hop-path with respect to the circuit C:* Let $C_x = p \cup p_c$, such that $p_c \cap C = p_c$, and $V(p) \cap V(C) = \{v_I, v_T\}$, where v_I and v_T are respectively the initial and terminal nodes of the path p. The path length $len]p[$ will not be affected by the contraction of circuit C, and $len[p'_c] = len[p_c]$ as per Theorem 3, where p'_c is the path in G' corresponding to p_c. Therefore, $ST(C'_x) = -(len]p[+len[p'_c]) = ST(C_x)$, where C'_x is the circuit in G' corresponding to C_x. Hence, $C_y = C'_x$.

Case 2: *C_x contains some hop-paths with respect to the circuit C:* Let

$$C_x = p \cup (\cup_{i=1}^{n} p_i) \cup \left(\cup_{i=1}^{m} p_{h_i}^{C}\right),$$

where p is a path such that $p \cap C = \{v_I, v_T\}$, p_i a path in C ($p_i \cap C = p_i$), and $p_{h_i}^{C}$ a hop-path with respect to the circuit C different from p. Further, let $p_{h_i}^{\prime C}$ be the path in C having the same end nodes as those of $p_{h_i}^{C}$, and

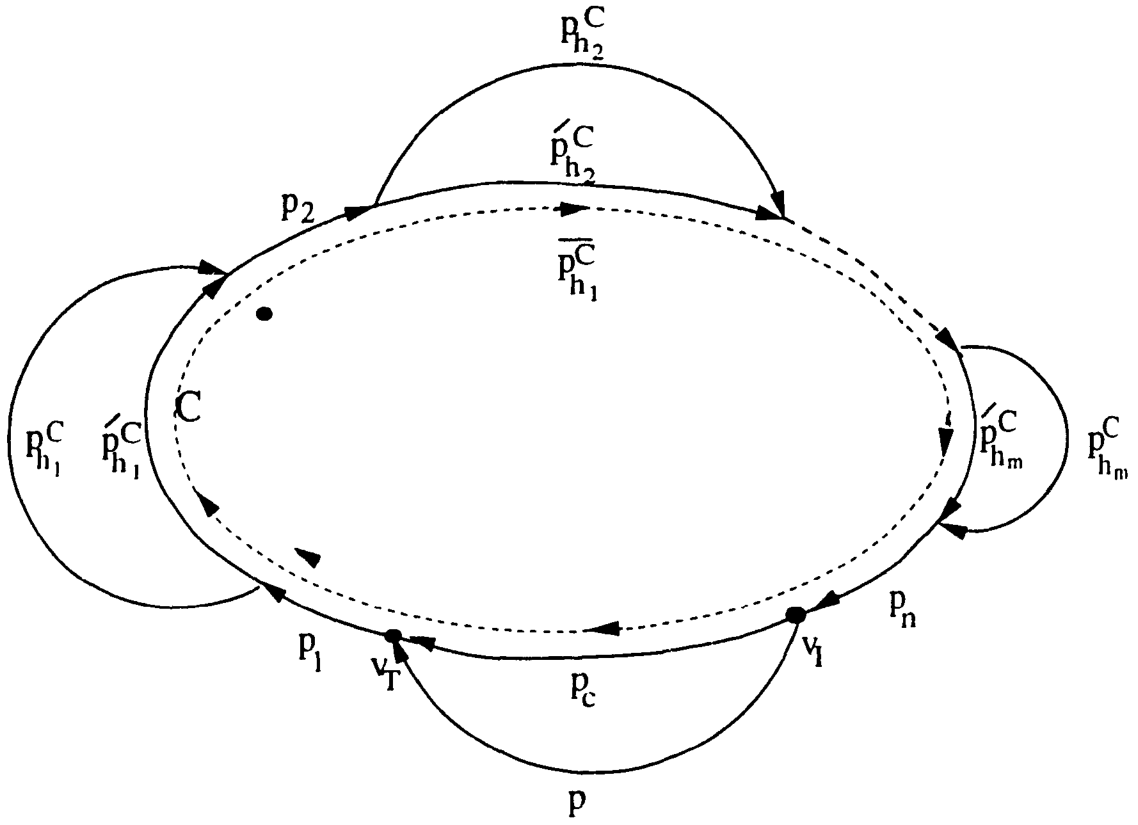


Figure 2.5: An illustration of the definitions used in the proof of Theorem 4.

$\overline{p^C_{h_1}}$ is the path in C whose initial node is the terminal node of $p^C_{h_1}$ and its terminal node is the initial node of $p^C_{h_1}$. Also, let p_c be the path in C whose initial and terminal nodes are v_I and v_T, and $\overline{p_c}$ the path in C whose initial node is v_T, and terminal node v_I. (Figure 2.5 illustrate the above path definitions). Let us refer to the circuit in G' consisting of the path p (excluding it initial and terminal nodes), the pseudo nodes due to the removal of v_I and v_T, and the supernode due to the contraction of C as C_r. As per Corollary 2, C_r will have the same slack time as that of the circuit $p \cup \overline{p_c}$.

$$len[C_x] = len[p[+ \sum_{i=1}^{n} len[p_i[+ \sum_{i=1}^{m} len[p^C_{h_i}[\qquad (2.12)$$

Since $p^C_{h_i}$ together with $\overline{p^C_{h_i}}$ constitute a circuit,

$$len[p^C_{h_i}[+len[\overline{p^C_{h_i}}[\leq 0 \qquad (2.13)$$

However,

$$len[p'^C_{h_i}[+len[\overline{p^C_{h_i}}[= len[C'] = 0 \qquad (2.14)$$

From (2.13) and (2.14)

$$len[p^C_{h_i}[\leq len[p'^C_{h_i}[\qquad (2.15)$$

substituting (2.15) in (2.12)

$$len[C_x] \leq len[p[+ \sum_{i=1}^{n} len[p_i[+ \sum_{i=1}^{m} len[p'^C_{h_i}[\qquad (2.16)$$

The path $\overline{p}$ composed of all the paths in (2.16) excluding p begins at v_l and ends at v_l. As all these paths are in C', $\overline{p}$, in general, may trace C' several times. Since $len[C'] = 0$, regardless of the number of times $\overline{p}$ traces C, $len[\overline{p}[= len[\overline{p_c}[$. Therefore, (2.16) becomes

$$len[C_x] \leq len]p[+len[\overline{p_c}[= len[C_r] \qquad (2.17)$$

which can be rewritten as

$$ST[C_x] \geq ST[C_r] \qquad (2.18)$$

Thus, $C_y = C_r$ and hence, the theorem.

It is clear from Theorems 1 and 3 that the contraction of a critical circuit using the Contraction Algorithm neither affects the scheduling problem, nor alters the other parameters such as the length of an I/O path or the slack time of any remaining circuit. Further, as per Theorem 4, the flexibility of a node may not get increased due to the depletion of some circuits as a result of contracting a critical circuit.

Using the Contraction Algorithm we can contract all critical circuits in the given DFG; however, this is not enough to convert the DFG into an acyclic one. A DFG in general may have non-critical circuits. To contract the non-critical circuits existing in the DFG, we have developed a technique. The critical circuits are first contracted; then, one at a time, the non-critical circuits are converted to critical ones, and contracted. The process of choosing a circuit, converting it to a critical one, and contracting it is repeated until the remaining graph is acyclic. This technique will be treated in the following subsections.

2.4.1 Converting a non-critical circuit to a critical one

A non-critical circuit has a positive slack time, or equivalently, a negative length. Therefore, to convert a non-critical circuit to a critical one, a delay equal to the slack time of that circuit should be induced to its nodes and/or edges. This delay is referred to as the shimming delay. The length of the circuit after introducing this shimming delay becomes zero, since its length to start with is the negative of its slack time. The shimming delay may be either added to the computational delay of the nodes or to the ideal delay of the edges. In general, it is more advantageous to add the shimming delay to the computational delay of the nodes. The shimming delay associated with a node can be considered as its scheduling flexibility upon mapping the time schedule to a processor assignment. On the other hand, the shimming delay added to the edges will not contribute to the scheduling flexibility. Hence, as a rule, we should not assign shimming delay to the edges unless we are forced to do so. As a matter of fact, in some cases, as will be shown later, it is unavoidable to add some shimming delay to the edges.

2.4.2 Contracting a circuit after its conversion to a critical one

Once all the critical circuits in the given DFG G have been contracted, it is clear from Corollary 2 that the non-critical circuits of the resulting graph G' will have the same slack times as their corresponding non-critical circuits in G.

An important point of consideration is the basis on which a non-critical circuit is chosen for contraction at each step once all the critical circuits have been contracted. The circuit with the lowest slack time among the circuits in this graph G' is first chosen for conversion to a critical circuit, followed by contraction using the Contraction Algorithm. The main reason for choosing the circuit with the lowest slack time is that this choice, as will be shown in Theorem 5 below, will never lead to an incomputable graph (a graph with a circuit having a negative slack time). As long as the circuit chosen for conversion is based on the above criterion, any distribution of the shimming delay over the nodes (excluding supernodes) and/or edges of this circuit is a feasible one, given that the total added shimming delay is equal to the slack time of the circuit. This procedure is repeated until the resulting graph becomes acyclic.

Theorem 5 *Let C be a circuit having the lowest slack time $ST(C)$ among the circuits of a computable DFG G, having no critical circuits. Any distribution of the* *shimming delay, equal to $ST(C)$, over the nodes and edges of C will make C critical and at the same time will not result in a negative slack time for any of the other circuits in the resulting graph.*

Proof:

Since G is computable and has no critical circuits, $ST(C_i) > 0$ for every $C_i \subset G$. Assume that C_x is any circuit in G other than C. If C_x has no common nodes or edges with C then the proof is evident. If C_x has some common nodes and edges with C, let the total shimming delay added to these common nodes and edges be d^s. The slack time of the circuit corresponding to C_x in the resulting graph will be

$$ST_{new}(C_x) = ST_{old}(C_x) - d^s, \tag{2.19}$$

where $ST_{old}(C_x)$ and $ST_{new}(C_x)$ are, respectively, the slack times of the circuit C_x before and after the distribution of the shimming delay equal to $ST(C)$, thus rendering C to become critical. However, $d^s \leq ST(C)$, but $ST(C) \leq ST(C_x)$ by assumption. Therefore, $d^s \leq ST(C_x)$. Hence, from (2.19), $ST_{new}(C_x) \geq 0$. ■

One might ask whether it is possible that the optimal schedule is not achievable if the circuits are contracted in the order mentioned above, but possible for some other order. We will now show that for any choice of the sequence of circuit contractions, all valid combinations of the distribution of the shimming delays are possible. Hence, all sequences are equivalent and thus will not affect the generality of the procedure. However, the choice of the sequence of contraction that has been made is only for the sake of simplicity of implementation.

Lemma 4 *Adding a shimming delay to a pseudo node Q is equivalent to adding this delay to each and every edge incident into Q, or to each and every edge incident out of it.*

The proof of this lemma is obvious as the pseudo nodes are not considered for scheduling, and any redistribution of the delay associated with a pseudo-node over the node itself, and/or all the edges incident into it and/or all the edges incident out of it, is a basic permissible retiming operation [72].

Theorem 6 *Let $W = \{d^s_{v_1}, d^s_{v_2}, ..., d^s_{v_N}, d^s_{e_1}, d^s_{e_2}, ..., d^s_{e_M}\}$ be any combination among all the possible combinations for distributing the shimming delays over the nodes and edges of G provided that the circuits are contracted in the order $C_{i_1}, C_{i_2}, ..., C_{i_k}$, where N is the number of nodes, M the number of edges and k the number of circuits in G. If the circuits in the original DFG G are contracted in any other order $C_1, C_2, ..., C_k$, all combinations $\{W\}$ are also possible.*

Proof:

Let G^c be the graph obtained by adding the shimming delays specified by W to the nodes and edges of the given DFG G. Without loss of generality, as per Lemma 4, it is assumed that no shimming delay is added to a pseudo node. It is obvious that all the circuits in G^c are critical. The following procedure will guarantee that the distribution W is achievable if the circuits are contracted in the order $C_1, C_2, ..., C_k$.

(a) Set j=k.

(b) Remove the shimming delays from all the nodes in C_j which are not in any circuit $C_i, (i = 1, 2, ..., j-1)$. Obviously all the circuits $C_i, (i = 1, 2, ..., j-1)$ are still critical after this step.

(c) Set j=j-1 and go to step b if j>0, otherwise STOP.

We get the same distribution of shimming delays by contracting the circuits in the order $C_1, C_2, ..., C_k$ and replacing the shimming delays specified by W in an order which is reverse to that of their removal as per the above procedure. ■

The conversion of a non-critical circuit to a critical one involves associating shimming delays to the nodes and/or edges of that circuit such that the total added shimming delay is equal to the slack time of that circuit. As to which nodes are chosen and as to how much shimming delay is associated to each is a complicated process and, in general, may be done in many different ways. However, we present the following simple algorithm to accomplish this task. The idea behind this algorithm is that the nodes with higher computational delays should be given more flexibility in scheduling.

Conversion Algorithm

1. Let, in a non-critical circuit C, Ω be the sequence of nodes in decreasing order of their computational delays. Remove from Ω all the supernodes. Let $i = 1, 2, \cdots, K$ be the index set of Ω.

2. Set $i = 1$.

3. Set $d^s_{v_i} = \lceil \frac{ST(C)}{|\Omega|} \rceil$.

4. Set $ST(C) = ST(C) - d^s_{v_i}$, $\Omega = \Omega - \{v_i\}$, and $i = i + 1$.

5. If $ST(C) \neq 0$ Goto step 3.

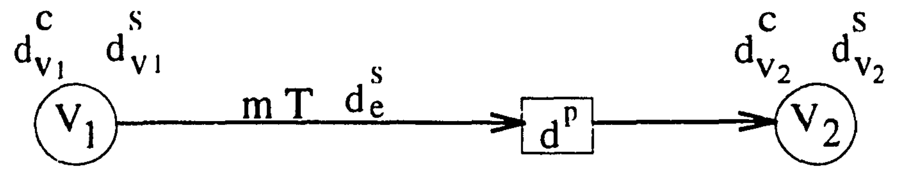
6. For all nodes $v_i \in \Omega$, set $d^s_{v_i} = 0$.

2.4.3 The scheduling algorithm

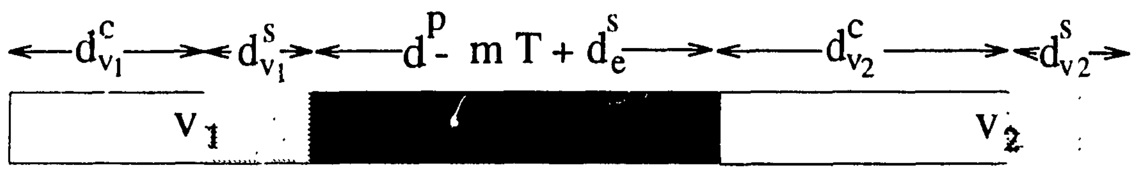
As mentioned earlier, a graph can be transformed into an acyclic one through a sequence of circuit contractions. Any of the known algorithms can then be used to schedule the resulting acyclic graph. Finally, the circuits are scheduled in an order that is reverse to that of their contraction. A circuit can be scheduled by simply starting from the supernode (which is already scheduled) to which the circuit was contracted, and scheduling the rest of the circuit nodes by tracing them sequentially in the direction of the edges. This technique is formalized in the following algorithm.

Scheduling Algorithm:

1. *Initialization:* Let S=Φ, where S is the set of all supernodes. Set i=1.
2. *Cyclic to acyclic conversion:* Repeat the following until G is acyclic.
   (a) Choose the circuit C_i with the lowest slack time. If $ST(C_i) = 0$, go to step (b). Otherwise, convert C_i to a critical circuit using the Conversion Algorithm.
   (b) Contract the critical circuit C_i to one of its nodes, say s_i, as per the Contraction Algorithm.
   (c) Set $S = S \cup \{s_i\}$, $G = G'$, and i=i+1, where G' is the graph resulting after the contraction carried out in step b.
3. *Scheduling the acyclic graph:* Schedule the acyclic graph G according to the precedency constraints using, for example, the critical path method[59, 62].
4. *Scheduling the circuits:* Repeat the following until all the circuits are scheduled, or equivalently until i=0.
   (a) Set i=i-1.
   (b) Schedule the circuit C_i starting from the already scheduled supernode s_i.



(a)



(b)

Figure 2.6: (a) A DFG segment after some circuit contractions, and (b) the corresponding time schedule.

The pseudo nodes are not considered for scheduling. However, their computational delays are considered for the relative scheduling times of the DFG nodes. The shimming delay added to an edge imposes a further delay between the firing times of its end nodes. To illustrate this, consider the example depicted in Figure 2.6(a). The node v_i (i=1,2) has a computational delay equal to $d^c_{v_i}$ and a shimming delay equal to $d^s_{v_i}$. The edge connecting v_1 to the pseudo node (rectangular in the figure) has an ideal delay equal to mT and a shimming delay equal to d^s_e. The pseudo node itself has a computational delay d^p. The corresponding time schedule is shown in Figure 2.6(b). Assume that v_1 fires at time 0, then the firing time of v_2 is $d^c_{v_1} + d^s_{v_1} - mT + d^s_e + d^p$. The shaded portion of the time specified for scheduling v_i represents the shimming delay added to it. Therefore v_1 can be scheduled to fire at any time between 0 and $d^s_{v_1}$ for a period of $d^c_{v_1}$. The actual firing time can be determined when the processor schedule is being formed.

As stated in the Scheduling Algorithm, the first step in scheduling after the contraction of all the circuits of the given graph is scheduling the acyclic graph. The critical path method used for this scheduling is presented in the next section.

2.5 The Critical Path Method

The critical path (CP) technique has been used as a tool for list scheduling. In list scheduling [47, 73], the tasks in a given acyclic DFG are ordered in a such way that when they are executed in that order, each task finds all its operands ready for execution. Further, since for a DFG there are several permissible orders, the list should be ordered such that the total execution time is minimal.

Basically, the CP method assigns weights to the nodes. A node is assigned a weight that is equal to the longest path from the root to that node, where a root is a node having no input edges. To start with, a root is assigned a weight of zero. Then, recursively, each node is assigned a weight when all its precedent nodes have been assigned weights. The weight of a non-root node as computed with respect to a precedent node is the sum of the weight of the precedent node, the delay of the precedent node itself, and the delay of the corresponding edge connecting the two nodes. The weight assigned to a node is the maximal among all such weights, i.e., by considering all precedent nodes. The nodes of the longest I/O path, which begins with the root and ends with the leaf (a node having no outgoing edges), can be determined by traversing the path backwards, that is, starting from the leaf to the node that gave rise to its label and so on, until reaching the root. Without loss of generality, as stated in Chapter 1, we assume that the DFG has only one root and only one leaf.

The techniques, which have been employed to schedule cyclic DFGs using the

CP method, convert the given DFG into an acyclic one by replacing each delay edge by an input node and an output node. Hence the resultant acyclic DFG has no delay edges. On the other hand, our method for such a conversion results in an acyclic DFG that may have delay edges. Therefore, the weights assigned to the nodes, w_i's, are computed as follows

$$w_i = \begin{cases} 0, & \text{if } v_i \text{ is a root} \\ \max_{v_j \in pred(v_i)}(w_j + d_{v_j} + d_{(v_j,v_i)}), & \text{otherwise} \end{cases}$$

where $pred(v_i)$ is the set of precedent nodes of node v_i, $d_{v_j} = d^c_{v_j} + d^s_{v_j}$, $d_{(v_j,v_i)} = d^s_{(v_j,v_i)} - Tn_{(v_j,v_i)}$, and $n_{(v_j,v_i)}$ is the ideal delay of edge (v_j, v_i). Using the CP method described above for the scheduling of an acyclic graph, the weight assigned to a node represents its scheduling time.

If the CP method is used, with the weights as defined above, the acyclic DFG will be scheduled delay-optimally. It is to be noted that scheduling the resulting acyclic DFG delay-optimally does not guarantee, in general, a delay optimal schedule for the given graph, as will be shown in the next chapter. A formal statement of the CP method used in our implementation is presented below.

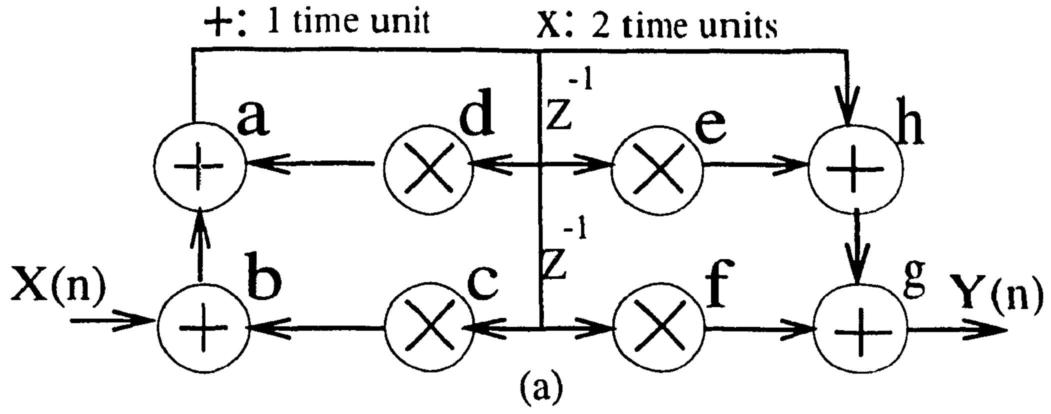
1. *Initialization:* Set $Indeg(v_i)$ to be the in-degree of of the node v_i in the given graph G. Set $t_s(v_0) = 0$, where $t_s(v_0)$ is the scheduling time of the root of the graph, and set $t_s(v_i) = -\infty$ for all other nodes. Push on the stack S all edges incident out of the node v_0.

2. If $S = \Phi$, halt; otherwise pop an edge $e_j = (v_I, v_T)$ out of S.

3. Set $t_s(v_T) = \max(t_s(v_T), t_s(v_I) + d_{v_I} - n_{e_j}T)$, and decrement $Indeg(v_t)$ by 1.

4. If $Indeg(v_t) = 0$, push on S all edges outgoing from the node v_T.

5. Go to step 2.



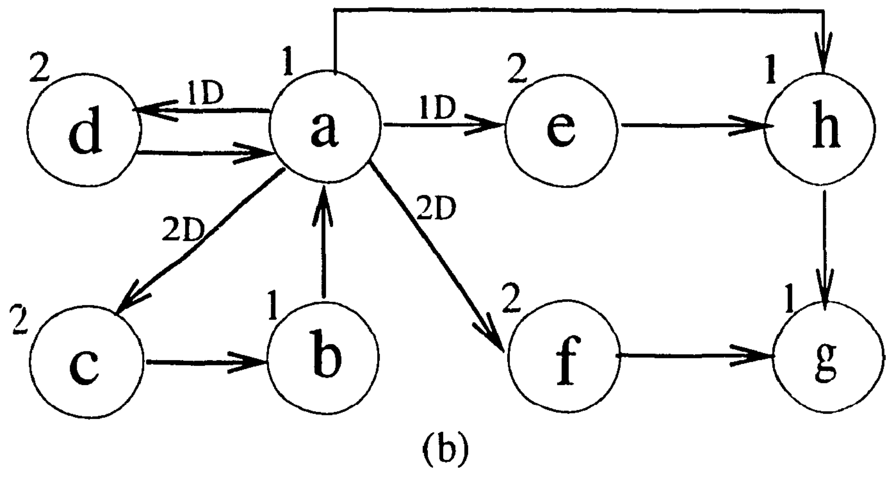


Figure 2.7: (a) A second-order IIR filter, and (b) its DFG representation.

2.6 Examples

Example 1: Second-order IIR filter

The scheduling procedure described in this thesis is now illustrated for the DFG of the second-order IIR filter shown in Figure 2.7(a), which is redrawn in DFG representation in Figure 2.7(b). Let C_1 and C_2, respectively, denote the circuits **a-d-a** and **a-c-b-a**. Then,

$$T_0 = \max\left(\frac{D_{C_1}}{N_{C_1}}, \frac{D_{C_2}}{N_{C_2}}\right) = \max\left(\frac{3}{1}, \frac{4}{2}\right) = 3.$$

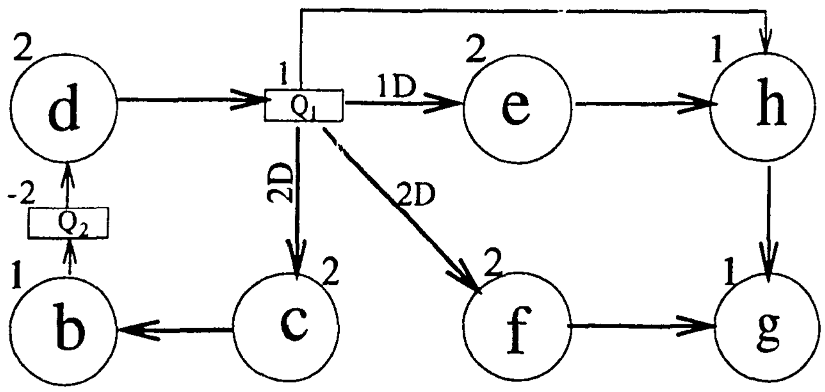


Figure 2.8: The graph after contracting the circuit **a-d-a** to node **d**.

The slack time of the circuit C_2 is $T_0 N_{C_2} - D_{C_2} = 2$. Contracting the critical circuit C_1 to the node **d** will result in the DFG shown in Figure 2.8, where the rectangular nodes are the pseudo nodes created as per the Contraction Algorithm. The delay assigned to Q_1 is equal to $len]d,a]$ which is equal to the computational delay of node a, 1. The delay assigned to Q_2 is equal to $len[a,d[$ which is equal to the computational delay of node a minus the delay of the edge (a,d), that is,

$$len[a,d[= d^c_a - n_{(a,d)}T = 1 - 1 \times 3 = -2$$

To convert the circuit corresponding to C_2 to a critical one, as per the Conversion Algorithm, a shimming delay of 1 is added to each of the nodes **b** and **c**. Contracting the circuit **c-b-**Q_2**-d-**Q_1**-c**, corresponding to C_2, to node **b** results in the acyclic graph shown in Figure 2.9 The time schedule of the acyclic graph involves the scheduling of the nodes **b, h, g, e** and **f**. The critical path method is used to schedule the resulting acyclic graph for iteration n as detailed below.

1. Node **b** is scheduled at time x (any reference time) and given a flexibility of 1; hence, it occupies two cells in the time schedule.

2. At this step, all precedents of Q_3 are scheduled. The scheduling time of Q_3 is

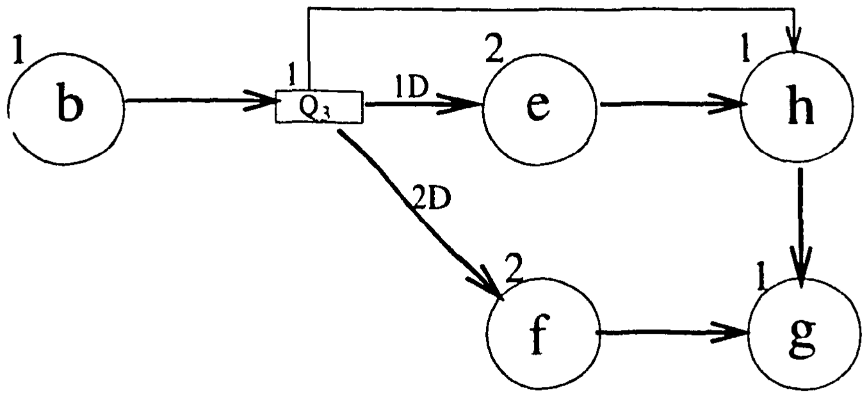


Figure 2.9: The graph after contracting the circuit **c-b-Q_2-d-Q_1-c** (a-c-b-a in the original DFG) to node **b**.

$x + 2$ as it follows node b in scheduling. However, Q_3 is a pseudo node and does not need to be scheduled.

3. Virtually, Q_3 is scheduled at time $x + 2$; therefore, the instance of node e belonging to iteration $(n + 1)$ should be scheduled at time $x + 3$. Hence, node e at iteration n can be scheduled at time x $(x + 3 - T)$. In the same manner, node f is scheduled at time $x - 3$ $(x + 3 - 2T)$.

4. Scheduling the nodes e and Q_3 enables the scheduling of node h. As node e is scheduled at time x for two time units and node Q_3 at $x + 2$ for unit time delay, node h is scheduled at time $x + 3$.

5. The precedents of node g are the nodes h and f; hence, the earliest scheduling time of node g is $x + 4$

Scheduling the circuit contracted last which is b, Q_2, d, Q_1, c relative the scheduled supernode b will result in the sequence of scheduling times $x, x+2, x, x+2, x-3$. Hence node d is scheduled at time x and node c at time $(x - 3)$.

Scheduling the circuit (b, d, b) will result in scheduling node a right after node d, that is, starting at time $x + 2$. The scheduling times of all the nodes are shown

Node	Scheduling Time	
		x = 3
b	x	3
e	x	3
f	x-3	0
h	x+3	6
g	x+4	7
d	x	3
c	x-3	0
a	x+2	5

Table 2.1: The scheduling times of the nodes of the IIR filter

in Table 2.1, and the time schedule is shown in Figure 2.10, where the shaded cells represent the flexibility of the indicated nodes as given by the shimming delays assigned to them.

0	1	2	3	4	5	6	7
			b	b		h	g
f	f		e	e			
c	c	c	d	d	a		

Figure 2.10: A time schedule for the second-order IIR filter.

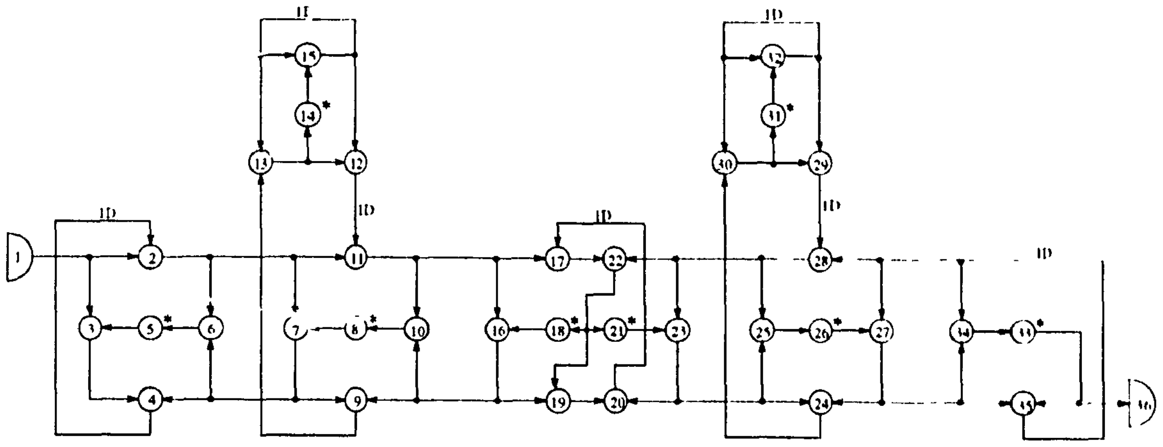


Figure 2.11: A DFG of the fifth-order elliptic filter considered in Example 2.

Example 2: A fifth-order wave digital filter

Consider the DFG of a fifth-order elliptic wave filter shown in Figure 2.11, which has been considered in [68]. It is assumed that an addition requires one time unit and a multiplication (marked by * in figures) requires two time units. This DFG contains 43 elementary circuits, two of them are critical, namely,

$$C_1 = (11, 17, 22, 18, 16, 10, 8, 7, 9, 13, 14, 15, 12, 11), \text{ and}$$

$$C_2 = (3, 4, 2, 11, 17, 22, 18, 16, 10, 8, 7, 6, 5, 3).$$

The iteration period bound is found to be 16.

According to the Scheduling Algorithm, the critical circuits are to be contracted first. Contracting C_1 to node 22 results in the DFG shown in Figure 2.12, where the rectangular nodes are the pseudo nodes created as per the Contraction Algorithm. The circuit $(3, 4, 2, q_1, 22, q_4, 6, 5, 3)$ which corresponds to C_2 in the original DFG is now contracted. The DFG obtained after contracting this circuit is shown in Figure 2.13. Although there were 43 circuits to start with, only five of them needed to be contracted to convert the original DFG into an acyclic graph. The non-critical circuit with the lowest slack time is $C_3 = (22, 21, 23, 25, 26, 27, 24, 30, 31, 32, 29, 28, 22)$,

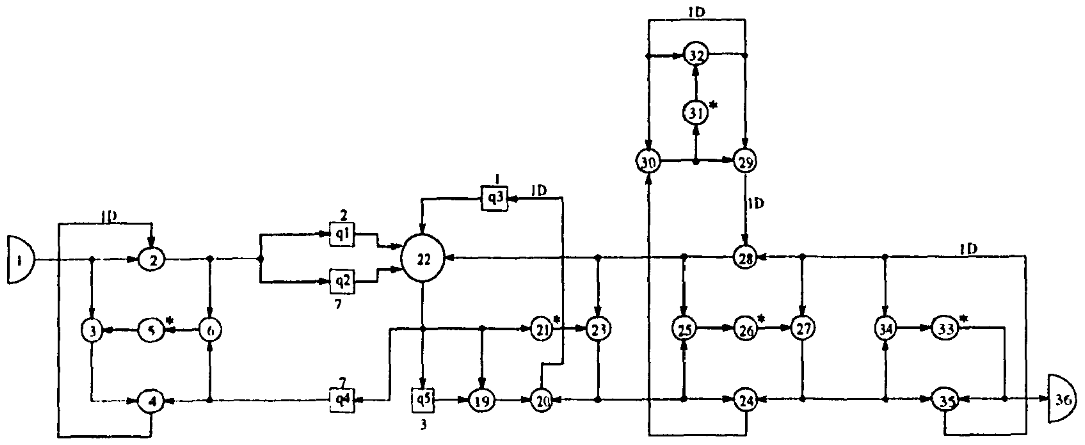


Figure 2.12: The DFG of the wave filter after contracting the circuit C_1.

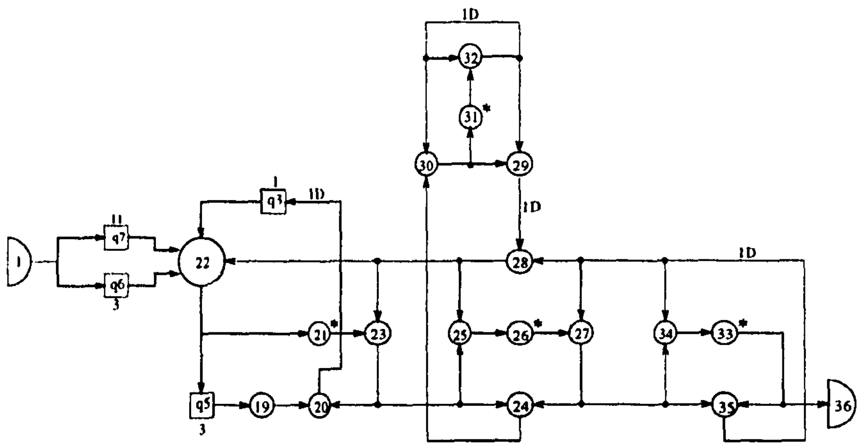


Figure 2.13: The DFG of the wave filter after contracting the circuit C_2.

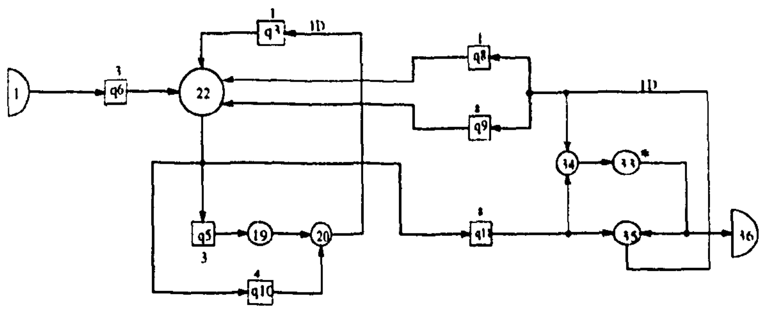


Figure 2.14: The DFG of the wave filter after contracting the circuit C_3.

Circuit	Slack time prior to contraction	Shimming delay distribution
C_1	0	-
C_2	0	-
C_3	1	1 (node 21)
C_4	2	1 (node 33) 1 (node 34)
C_5	9	5 (node 19) 4 (node 20)

Table 2.2: Distribution of shimming delays for the wave filter DFG (T=16).

having a slack time of unity. The Conversion Algorithm proposed in this thesis will assign a shimming delay of unity to node 21 because it has the lowest index among the nodes of the highest computational delay. Contracting C_3 to node 22, after it being converted to a critical circuit, results in the DFG shown in Figure 2.14. The above steps along with the remaining steps of circuit contraction are summarized in Tables 2.2 and 2.3.

Using the critical path method, the resulting acyclic graph (Figure 2.16) is now scheduled. Since there is only one node (node 22) in this graph, it can be scheduled at any place on the time scale. Starting from the scheduled node 22, nodes 20 and 19 are next scheduled upon scheduling circuit C_5. Scheduling circuit C_4 will result in scheduling the nodes 34,33, and 35. Proceeding in this fashion by scheduling the circuits in an order reverse to that of their contraction, the time schedule is

Circuit	Node sequence in the original DFG	Node sequence prior to contraction	Ref. figure
C_1	11, 17, 22, 18, 16, 10, 8, 7, 9, 13, 14, 15, 12, 11	Unchanged	Fig 2.11
C_2	3, 4, 2, 11, 17, 22, 18, 16, 10, 8, 7, 6, 5, 3	$3, 4, 2, q_1, 22, q_4, 6, 5, 3$	2.12
C_3	22, 21, 23, 25, 26, 27, 24, 30, 31, 32, 29, 28, 22	Unchanged	Fig 2.13
C_4	22, 21, 23, 25, 26, 27, 34, 33, 35, 28, 22	$22, q_7, 34, 33, 35, q_8, 22$	Fig 2.14
C_5	17, 22, 18, 16, 19, 20, 17	$22, q_5, 19, 20, q_3, 22$	Fig 2.15

Table 2.3: The contracted circuits in the DFG of the wave filter and their versions prior to contraction.

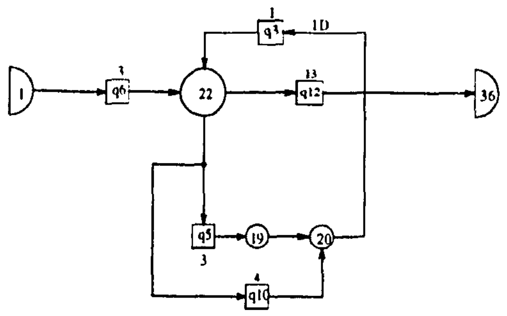


Figure 2.15: The DFG of the wave filter after contracting the circuit C_4.

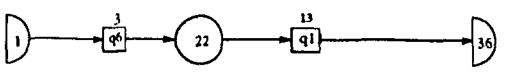


Figure 2.16: The DFG of the wave filter after contracting the circuit C_5.

-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			22				19	19	19	19	19	19	20	20	20	20	20
												34	34	33	33	33	35
		28		21	21	21	23	25	26	26	27	24	30	31	31	32	29
2											6	5	5	3	4		
	11	17		18	18	16	10	8	8	7	9	13	14	14	15	12	

Figure 2.17: A time schedule for the wave filter (T=16).

completed as shown in Figure 2.17. The shaded cells in this schedule represent the shimming delays added to the different nodes.

2.7 Summary

In this chapter, we have presented a new methodology for obtaining a time schedule at compile time for data flow graphs representing iterative DSP algorithms. This methodology is applicable if the given algorithm has no data-dependent decision making operations. The resulting time schedule contains the firing time as well as the scheduling flexibility of each task in the given algorithm. The time schedule is used as an input to the processor assignment algorithm which will be introduced in Chapter 4.

Unlike most heuristics developed to handle the scheduling problem, the proposed algorithm guarantees a solution in view of the theoretical justification presented in this chapter.

We have shown that any cyclic, fully specified DFG can be converted to an acyclic one through a sequence of transformations. This conversion is accomplished by contracting each circuit in the graph to a supernode using the proposed Contraction Algorithm. Then, the resulting acyclic graph is scheduled by using the critical path method. These steps needed to produce the time schedule are presented formally in the Scheduling Algorithm.

The theory developed in this chapter for scheduling incorporates. for the first time, the flexibility of the nodes within the time schedule. This feature, as will be seen in Chapter 4, makes the task of fine tuning the processor assignment relatively simple. Hence, for most cases, our processor assignment is optimal.

To illustrate the theory developed in this chapter we have applied it to two DSP applications, the second one being a widely used benchmark problem.

Chapter 3

Delay Optimal Schedule

The previous chapter dealt with the question of scheduling an algorithm given by a DFG such that the throughput is maximized. In most DSP applications, this is equivalent to maximizing the sampling rate of the input signal. In this chapter, we will address the question of optimizing the input/output delay. For example, consider a single input single output program for which the input stream is $x_1, x_2, x_3, x_4, \cdots, x_n$, and the output stream is $y_1, y_2, y_3, y_4, \cdots, y_n$, n being the number of input samples. The main goal of Chapter 2 was to produce a time schedule that minimizes the time between producing y_i and producing y_{i+1}. (Note that due to the periodic nature of programs, the time between producing any two consecutive outputs is fixed). In this chapter, however, we will present a technique that preserves the optimality criterion used in the previous chapter while at the same time optimizes for the input/output delay. This is equivalent to minimizing the time between consuming x_i and producing y_i.

In a DFG, the minimum input/output delay is the length of the longest path between the input node and the output node. This path is referred to as the critical I/O path. Further, a scheduler that achieves an input/output delay equal to the length of the critical I/O path, is referred to as a delay-optimal schedule. In other words, a schedule is *delay-optimal* if and only if it achieves the delay bound L_0, the minimum delay between consuming an input sample and producing the output sample belonging to the same iteration, as given by

$$L_0 = \max_{p \in I/O\ path} len[p].$$

Recall that

$$len[p] = \sum_{v_i \in V(p)} d^c_{v_i} - T \sum_{e_i \in E(p)} n_{e_i}.$$

where $d^c_{v_i}$ is the computational delay of node v_i, n_{e_i} the number of ideal delays of edge e_i and T the iteration period.

3.1 Rate- and Delay-Optimal Schedule

In order to achieve the delay optimality, the distribution of the shimming delays should not bring the length an I/O path to be greater than the delay bound.

The heuristic described in [43] searches for a schedule that is rate- and delay-optimal at the same time. On the other hand it has been shown in [68] that the schedule might not exist if both the iteration period and the number of processors are fixed, which means that the processor- and rate-optimality may not, in general, be achieved at the same time. Although several researchers have provided scheduling techniques that ensure del y and rate optimality at the same time, none has given a rigorous proof as to whether they can always be achieved simultaneously. We will now establish that the delay-optimality is achievable for any value of T, thus making it possible to attain both rate- and delay-optimality at the same time.

Theorem 7 *A DFG can always be scheduled delay-optimally for an iteration period T_0, the iteration bound.*

Proof:

Proving this theorem is equivalent to establishing that the introduction of the shimming delays needed to convert non-critical circuits to critical circuits can always be carried out without making the length of any I/O path greater than the delay bound L_0.

Assume that C is a non-critical circuit with a slack time $ST(C) = -len[C] > 0$, and that every edge in C is contained in some I/O path. Let $\{p_c\}$ be the set of all such I/O paths. Let S be the total amount of shimming delay that has to be added to the edges of C in order to make every edge of C belong to at least one critical I/O path, without at the same time making the length of any path in $\{p_c\}$ to exceed the delay bound. This can always be done, as is shown by the following procedure.

1. Initialize $E' = E(C)$.
2. Repeat steps 3 and 4 until $E' = \Phi$.
3. Choose any edge $e \in E'$. Let $\{p_e\}$ be the set of all I/O paths such that $e \in E(p_e)$. Let $p'_e \in \{p_e\}$ be a path of maximal length. If p'_e is non-critical, make it critical by adding to edge e, a shimming delay equal to $L_0 - len[p'_e]$.
4. Set $E' = E' - E(p'_e)$.

If $S \geq ST(C)$, then the shimming delay can always be distributed in such a way that C becomes critical without making the length of any of the paths $\{p_e\}$ exceed the delay bound. We will now show by contradiction, that $S \nless ST(C)$.

Let $S < ST(C)$. Let $ST'(C)$ be the slack time of the circuit C, after the distribution of the total shimming delay S over the edges of C, as per the procedure

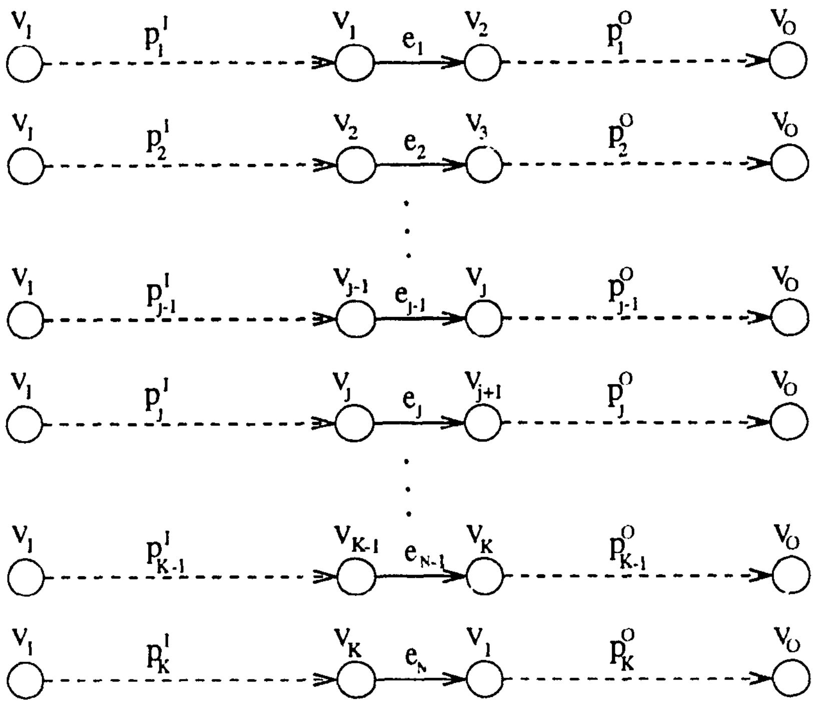


Figure 3.1: An illustration used in the proof of Theorem 7.

described above. Then we have,

$$ST'(C) = ST(C) - S > 0.$$

It is clear that each edge $e_j = (v_j, v_{j+1})$, $j = 1, ..., K$, of the circuit C is contained in at least one critical I/O path, where K is the total number of edges in the circuit C. Figure 3.1 shows a critical path for each of the edges e_j. Let p_j^I stand for the path from the input node v_I to the node $v_j \in E(C)$, while p_j^O for the path from node v_{j+1} to the output node v_O. Further, let the path $p_{e_j} = (v_j, e_j, v_{j+1})$. It should be observed that the critical I/O paths associated with the edges $\{e_j\}$ need not all be distinct.

Consider the critical I/O path $p_j^I \cup p_{e_j} \cup p_j^O$ and the path $p_j^I \cup p_{j-1}^O$. Then we have the inequality,

$$len[p_j^I] + len[p_{e_j}] + len[p_j^O] \geq len[p_j^I] + len[p_{j-1}^O]. \tag{3.1}$$

Adding (3.1) for $j = 1, 2, ..K$, and recognizing that $p_0^O = p_K^O$, we get

$$\sum_{j=1}^{K} len[p_{e_j}] \geq 0.$$

Since

$$\sum_{j=1}^{K} len[p_{e_j}] = len[C] = -ST'(C),$$

we have

$$ST'(C) \leq 0.$$

This contradicts the assumption that $S < ST(C)$, hence the theorem. ∎

As per the above the theorem, a DFG can always be scheduled rate and delay-optimally at the same time. However, it is not clear from the theorem how we can achieve delay optimality. Recall that rate-optimality is achieved in view of the fact that the iteration bound is used to compute the delays of the different edges. These delays are used for scheduling the different nodes of the DFG. The question as how to obtain the delay optimality is treated in the next section.

3.2 Obtaining Delay Optimality

The technique employed by Gelabert and Barnwell [43] to ensure the delay optimality is to enumerate all I/O paths and keep track of their lengths as the nodes are scheduled. However, this technique is cumbersome, since every time a node is scheduled, each I/O path containing that node has to be altered in terms of the flexibility of its nodes. Further, enumerating all I/O paths, in the worst case, has an exponential complexity.

We will now present a simple technique to ensure the delay optimality, which is achieved if and only if the output node is scheduled to finish execution exactly after L_0 time units of the scheduling time of the input node. In other words, a schedule is delay-optimal if and only if all the nodes on the critical path are scheduled without any flexibility. As shown in Chapter 2, the nodes of a critical circuit have no flexibility in being time scheduled. Therefore, if all the nodes lying on the critical path belong to a critical circuit, then their scheduling times are fixed, and hence, the schedule of the DFG is automatically delay-optimal provided that it has already been scheduled rate-optimally. However, the case when all the nodes of the critical I/O path are in a critical circuit is a special case and an effort has to be made to ensure delay-optimality in the general case. If it is possible to alter the graph such that a critical circuit is introduced to contain all the nodes of the critical I/O path without affecting the data dependency among the nodes, then we can always achieve delay optimality by introducing such a circuit.

Before we address this question of introducing such a critical circuit, let us consider the following scheduling concept. Understanding this concept will help in proposing a method of introducing such a circuit. Let x and y be two nodes such that there is a path p_{xy} from x to y whose length $len[p_{xy}[$ is greater than or equal to the length of any other path from x to y. If x is scheduled at time 0, then node y should be scheduled at a time greater than or equal to $len[p_{xy}[$ to preserve the integrity of the data dependency. Assume now that node y is scheduled at time $len[p_{xy}[$. Let us now introduce a new path from node y to node x, say p_{yx}, such that x and y retain their scheduling times. Fixing the scheduling of node y at time $len[p_{xy}[$ will restrict node x to be scheduled at time 0 or earlier, but we require x to be scheduled at time 0; hence,

$$t_s(x) = t_s(y) + len[p_{yx}[= 0,$$

where $t_s(x)$ is the scheduling time of node x. Therefore,

$$len[p_{yx}[= -len[p_{xy}[. \tag{3.2}$$

Subtracting the computational delay of node y from both sides of (3.2), we get

$$len]p_{yx}[= -len[p_{xy}]. \tag{3.3}$$

Hence, we can add a path from node y to node x whose length is $-len[p_{xy}]$ without affecting the scheduling of the nodes x and y. The only implication of introducing such a path is that node y is scheduled exactly at time $len[p_{xy}[$, provided that node x has been scheduled at time 0.

This leads us to the conclusion that we can achieve the delay optimality by introducing a path from the output node to the input node whose length is equal to the negative of the delay bound. (This path will be referred to as a delay-optimality path). By substituting the input node I for x and the output node O for y, we get

$$len]p_{OI}[= -L_0. \tag{3.4}$$

Introducing the path p_{OI} results in a new circuit that consists of this path and the critical I/O path. Let us refer to this circuit as a delay-optimality circuit and denote it by C_d. The slack time of this circuit is

$$ST(C_d) = -len[C_d] = -(len[p_{IO}] + len]p_{OI}[) \tag{3.5}$$

where p_{IO} and p_{OI} are respectively the critical I/O path and the delay-optimality path. Substituting (3.4) in (3.5) yields

$$ST(C_d) = 0, \tag{3.6}$$

and hence C_d is a critical circuit. As a matter of fact, for every I/O path, there will be a new circuit consisting of that path and the delay-optimality path. Let p'_{IO} be any I/O path (not necessarily critical), then the slack time of the circuit C'_d consisting of this path and the delay-optimality path is

$$\begin{aligned} ST(C'_d) &= -(len[p'_{IO}] + len]P_{OI}[) \\ &\geq -(len[p_{IO}] + len]p_{OI}[) \\ &\geq 0 \end{aligned}$$

Therefore, introducing a delay-optimality path will not result in a non-computable circuit.

One possible representation of the delay-optimality path is the one that consists of a pseudo node v_d^p of a computational delay of $-L_0$, a zero-delay edge from the output node to v_d^p, and a zero-delay edge from v_d^p to the input node. As explained earlier, adding such a path will increase the number of circuits in the graph by the number of I/O paths, and hence increase the time complexity of the circuit finding algorithm.

This problem of increased complexity resulting from ensuring delay-optimality, can be reduced, substantially in most applications, using the following approach.

1. Compute T_0 by finding the circuits in the given graph .

2. Compute L_0 (for example, by using, Bell-Ford algorithm [42]).

3. Add the delav optimality path and contract the delay-optimal circuit.

4. Find the circuits of the resulting graph.

5. Perform the Scheduling Algorithm.

It should be noted that in most DSP applications, it may not be necessary to compute T_0 to find L_0. and thus step 1 is not needed. In such cases, T_0 is computed in step 4. This is due to the fact that in most applications, there is a dependency between the output at an iteration and the input for the same iteration. That is, there is at least one path from the input node to the output node which contains no ideal-delays. The length of an ideal-delay-free path is not a function of the iteration period. In most cases the critical I/O path 's one of these ideal-delay-free paths, and can be determined without a knowledge of the iteration period. Further, even if T_0 is needed in order to compute L_0, it may not be necessary to find all the circuits in the given graph, (see, for example [74, 75]).

3.3 Effect of Delay-Optimality on Node Flexibility

As mentioned previously, even though it is more helpful to add the shimming delays to the nodes rather than the edges, it may not always be possible. For example, a non-critical circuit in which each of its nodes belongs to some critical I/O path, may not receive any shimming delay on its nodes. The shimming delay of such a circuit should be distributed on the edges which do not belong to any critical I/O path. A non-critical circuit whose each and every node and edge belongs to a critical I/O path, may not exist as per Theorem 7. As an example of a case where the shimming delay has to be assigned to an edge is depicted in Figure 3.2. The circuit ABFA is critical with a loop-bound of 5, while the circuit BCDEB is non-critical with a slack time of unity. The only critical I/O path is ABCDEG with a path length of 7. The shimming delay, equal to the slack time, has to be added on the edge (E,B); otherwise, the length of the critical I/O path will increase by unity. Hence, ensuring delay optimality is sometimes at the expense of the flexibility that can be assigned to the nodes of the DFG, if the delay-optimality was not required.

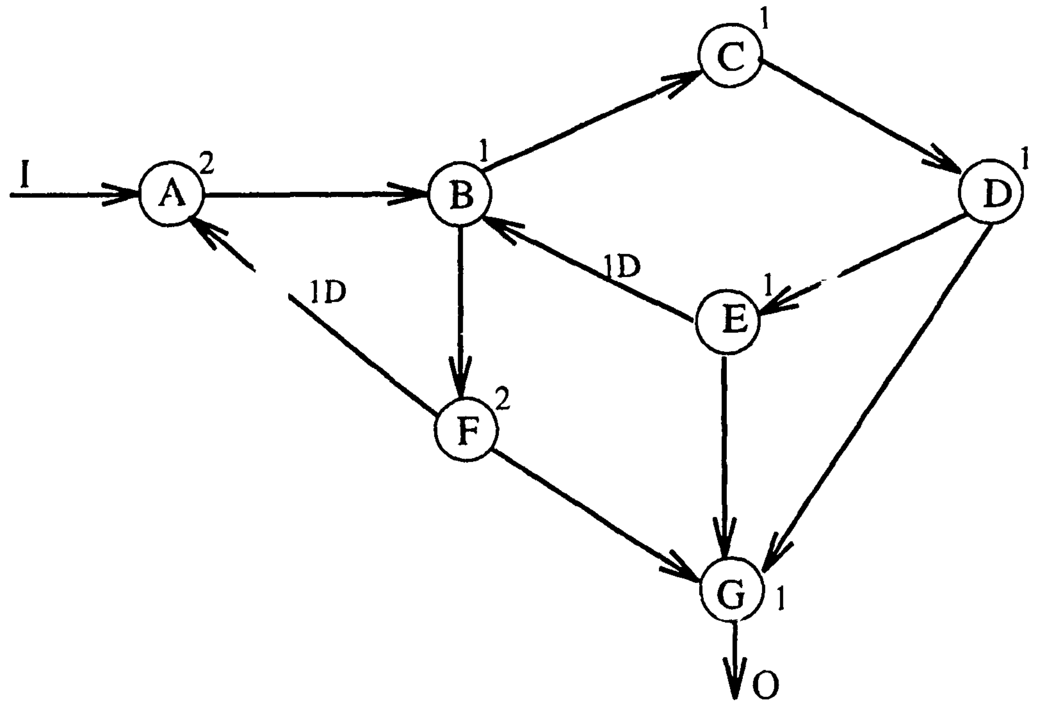


Figure 3.2: A DFG in which the nodes of the non-critical circuit are all contained in a critical path.

3.4 Example

A fourth-order lattice filter

Consider the fourth-order all-pole lattice filter shown in Figure 3.3, considered in [68]. It is assumed as in [68] that an addition requires one time-unit and a multiplication five time-units. The critical path of this graph is $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)$ which has a length of 28. To convert the DFG to a delay-optimal graph (a graph whose schedule is delay optimal regardless of the scheduling procedure used), we add a pseudo node v_d^p with a computational delay of -28 and two delay-free edges incidents on it, one coming from the output node and one going to the input node. The graph after introducing this delay-optimality path (dashed in the figure) is depicted in Figure 3.4. Adding this path will create the critical circuit

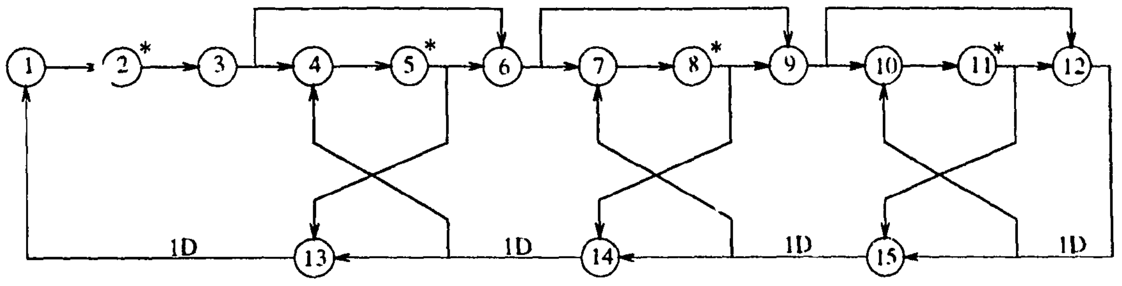


Figure 3.3: A DFG of a fourth-order all-pole lattice filter considered in Example 2.

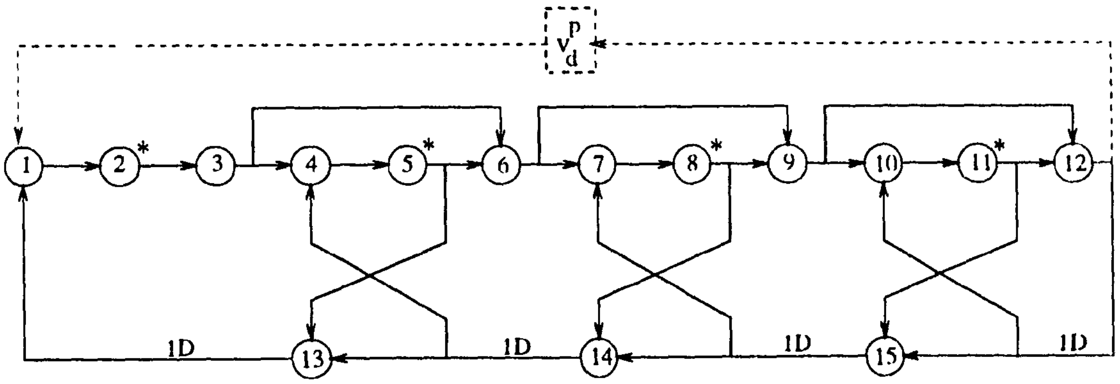


Figure 3.4: The DFG of a fourth-order all-pole lattice filter after introducing the delay-optimality path.

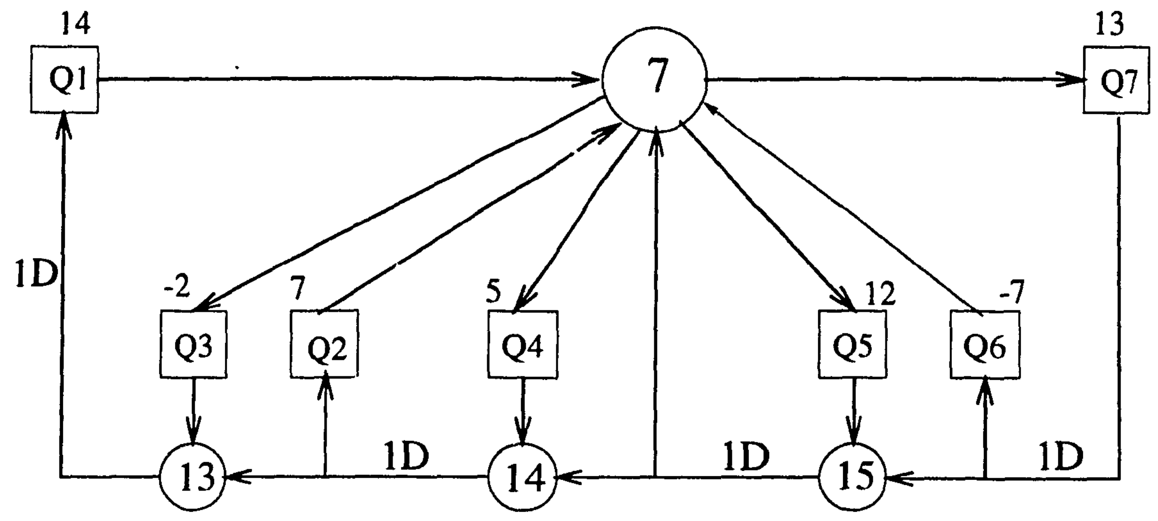


Figure 3.5: The DFG of the lattice filter after contracting the delay-optimality critical circuit.

$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, v_d^p, 1)$. The DFG resulting due to the contraction of this circuit to node 7, is shown in Figure 3.5. This DFG can now be analyzed for circuit extraction and consequently for finding the iteration period. The iteration period is found to be 14. The contraction steps of the remaining circuits are illustrated in Figures 3.6, 3.7, and 3.8.

Although the graph in Figure 3.8 has two circuits, no further contraction is needed since the graph has only one node. node 7 can now be scheduled at any reference time, which is chosen to be 14. Scheduling the last contracted circuit which is $(7, Q_5, 15)$ will add node 15 to the time schedule. Since $len[p = 7 \rightarrow 15[= 13$, node 15 is scheduled at time 27 (14+13). Upon the scheduling of the circuit $(7, Q_4, 14, Q_2)$, node 14 is time scheduled. node 14 has to be scheduled 5 time units after the end of the scheduling time of node 7, which is time 20. Similarly, scheduling the circuit $(7, Q_3, 13, Q_1)$ will add node 13 to the time schedule at time 13. Finally, the delay-optimality circuit is scheduled which will add the rest of the nodes to the time schedule. These nodes are scheduled consecutively with node 7 as a reference, the supernode to which the delay-optimality circuit was contracted.

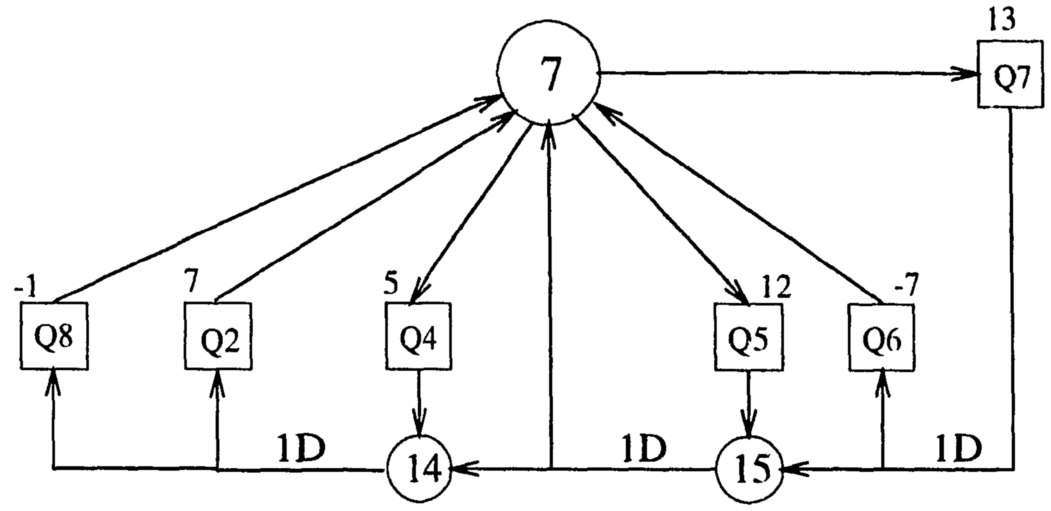


Figure 3.6: The DFG of the lattice filter after contracting the circuit $7, Q_3, 13, Q_1$ corresponding to the circuit $1,2,3,4,5,13$ in the original DFG.

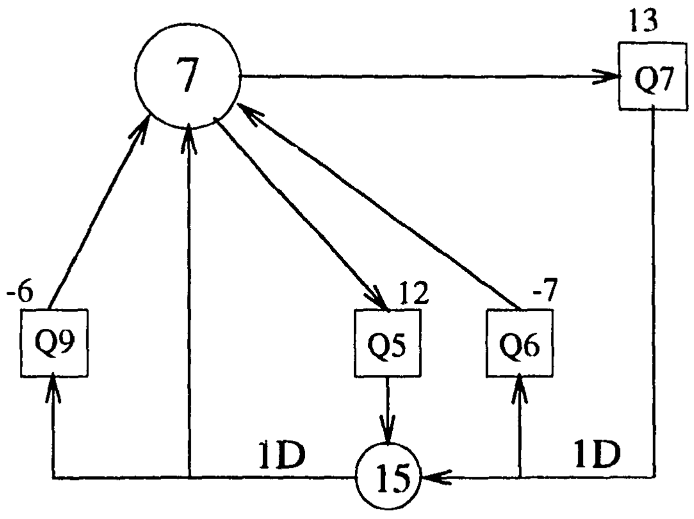


Figure 3.7: The DFG of the lattice filter after contracting the circuit $7, Q_4, 14, Q_2$ corresponding to the circuit $4,5,6,7,8,14$ in the original DFG.

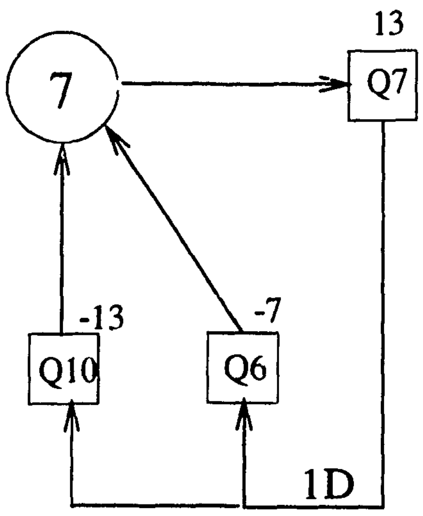


Figure 3.8: The DFG of the lattice filter after contracting the circuit $6, Q_5, 15$ corresponding to the circuit $7, 8, 9, 10, 11, 15$ in the original DFG.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
													13	7						14							15
1	2	2	2	2	2	3	4	5	5	5	5	5	6		8	8	8	8	8	9	10	11	11	11	11	11	12

Figure 3.9: A time schedule for the lattice filter DFG.

The time schedule for the the DFG of this example is shown in Figures 3.9. Obviously this time schedule is delay-optimal as the input node (node 1) is scheduled at time 0 and the output node (node 12) is scheduled at time 27, making the total delay equal to 28, which is equal to the delay bound.

3.5 Summary

In this chapter, we have proposed a technique to transform a given graph to a delay-optimal graph. This new graph has the characteristic that when it is scheduled rate-optimally, the resulting schedule is delay-optimal as well. To carry out this transformation, we have first proved that it is always possible to achieve both rate and delay optimality at the same time. Then, we have proposed a simple (in terms of implementation and complexity) method to achieve delay optimality. This technique is based on the fact that the nodes of a critical circuit have no flexibility in scheduling. Thus, modifying the graph so that each node/edge on the critical I/O path belongs to some critical circuit, will ensure delay-optimality. The simplest way to make every such node/edge to belong to a critical circuit is to introduce a single circuit that contains the entire critical I/O path.

Finally, to illustrate the algorithm presented in this chapter, we have applied it to the case of scheduling a lattice filter, which is a well-known benchmark problem.

Chapter 4

Processor Assignment

In scheduling on a multiprocessor system, the goal is to obtain a code for each processor in the system such that the task of the given algorithm is achieved. The first step in achieving this goal is to distribute the different tasks of a given program on the processors of the given system. As mentioned in Chapter 2, scheduling may belong to one of the five categories, namely, fully dynamic, static assignment, self timed, fully static, and cyclo-static. Given a time schedule as obtained from the Scheduling Algorithm presented in Chapter 2, a processor assignment can be implemented for any of the last three categories. In this thesis, however, we are interested in finding a fully static processor allocation scheme.

In static scheduling, the tasks of a given program are assigned to the processors at compile time. Hence, there is no overhead due to run-time allocation of tasks on the processors. Furthermore, the data dependency is also resolved at compile time, thus ensuring the availability of the operands necessary to compute the operations. In our case, the availability of operands is guaranteed by the correctness of the time schedule. Having produced the time schedule, the scheduling problem is reduced to assigning the tasks in the time schedule to the processors in such a way that the timing specified by the time schedule is preserved. Furthermore, the number of processors needed to assign all the tasks should be minimized

4.1 Processor Optimality

In practice, the number of processors in a given system is fixed and hence the notion of minimizing processors in not clear. However, in view of the synchronization overhead between processors, if a task can be scheduled optimally on n processors, then using more than n processors will, in general, degrade the overall performance. Further, in systems where multiprogramming is possible, the extra processors may be used to perform other tasks. For special-purpose architectures, minimizing the number of processors may be done at the design time. The minimum number of processors is computed for different applications, and the design of the processing unit is done accordingly. In cases where the minimum number of processors required to schedule a certain DFG is greater than the number of processors in a given system, other optimality criteria, rate-optimality in particular, need to be sacrificed. As to how we can achieve a compromise between the number of processors needed and satisfying certain other optimality criteria is examined later in this section.

Given a time schedule that achieves an iteration period of T and an I/O delay of L_0 for a given DFG G, we want to find the minimum number of processors needed to execute the program represented by G. The minimum number or processors needed is referred to as the *processor bound* (P_0), and the assignment achieving this bound is referred to as the *processor optimal assignment.*

In a single processor system, the iteration period is obviously equal to the total computational delay of the graph G. That is, an iteration may not start unless all the computations belonging to the previous iteration have been completed. Let the total computational delay of the graph G be D_G. Then, in a single processor system, the iteration I_0 equals D_G.

Suppose we are required to find the processor bound, when the iteration period T is a given fixed quantity. In T time units, P processors can perform at most a total computation of TP. Hence, for a given total computational delay D_G, at least D_G/T processors are needed; D_G/T is not necessarily an integer quantity, whereas the number of processors has to be an integer. Hence, the processor bound can be expressed as

$$P_0 = \lceil D_G/T \rceil. \tag{11}$$

For an acyclic DFG, T is usually an input parameter which is constrained by the hardware resources available for computation, whereas for a cyclic DFG it is the iteration period bound T_0, unless T is required to be greater for a specific application.

The processor bound is not always achievable as the precedency constraints might limit the maximum parallelism. Since P_0 could be greater than D_G/T, the average processor efficiency may be less than 100%. If D_G/T is a non integer quantity, then a 100% processor-efficiency may not be achieved, because the algorithm with such a bound can not be implemented on less number of processors for the same value of the iteration bound. The maximum computational delay that can be achieved by P_0 processors in T units of time is P_0T; however, the actual computation performed is D_G. Hence, the maximum average processor efficiency can be expressed as

$$\begin{aligned}\eta &= \frac{D_G}{P_0T} \\ &= \frac{D_G}{\lceil D_G/T \rceil T}\end{aligned}$$

For example, if $D_G = 9$ time units and $T = 2$ time units, then $P_0 = 5$; hence $\eta = 90\%$.

If in a given system the number of processors is less than the processor bound, then there is no choice but to sacrifice the rate-optimality, that is, T has to be greater than T_0. Let P be the number of processors such that $P < P_0$. The minimum iteration period for this system is

$$T = \lceil D_G/P \rceil > \lceil D_G/P_0 \rceil = T_0$$

The DFG model presented in this thesis does not take into account communication delays for transferring data between the processors, as if all the processors are on a single chip. The assumption that all the processors are on a single chip is a reasonable one in view of the simple node operations involved in DSP applications, thus allowing the implementation of multiple processors on a single chip. The scheduling theory developed in this thesis is valid on a parallel system under the following assumptions.

1. All processors are identical,

2. each processor can uniformly communicate with all other processors,

3. the access time of an operand from the memory of another processor is equal to the access time of an operand from a local memory, and

4. processors are dedicated to the scheduled program, i.e. they are uninterruptible.

Since in our scheduling theory, no preference is assigned between processors and an operation has a fixed computational delay regardless of the processor executing it, the processors are assumed to be identical. This is clear from the time schedule, which provides all the timing information without any consideration to the processor assignment. Each processor should uniformly communicate with all the other processors, so that the transfer of data between processors can take place in a fixed time, regardless of the spatial relations among the processors. Because of the nature of real time applications, processors should not be interrupted. For off-line applications, a processor may be interrupted, but in this case, all processors have to be halted until the interrupt is handled. Then, all processors should resume processing simultaneously. This is important, because the synchronization among the processors and the data-dependency resolution have to be maintained by the relative firing times of the different operations that could be running on different processors.

4.2 Processor Assignment Algorithm

Given the time schedule, produced by the Scheduling Algorithm, it is required to find the best possible fully static processor assignment. Let us represent the processor schedule by a processor assignment matrix PAM of order $(P \cdot T)$, where P is the number of processors needed to assign the tasks associated with the nodes, and T is the iteration period. In the PAM, the tasks associated with row i are those which are to be allocated to processor P_i, while the tasks associated with column j are those to be executed at time j of each iteration by the different processors. It is to be noted that the PAM represents one iteration of program execution. Hence, a schedule for a periodic program is a periodic extension of the PAM in the time space, that is, repeating the PAM matrix horizontally.

As we are interested in periodic programs, it is adequate to find a single iteration schedule, which we shall refer to as the PAM. Despite the fact that a PAM represents a single iteration, the operations in this matrix could belong to different iterations. For example let A and B be two operations in a PAM. It is possible that A in the PAM belongs to iteration m while operation B belongs to iteration n, where m and n are two iteration indices that could be different. Therefore, along with the operation reference, a PAM should contain information about the iteration index for each and every operation. The iteration index will be represented by a superscript, that is, a superscript of an operation in the PAM represents the iteration index of that operation.

The operations in a time schedule belong to a single iteration. That is, the time displacements between the different operations represent the data dependency among these operations to produce an output for an input belonging to the same iteration. In general, the time schedule covers a time span greater than the iteration period, and hence the PAM would contain operations belonging to different iterations. The number of columns of the PAM corresponding to a time schedule should be equal to T.

The basic operation of constructing the PAM from a given time schedule is to shift each operation by a multiple (positive or negative) of T time units such that all operations fit in T consecutive columns. For example, let x be an operation starting at the time unit $mT + n$ of the time schedule, where m and n are integers and $0 \leq n < T$. In the PAM, operation x should start at location n with a superscript of $-m$, provided the current iteration is assumed to be iteration 0. If the computational delay of a node requires that its allocation extends beyond the last column of the PAM, the excess part of the operation should circularly wrap to the first column of the PAM. For example, let x be an operation with a computational delay of 3 whose starting time in the time schedule is at time $(T - 1)$. Further, let the PAM be the matrix resulting after shifting all operations in the time schedule such that they fit in between time 0 and time $(T - 1)$. Then, node r will occupy a $(T - 1)$th time slot with a superscript k and the first and second time slots with a superscript $k - 1$, where k is the index of a reference iteration.

Although shifting all operations to fit in a T-column matrix will produce a valid PAM, the above technique does not take processor minimization into account, that is, minimizing the number of rows in the PAM. To attempt to minimize the number of processors, we first make the following observations.

1. The nodes should be scheduled in an increasing order of their flexibility because at early stages of scheduling, the PAM has more vacant time slots, and hence a node with a small scheduling flexibility can be scheduled rather easily.

2. Nodes with higher computational delays are more critical in scheduling than those with lower computational delays. Hence, nodes having the same flexibility are scheduled in a decreasing order of their computational delays.

3. A convenient place for scheduling a node is the best fit place, a place in which no node with greater computational delay may fit.

4. Producing gaps in the schedule should be avoided as much as possible. (A gap is an entry or a sequence of entries in the PAM in which no unscheduled node may fit).

Using the basic operation of mapping a time schedule to a processor assignment, and taking note of the above observations, we now establish the following processor assignment algorithm.

Processor Assignment Algorithm:

1. Let $V(G) = \{v_1, v_2, \ldots, v_N\}$ be the node set of the given DFG G, such that $d^s_{v_i} \leq d^s_{v_{i+1}}$ and $d_{v_i} \geq d_{v_{i+1}}$ if $d^s_{v_i} = d^s_{v_{i+1}}$ for all $(i = 1..N)$, where $d^s_{v_i}$ is the shimming delay assigned to v_i and d_{v_i} is the computational delay. Compute the processor bound P_0 using (4.1) and initialize the PAM to an empty matrix of order $P_0 \cdot T$. Let $\xi(j, v_i)$ be a boolean function giving a *true* value if and only if node v_i can be assigned to row j of the PAM. Let the boolean function $Tight(j, v_i)$ be *true* if node v_i can be assigned to row j, say between PAM(j, x) and PAM(j, y), such that PAM$(j, x-1)$ and PAM$(j, y+1)$ are occupied, otherwise $Tight(j, v_i) =$ *false*

2. For i= 1 to N, repeat the following.

3. Search for a row j such that, $Tight(j, v_i) =$ *true*. If there exists such a row, go to step 9

4. Let $S = \{r_j\}$ be the set of all rows in the PAM such that $\xi(r_j, v_i) =$ *true*. If $S = \Phi$ go to step 8.

5. Let $S' = S$. Remove from S' every r_j that satisfies the following relation: if node v_i is allocated to row $r_j \Longrightarrow \xi(r_j, v_k) =$ *false* for $k = i+1, \ldots, N$. If $S' = \Phi$, then set $S'' = S$, and go to 7.

6. Let $S'' = S'$. Remove from S'' every r_j that satisfies the following relation: if node v_i is allocated to row $r_j \Longrightarrow r_j$ will contain a gap (a contiguous time slot in a row that can not accommodate any node v_k for $k = i+1, \ldots, N$). If $S'' = \Phi$, then set $S'' = S'$.

7. Let row j in S'' be the row which has the minimal number of empty cells. Go to step 9.

8. Create a new row j in the PAM.

9. Allocate node v_i to row j in the PAM.

Thus far, the flexibility assigned to the nodes is due to their existence in non critical circuits. If the delay optimality is not required, then the non circuit nodes will also have scheduling flexibility. In fact, each such node will have a scheduling flexibility of $(T - 1)$, the maximum flexibility possible. This is due to the fact that adding positive delays to the feed-forward part of a DFG may not affect the integrity of the data dependency. To use the scheduling flexibility of the non-circuit nodes, we propose the following clustering technique.

4.3 The Clustering Technique

In this section, we present a procedure to utilize the flexibility of acyclic nodes and that of the relative scheduling of the nodes belonging to disjoint circuits. Before presenting this procedure, we give the following preliminaries. Let γ_j be the sub PAM that contains the schedule of the cluster of nodes which have been collapsed to the node v_j using the Contraction Algorithm. Further, let us represent the scheduling time of such a cluster by the scheduling time of the node v_j. It is noted that each of the acyclic nodes in the given DFG represents a separate cluster. The clustering technique aims at finding a time shift for each of the clusters, in order to reduce the number of processors used. Let us refer to the time shift applied to a cluster relative to its location in the original PAM as its offset. It is clear that not all entries of a submatrix corresponding to a cluster are necessarily occupied

Let us assume that the node set in the resulting acyclic graph G_a is $V_a = \{\gamma_1, \gamma_2, \cdots, \gamma_l\}$, ordered in a way such that the total computational delay of the nodes in γ_i is not less than that of γ_{i+1}, $i = 1, 2, \cdots l - 1$. This procedure involves assigning adequate flexibility to a given cluster such that it can be shifted by any amount for the sake of processor minimization. Due to the periodic nature of the task allocation, a shift greater than $(T - 1)$ will result in an unnecessary increase in the I/O delay. Hence, each cluster is assigned a flexibility of $(T - 1)$. To find the time schedule from the given acyclic DFG, a breadth first search is applied to the acyclic graph to preserve the integrity of the data dependency. The following is an algorithm to accomplish this task.

Clustering Algorithm:

1. Fix the scheduling time of γ_1 to be the scheduling time of v_1, and place γ_1 in the PAM, i.e., assign a shift of 0 to γ_1, where γ_1 is the cluster containing the input node.

2. Let β be the set of nodes $\{v\}$ such that v is a node in G_a and all its immediate precedents in G_a have been mapped to the PAM. If $\beta = \Phi$, halt.

3. For every node $v \in \beta$, set

$$t_s[v] = \max_{w \in pred[v]} (t_s[w] + d_w - n_{(w,v)}T),$$

   where $t_s[v]$ is the scheduling time of node v, $d_u = d_u^c + d_u^s$ the total delay of node w, $n_{(u,v)}$ the ideal delay of the edge (w, v), and $pred[v]$ the set of all immediate precedents of v.

4. Choose a node $v_j \in \beta$ whose corresponding cluster has the maximal total computational delay.

5. Try all the different possible shifts for γ_j from 0 to $(T-1)$, and choose the one that minimizes the number of rows that have already been used in the PAM. If more than one such shift give the same number of rows, use the one that results in a minimum number of occupied time slots in the last of the rows already used, and place γ_j in the PAM using this shift.

6. Let $\beta = \beta - \{r_j\}$.

7. Go to step 2.

It is to be noted that the Clustering Algorithm may not be applicable if the schedule is required to be delay optimal. The technique which is used to guarantee the delay optimality introduces an additional path (delay-optimality path) from the output node to the input node. This will result in each node of the graph to be contained in some circuit, since, by definition, there is a path from the input node to every node in the graph and a path from every node to the output node. As a result, when this graph is reduced to an acyclic graph by a series of contractions, it will result in a single node. The delay flexibility of the nodes, which are not contained in any circuit in the original DFG, is exploited during the conversion of the non-critical circuit containing these nodes. These circuits, as mentioned earlier, result from introducing the path p_{OI}. As to how the flexibility of such nodes is exploited is answered in the following theorem.

Theorem 8 *Let p_v be the longest I/O path passing through an acyclic node v in G, implying that the maximum delay flexibility of v is $L_0 - len[p_v]$. When the delay-optimality path p_{OI} is introduced, the circuit consisting of p_v and p_{OI} will be the circuit of minimum slack time among all the circuits containing v. The slack time itself equals $L_0 - len[p_v]$.*

Proof:

Each I/O path containing v results in a circuit containing v when p_{OI} is introduced. Since p_i is the path with the maximum path length among all such paths, the circuit C_i consisting of p_i and p_{OI} has the maximum path length among all the circuits containing v. However, $ST[C] = -len[C]$, and hence, the first part of the theorem

$$ST[C_i] = -len[C_i] = -(len[p_i] + len]p_{OI}[). \tag{4.2}$$

Using (3.1), the above equation becomes

$$ST[C_i] = L_0 - len[p_i]. \tag{4.3}$$

and hence, the second part of the theorem. ■

The maximum flexibility of a cyclic node is equal to the slack time of the circuit with the lowest slack time among all the circuits containing that node. Thus, by Theorem 8, the flexibility of an acyclic node can be completely utilized even after introducing the delay-optimality path.

Schedule Refining Technique

In general, the choice as to how the shimming delays are distributed affects the final processor schedule. In order to reduce the number of processors, we propose the following approach. Once the PAM is obtained using the Scheduling Algorithm and the Processor Assignment Algorithm, the processor bound is checked. If the number of rows in the PAM is equal to the processor bound, then the PAM is the output; otherwise, the columns whose entries exceed the processor bound are identified. Let the number of entries in such a column be R. Then the $(R - P_0)$ nodes, which had the highest flexibility to start with among the nodes scheduled in that column,

0	1	2	3	4	5	6	7
			b	b		h	g
f	f		e	e			
c	c	c	d	d	a		

Figure 4.1: A time schedule for the second-order IIR filter

are chosen for shifting. The whole scheduling procedure is repeated by increasing by unity, if possible, the shimming delay associated with every such node. At this stage, the remaining slack time is redistributed using the Conversion Algorithm. If there exists a column in which no node had any flexibility originally, then the processor bound is modified to be equal to the number of entries in that column. If this procedure is repeated a pre-specified number of times without achieving the processor bound, then the process is stopped, and the best PAM obtained thus far is the output. This technique of modifying the distribution of the shimming delays to reduce the number of processors will be referred to as the Schedule Refining Technique.

4.4 Examples

Example 1: Second-order IIR filter

For the DFG of the second-order filter shown in Figure 2.7, a time schedule was obtained in Chapter 2 as shown in Figure 2.10. For convenience, this time schedule is presented again in Figure 4.1. Given this time schedule, we are required to obtain a processor assignment with view to processor minimization. The application of the Processor Assignment Algorithm to the given time schedule is detailed in the following steps.

1. The nodes are queued for scheduling in the order $\{d, e, f, a, g, h, c, b\}$ as per step 1 of the Processor Assignment Algorithm. Further, the PAM is initialized to an empty matrix of dimension 4×3, as the processor bound is 4 and the iteration period 3.

2. Node d is located at time 3 in the time schedule, and hence, it will be mapped to start at time 0 in the first row of the PAM with an iteration index of $(n-1)$.

3. Node e is located at time 3 in the time schedule, and hence is mapped to start at time 0 in the second row of the PAM with an iteration index of $(n-1)$. Since upon mapping node d, row 1 is occupied at columns 0 and 1, we have to map e to another row.

4. Similarly, node f will map to row 3 at columns 0 and 1 with an iteration index of n.

5. Node a is located at time 5 in the time schedule, and hence is scheduled at column 2 of the PAM. Since it can be tightly allocated to row 1, it is scheduled in row 1.

6. Similarly, node g should be scheduled at column 1; hence, it should be allocated to row 4, as this column is occupied in all the preceding rows.

7. Node h is located at time 6 in the time schedule, and hence is scheduled at column 0 of the PAM.

8. Node c is located at time 0 in the time schedule with a delay of 2 and a flexibility of 1. Hence, it can be scheduled to start either at column 0 or at

0	1	2
d^{n-1}	d^{n-1}	a^{n-1}
e^{n-1}	e^{n-1}	
f^{n}	f^{n}	
h^{n-2}	g^{n-2}	
b^{n-1}	c^{n}	c^{n}

Figure 4.2: A processor assignment for the second-order IIR filter.

column 1 of the PAM. Since neither of these columns is free in any of the available rows of the PAM generated thus far, a new row has to be created. Node c is then mapped to this row (row 5).

9. Node b is located at time 3 in the time schedule with a unit delay and a flexibility of unity. Hence, it is scheduled in row 5.

The final PAM is shown in Figure 4.2. Obviously, this processor assignment is not processor optimal, since the number of rows is greater than the processor bound by unity. We now apply the Clustering Algorithm to seek a reduction in the number of processors. From the acyclic graph in Figure 2.9, we see that the graph can be divided into five clusters: one for all the circuit nodes, and one for each of the acyclic nodes. Let γ_1 be the submatrix corresponding to the cluster of the circuit nodes a, b, c and d, while γ_2, γ_3, γ_4, and γ_5 are the submatrices corresponding to the acyclic nodes e, f, g and h, respectively. The submatrix γ_1 can be extracted easily from the PAM in Figure 4.2; this submatrix is shown in Figure 4.3 and denoted by PAM_0. The application of the Clustering Algorithm to the given graph is detailed in the following steps.

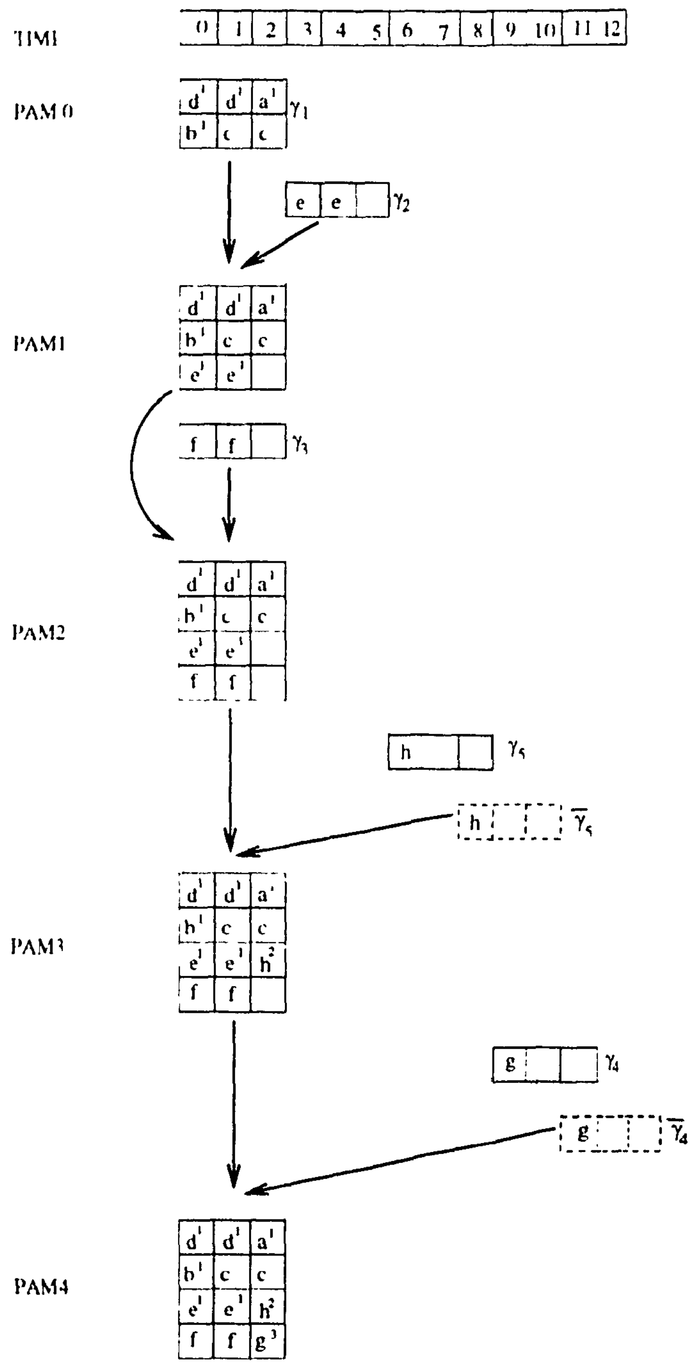


Figure 4.3: The steps of applying the Clustering Algorithm to the DFG of the second-order IIR filter.

0	1	2
d^{n-1}	d^{n-1}	a^{n-1}
b^{n-1}	c^{n}	c^{n}
e^{n-1}	e^{n-1}	h^{n-2}
f^{n}	f^{n}	g^{n-3}

Figure 4.4: An optimal processor assignment for the second-order IIR filter

1. The PAM is initialized to PAM_0.

2. Once γ_1 is scheduled, γ_2 and γ_3 can now be scheduled since they have only γ_1 as their precedent. The submatrix γ_2 can be scheduled without any time shift to produce PAM_1 as shown in Figure 4.3. Similarly, γ_3 is scheduled to produce PAM_2.

3. The only cluster that can now be scheduled is γ_5. With a shift of 2 time units, it can be scheduled to form PAM_3.

4. Finally, γ_4 is scheduled. A shift of this cluster by 2 time units results in PAM_4, as shown in Figure 4.3.

5. $PAM_c = PAM_4$, where PAM_c is the final PAM after the application of the Clustering Algorithm.

The processor assignment for the DFG of the given second-order IIR filter as obtained after applying the Clustering Algorithm is shown in Figure 4.4, and clearly processor optimal.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
										34	20	33	33		
					19				27	24	30	31	31	32	35
28		21	21		23	25	26	26	6	5	5	3	4	2^{-1}	29
17	22	18	18	16	10	8	8	7	9	13	14	14	15	12	11^{-1}

Figure 4.5: A processor assignment for the wave filter (T=16).

-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
			22				19	19	19	19	19	19	20	20	20	20	20	20
													34	34	33	33	33	35
		28		21	21	21	23	25	26	26	26	27	24	30	31	31	32	29
2												6	5	5	3	4		
	11	17		18	18	16	10	8	8	8	7	9	13	14	14	15	12	

Figure 4.6: A time schedule for the wave filter (T=17).

Example 2: A fifth-order wave digital filter

Applying the Processor Assignment Algorithm to the time schedule shown in Figure 2.17 for the wave digital filter of Figure 2.11 results in the PAM shown in Figure 4.5. Obviously this PAM is not processor optimal. As a matter of fact, neither the Clustering Algorithm nor the Schedule Refining Technique may result in a processor optimal PAM for this DFG.

We now repeat the process of processor assignment for the same example with an iteration period of 17 instead of 16. We do this to attempt to reduce the number of processors at the expense of rate optimality. Applying the Scheduling Algorithm detailed in Chapter 2 results in the time schedule shown in Figure 4.6. The shimming delays in this case are redistributed as shown in Table 4.1 as per the Conversion Algorithm. The PAM corresponding to the time schedule obtained by using the Processor Assignment Algorithm is shown in Figure 4.7. It is clear that

Circuit	Slack time prior to contraction	Shimming delay distribution
C_1	1	1 (node 8)
C_2	0	
C_3	2	1 (node 21) 1 (node 26)
C_4	2	1 (node 33) 1 (node 34)
C_5	10	node 19 $\leftarrow$ 5 node 20 $\leftarrow$ 5

Table 4.1: Distribution of shimming delays for the wave filter DFG (T=17).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
											20	34	33	33		
											24	30	31	31	32	35
28		21	21		23	19	26	26	27	6	5	5	3	4	2^{-1}	29
17	22	18	18	16	10	25	8	8	7	9	13	14	14	15	12	11^{-1}

Figure 4.7: A processor assignment for the wave filter (T=17).

this PAM is not processor optimal. The Clustering Algorithm can not be used to reduce the number of processors, as the acyclic DFG corresponding to the wave filter consists of only one node, which implies that there is only one cluster in the whole graph. However, we can apply the Schedule Refinement Technique to the given time schedule so that the number of processors used is reduced.

The Schedule Refining Technique alters the flexibility of the nodes in the columns whose entries exceed the processor bound, such that the number of rows in the PAM is reduced. In the first attempt, we increase the shimming delay associated with each of the nodes 20, 33, and 34 by one time unit. This deprives nodes 21 and 26 from their shimming delays, but assigns instead a shimming delay of unity to each of the nodes 24 and 31 as per the Conversion Algorithm. The resulting processor assignment is still non-optimal in terms of the number of processors used.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
						19			34	33	33	35	31	31	32	29
28		21	21	23	25	26	26	27	6	24	5	5	3	4	2^{-1}	11^{-1}
17	22	18	18	16	10	8	8	7	9	13	30	14	14	15	12	20

Figure 4.8: An optimal processor assignment for the wave filter (T=17).

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	2	2	2	2	3	4	5	5	5	5	5	6
7^{-1}	8^{1}	8^{-1}	8^{-1}	8^{-1}	8^{-1}	9^{-1}	10^{-1}	11^{1}	11^{-1}	11^{-1}	11^{-1}	11^{-1}	12^{1}
						14^{-1}							13
													15^{-1}

Figure 4.9: A processor assignment for the lattice filter DFG.

After the first attempt, nodes 20 and 33 remain in conflict, that is, they can not both be scheduled without exceeding the processor bound with the present distribution of the shimming delays. In the second attempt, we increase the shimming delay of each of the nodes 20 and 33 by unity; unfortunately, it does not result in an optimal assignment either. In the third attempt, the shimming delay of node 20 is increased by unity, while node 32 assigned a unit shimming delay. The latter is done in view of the conflict between nodes 32 and 33, and the fact that node 33 can not be assigned any further shimming delay. The resulting processor assignment is optimal and is shown in Figure 4.8.

Example 3: A fourth-order lattice filter

Given the time schedule for the lattice filter shown in Figure 3.3, the corresponding PAM can be obtained using the Processor Assignment Algorithm, and is shown in Figure 4.9. Obviously, this PAM does not correspond to a processor optimal

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	2	2	2	2	3	4	5	5	5	5	5	6
7^{-1}	8^{-1}	8^{-1}	8^{-1}	8^{-1}	8^{-1}	9^{-1}	10^{-1}	11^{-1}	11^{-1}	11^{-1}	11^{-1}	11^{-1}	13
12^{-2}						14^{-1}							15^{-1}

Figure 4.10: A processor-optimal but not delay-optimal processor assignment for the lattice filter DFG

assignment. The Clustering Algorithm can not be used, since the acyclic graph consists of only one single node. Further, the Schedule Refinement Technique is of no use, since no node has any flexibility.

It is possible to reduce the number of processors used by sacrificing the delay optimality. If that is the case, then the Clustering Algorithm can easily result in a PAM with one less processor. This can be accomplished by assigning to node 12 a shimming delay different from 0, which is a direct consequence of the Conversion Algorithm. This is due to the fact that each and every node of the lattice filter is contained in some critical circuit, except for node 12. The optimal processor assignment is given in Figure 4.10.

4.5 Summary

In this chapter, we have presented an algorithm to map a given time schedule to a processor assignment. It is noted that this step is also a compile time task as we are dealing with fully static scheduling. A compile time processor assignment has the possibility of providing a very high throughput, since the runtime overhead is avoided. The criterion considered in producing a processor assignment is to minimize the number of processors, thus attempting to achieve the processor bound.

To produce a processor assignment (represented by a processor assignment matrix PAM) from a given time schedule, we perform the basic operation of shifting the tasks to fit within a matrix of T columns, T being the iteration period. Using this basic operation and benefiting from scheduling premises, we have established a processor assignment algorithm.

To utilize the flexibility of acyclic nodes, we have proposed a clustering algorithm. In this algorithm, the clusters corresponding to the different supernodes are shifted in time so that the PAM resulting after carrying out these shifts has a minimal number of rows.

We have proposed a schedule refining technique to improve the processor assignment should the Processor Assignment Algorithm fail to achieve the processor bound This technique is a recursive procedure that increases the flexibilities assigned to the nodes that are the cause for the increase in the number of processors required.

The algorithms and the techniques developed in this chapter have been applied to the time schedules for the examples considered in the preceding two chapters.

Chapter 5

Performance Evaluation of the Scheduling Technique

After ensuring the correctness of any algorithm, it is natural to ask about its performance. In studying the performance of an algorithm, an important question is the kind of measure one ought to use. Most researchers are of the opinion that time and space are the best performance measures.

Further, the question of the kind of computer model that we should adopt in comparing the time complexities of different algorithms is still an open question. This, however, has led to the identification of the class of problems solvable in time, bounded by a polynomial in the length of the input, in our case this being the size of the DFG. This class of problems is independent of the computer model that is chosen. The other class of problems includes those which require time that grows exponentially with the size of the input. For these problems, one may not be able to compare different algorithms based on a standard measure. Despite the fact that the scheduling problem is NP-complete, it can be solved using a polynomial time algorithm, and hence, an upper bound on the time-complexity can be found. Using a polynomial-time algorithm for the scheduling problem implies that the optimal solution is not, in general, guaranteed. Hence, a comparison based on some benchmark problems is a necessity.

In this chapter, we first present a systematic derivation of the time complexity of the scheduling technique developed in the last three chapters. In the next section, we will compare our algorithm with some of the existing work, taking into account time complexity and memory requirements, as well as the optimality measures considered for scheduling.

5.1 Time Complexity

Time complexity describes the order of time required to complete a given algorithm in the worst case, the input being the variable parameter. To evaluate the computational complexity of an algorithm, finding upper bounds on the different modules of an algorithm is a common practice. For most algorithms finding an upper bound is usually feasible. However, making this bound tight enough to reflect the actual complexity of the algorithm is more subtle. On the other hand, sometimes it is required to find a lower bound on the complexity of an algorithm. This may be used to compare a given solution with the theoretically optimal one. As a matter of fact, finding a tight lower bound on the complexity of an algorithm is usually a cumbersome task.

In general, optimal scheduling of data flow graphs onto multiprocessors is considered computationally intractable or simply NP-complete [76]. The following theorem will prove the NP-completeness of this scheduling problem by showing that the scheduling search space grows exponentially with the size of the DFG.

Theorem 9 *The search space for scheduling an arbitrary DFG is tightly upper bounded by T^N, where T is the iteration period and N the number of nodes.*

Proof:

It is clear that the search space increases as the flexibilities of the nodes increase. In a graph for which no node has any flexibility, the search space is unity, and there is only one independent schedule. The maximum search space exists when each node of the given graph has infinite flexibility in scheduling. In periodic scheduling, however, a node has a maximum flexibility of $(T - 1)$ time units, that is, it can be scheduled to fire at any of the T consecutive time units. Hence, N nodes can be scheduled in at most T^N different ways. To prove that this bound T^N is a tight one, we need to show the existence of a graph whose nodes are highly flexible in scheduling. A straight forward example of such a graph is an acyclic graph, whose nodes have infinite flexibility, since inserting arbitrary positive delays to its edges is a permissible operation as far as the integrity of the data dependency is concerned. ■

We now derive an upper bound on the time complexity required by the different algorithms needed for static scheduling a data flow graph onto multiprocessors. This time complexity will cover all the algorithms presented in the previous three chapters.

It is recalled that the technique employed in Chapter 3 to ensure delay optimality alters the graph by introducing a feedback path from the output to the input node. Introducing such a path will result in increasing the number of circuits in the graph by the total number of its I/O paths. The time complexity of the existing efficient techniques for finding the circuits of a directed graph is an increasing function of the number of circuits. As finding the circuits of a graph is a crucial step in our scheduling algorithm, introducing the feedback path will, in general, increase the time complexity. This penalty in time complexity should be accounted for. In most DSP applications, however, the delay optimality is not usually an important issue [54]. As a matter of fact, in many cases, ensuring the delay optimality precludes minimizing the number of processors needed, which is a more critical issue. That is, the number of processors can be reduced if the delay optimality is ignored. Furthermore, we believe that this penalty is insignificant in view of the technique described in Chapter 3, and hence, we will not include it in the time complexity analysis.

To find the time complexity of the algorithms presented in this thesis, we have to study the complexity of each of the following tasks:

1. Enumerating all the elementary circuits of a directed graph,
2. contracting a critical circuit to a supernode,
3. converting a non-critical circuit to a critical one,
4. scheduling the resulting acyclic graph,
5. scheduling a circuit,
6. mapping the time schedule to a processor assignment, and
7. implementing the clustering algorithm.

Finding the elementary circuits

For enumerating the elementary circuits, we use the Johnson Algorithm [77] which has a time complexity

$$TC_1 = (N + M)(L + 1), \tag{5.1}$$

where L is the number of elementary circuits, N the number of nodes, and M the number of edges.

Circuit contraction algorithm

To compute the time complexity for contracting a circuit we need to know the number of nodes in that circuit prior to the time it is chosen for contraction. Since the number of nodes of a circuit varies through the different steps of the algorithm, i.e. the circuit contraction steps, we need to find an upper bound on this number. Further, we need an upper bound on the number of edges incident on the nodes of a circuit prior to its contraction, since upon visiting a node in a circuit to be contracted, each edge incident on that node is to be examined. (Recall that an edge is incident on a node if it is either incoming to or outgoing from that node). We now present a systematic way to find these bounds.

Theorem 10 *If C is a circuit in G, and C'' is the corresponding circuit in G'', the graph resulting after contracting a critical circuit C_0 from G, then $|V(C'')| \leq |V(C)| + 2$, where $|S|$ is the cardinality of the set S which is, in this case, equivalent to the number of nodes.*

Proof:

Case 1: If $C \cap C_0 = \Phi$, the proof is evident, and $|V(C'')| = |V(C)|$.

Case 2: $C \cap C_0 \neq \Phi$.

A circuit C such that $C \cap C_0 = \{p_1, p_2, \ldots, p_k\}, k > 1$, where p_i is a maximal path in C_0, and $p_i \cap p_j = \Phi$ for all $i \neq j$, will have no corresponding circuit in G'' as per Corollary 1. Recall that such a circuit will contain at least one hop-path with respect to the circuit C_0. Hence, a circuit C in G will have a corresponding circuit

in G' if and only if its node set is disjoint from that of C_0 (Case 1) or if $C \cup C_0 = p_c$, where p_c is a path consisting of at least a single node.
For each circuit C such that $C \cup C_0 = p_c$, C'' will acquire at most three more nodes: a pseudo node for the edge in C incident into the initial node of p_c, a pseudo node for the edge in C incident out of the terminal node of p_c, and the supernode; and it will loose the nodes belonging to p_c for which $|V(p_c)| \geq 1$; hence the proof. ■

Corollary 3 *Let $C_1, C_2, \ldots, C_k$ be a sequence of circuits such that C_1 is the first circuit chosen for contraction, C_2 the second, C_i the ith, and so on. The number of nodes of a circuit C_i prior to its contraction has an upper bound $|V(C)| + 2(i-1)$.*

Let the maximum number of nodes in a circuit C_i prior to its contraction be $|V(C_i)|_{max}$, where C_i is the ith circuit to be chosen for contraction. As per the above corollary,

$$|V(C_i)|_{max} = |V(C_i)| + 2(i-1) \tag{5.2}$$

Using the fact that $|V(C_i)| \leq N$, N being the number of nodes, (5.2) can be re-expressed as

$$|V(C_i)|_{max} = N + 2(i-1) \tag{5.3}$$

As mentioned earlier, along with finding an upper bound on the number of nodes of a circuit prior to its contraction, we need to know the maximum number of edges incident on its nodes. Recall that in the Contraction Algorithm, each edge incident on the nodes of the circuit to be contracted is altered by changing either its initial or terminal node. Hence, the number of operations required to contract a circuit is also linearly dependent of the number of edges incident on that circuit.

Theorem 11 *Let C be a circuit in G, and C' be its corresponding circuit in G', the graph resulting after contracting circuit C_0. The total number of edges incident on* *all the nodes of a circuit C'' excluding $E(C'')$ may not exceed $M - |E(C'')|$.*

Proof:

If $C \cap C_0 = \Phi$, the proof is obvious. If $C \cap C_0 \neq \Phi$, and C contains a hop-path with respect to C_0, then C will have no corresponding circuit in the graph resulting after contracting the circuit C_0, and hence the proof is also obvious.

Now we have to prove the theorem for the last case where $C \cap C_0 \neq \Phi$, and C contains only one common path with C_0, say p_c. Let the initial node of p_c be v_I and the terminal node v_T. Further, let v_1 and v_2 be the nodes such that $(v_1, v_I), (v_T, v_2) \in E(C)$. As per the Contraction Algorithm, all the edges incident on the nodes of C other than the nodes in p_c, will remain the same and no further edges will become incident on these nodes. The pseudo node which belongs to C'' as node v_I is deleted will become the terminal node of all the edges incident into v_I, and the pseudo node which belongs to C'' as node v_T is deleted will become the initial node of all the edges incident out of v_T. All edges incident on the nodes of p_c other than v_I and v_T will disappear from the edges incident on the nodes of C''. The two edges that connect the supernode with the pseudo nodes created as the nodes v_I and v_T are deleted are essentially circuit nodes in C''; hence the theorem ■

Corollary 4 *The maximum number of edges incident on all the nodes of circuit C excluding $E(C$'s $M - 2$.*

This is due to the fact that a non-trivial circuit has at least two edges. A trivial circuit is a self-circuit that contain one node and one edge. Such a circuit can be ignored for the purpose of scheduling.

For purposes of contraction, each node is visited once, and each of the edges incident on it is examined once; therefore, the worst time complexity for contracting a critical circuit C_i will be given by

$$TC_2 = O(|V(C_i)| + |E(C_i)|), \tag{5.4}$$

where C_i is the ith circuit to be chosen for contraction, and $|V(C_i)|$ and $|E(C_i)|$ are, respectively, the number of its nodes and edges prior to its contraction.

Using (5.3) and the bound in Corollary 4, (5.4) can be rewritten as

$$TC_2 = O(N + 2(i - 1) + M - 2). \tag{5.5}$$

Even if the number of circuits in the DFG G is greater than the number of nodes N, no more than N circuit contractions are required to convert G to an acyclic graph. For the worst case, let us assume that N contractions are needed. Let these circuits be $C_1, C_2, ..., C_N$ such that C_i is contracted before C_j if $i < j$. Summing up the time complexities, each given by (5.5), over all the contracted circuits, the total time complexity for contraction is

$$\begin{aligned} TC_3 &= \sum_{i=1}^{N} O(N + 2(i - 1) + M - 2) \\ &= O(N^2 + N(N - 1) + (M - 2) \cdot N) \\ &= O(N^2 + M \cdot N) \end{aligned} \tag{5.6}$$

Conversion algorithm

In the Conversion Algorithm, each node of the circuit to be converted is visited at most once. The Conversion Algorithm, presented in Chapter 2, stops visiting the nodes of a circuit to be converted when the slack time becomes zero. Hence, the time complexity for the conversion of a non-critical circuit C to a critical one is of the order of the number of its nodes, namely, $|V(C)|$.

Scheduling the acyclic graph

For the scheduling of the acyclic graph resulting after all the circuits have been contracted, we use the critical path method explained in Chapter 2. Let G_a be an acyclic graph with N_a nodes and M_a edges. In the critical path method explained in Chapter 2, each edge is accessed twice: when pushed on the stack and when popped out of it. Further, each node is examined once to be considered for scheduling. Hence, the time complexity of the critical path method is

$$TC_{cpm} = O(N_a + M_a) \tag{5.7}$$

Therefore, to find the time complexity for scheduling the acyclic graph, we need to know the number of its nodes and the number of its edges.

Theorem 12 *The number of nodes in the acyclic graph resulting from contraction all the circuits is bounded by $N_a = 2N - 1$, and the number of edges by $M_a = M + N - 1$.*

Proof:

For every deleted node due to a circuit contraction, at most two pseudo nodes are added. However, for every deleted pseudo node, there will be at most one pseudo node added since a pseudo node is either a join or a fork node. A joint node is node which has only one outgoing edge while a fork node is a nodes which has one incoming edge. Hence, the increase in the number of nodes after all the circuits are contracted, will not exceed $N - 1$ since the acyclic graph should have at least one node; hence the first part of the theorem. Similarly, the second part is established. ■

Substituting the above bounds in (5.7), the time complexity for scheduling the acyclic graph resulting after all the circuits are contracted, can be expressed as

$$\begin{aligned} TC_4 &= O(M_a + N_a) = O(2N - 1 + M + N - 1) \\ &= O(M + N) \end{aligned} \tag{5.8}$$

Scheduling a circuit

It is recalled that a circuit is scheduled by tracing its nodes starting from one of the nodes that has already been scheduled. As this time complexity is linear with the number of circuit nodes, it can be combined with the time complexity for circuit contraction.

Processor assignment

The time complexity for mapping the time schedule to processor assignment is

$$TC_5 = O(N^2 + N \cdot T \cdot P), \tag{5.9}$$

where T is the iteration period and P is the number of processors used. This is in view of the fact that as a node is mapped to the PAM, every other non-mapped node will be examined, and further every node will have a search space of PT.

Clustering

Since each cluster of the given graph is examined at most T times for the purpose of clustering, and the number of clusters is upper-bounded by N, the complexity of the clustering procedure is

$$TC_6 = O(N \cdot T) \tag{5.10}$$

Hence from (5.1), (5.6), (5.8), (5.9), and (5.10), the overall complexity of the algorithms is

$$\begin{aligned} TC &= O((N+M)(L+1)+N^2+M\cdot N+M+N+N^2+N\cdot T\cdot P+N\cdot T) \\ &= O((N+M)(L+1)+N^2+M\cdot N+N\cdot T\cdot P) \qquad (5.11) \end{aligned}$$

For most DSP applications $O(M) = O(N)$, and hence the time complexity becomes

$$TC = O(N\cdot L+N^2+N\cdot T\cdot P). \qquad (5.12)$$

5.2 Comparison with Previous Work

In Section 2.3, we have presented different methods employed to schedule deterministic data flow graphs onto multiprocessors. All these methods other than the rate-optimal ones produce poor throughput, and hence we will not consider them for comparison with our work. Rate-optimal methods can be categorized into two different groups: cyclo static, and fully static methods.

In both these methods, a schedule is determined by the topology of the cyclic graph, and an optimal overlapped execution of successive iterations is attainable. They apply to iterative (recursive) programs and are characterized by a schedule for one iteration which is periodically repeated. Fully static schedules are periodic in time while cyclo-static schedules are periodic in both time and processor space. That is, in a fully static case, the processor assignment completely specifies the processors that are going to execute the different tasks; however, in the cyclo-static case, the processor assignment specifies the tasks that are to be executed at each time unit independent of the physical processors. The strong point of a cyclo static method, which does not exist in a fully static method, is that it can produce a rate-optimal schedule, even when there are operations whose computational delays are larger than the iteration bound. As a matter of fact, cyclo static scheduling is a sort of unfolding if the complete processor schedule is to be determined at compile time, thus the memory requirement is increased. However, if the periodicity in the processor space is to be implemented in real time, then unfolding is unnecessary. In this case, there would be a run-time overhead for task switching over the different processors.

Basically, cyclo-static scheduling is accomplished by fixing both the iteration period and the number of processors, and conducting a depth-first search for cyclo-static solutions. Such a search can not guarantee a schedule, since a solution may not exist when both of the iteration period an the number of processors are fixed. The search space for cyclo static solutions is a huge one, and hence the basic problem with cyclo-static methods is the associated large time complexity. Further, as discussed earlier, this method of scheduling has also the drawback of either requiring an increased memory requirement or the need for run-time overhead, thus demanding more hardware resources.

As a consequence of the above discussion, we will compare our work only with the existing fully-static methods. The simplest method to produce a rate-optimal schedule is the critical path method operated on the cyclic DFG. In this method, the weight assigned to a node is its computational delay and the weight assigned to an edge is the negative of the product of its ideal delay and the iteration bound. Using Bell-Ford algorithm to find the longest distance between the root and every other node of the graph will result in a valid rate-optimal schedule. As the distance is calculated relative to the root, the schedule will also be delay-optimal. The principal drawback of this method is that it does not take processor minimization into consideration, and hence it rarely leads to an optimal processor assignment.

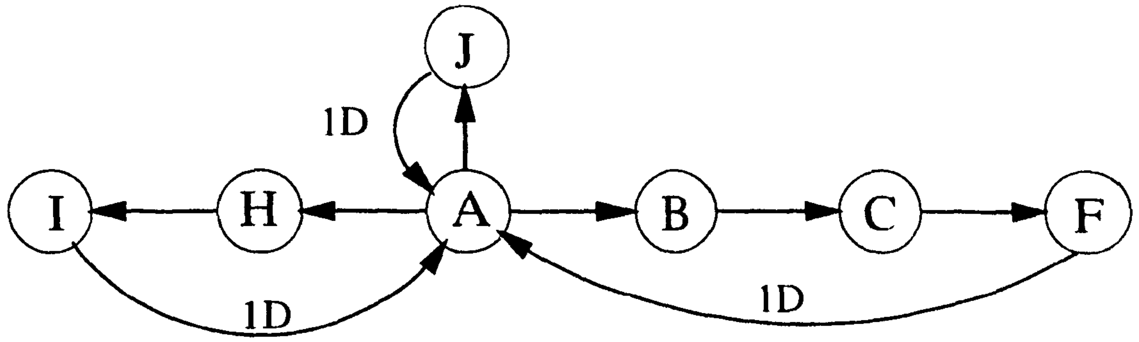


Figure 5.1: A DFG which may not be scheduled processor-optimally by the critical path method

0	1	2	3
A	B	C	F
	H	I	
	J		

Figure 5.2: A processor assignment for the DFG in Figure 5.1 as produced by the critical path method

Consider the simple graph shown in Figure 5.1 in which the computational delay of each node is unity. A is the input node (root) and F is the output node.

The longest distance between node A and the nodes B, C, F, H, I and J is 1, 2, 3, 1, 2, and 1 time unit, respectively. Hence the processor assignment using the critical path method is as shown in Figure 5.2. The resulting processor assignment is not optimal as it uses 3 processors as compared to 2, the processor bound. The processor bound can easily be achieved using any of the existing heuristics. The time schedule and the rate-, delay- and processor-optimal processor assignment for this DFG, using our method, are depicted in Figures 5.3 and 5.4, respectively.

0	1	2	3
A	B	C	F
	H	H	I
	J	J	J

Figure 5.3: A time schedule for the DFG in Figure 5.1 as produced by the Scheduling Algorithm

0	1	2	3
A	B	C	F
	H	J	I

Figure 5.4: An optimal processor assignment for the DFG in Figure 5.1 as produced by the Processor Assignment Algorithm

The maximum spanning tree method detailed in [70] and [54] is based on the scheduling of an equivalent acyclic DFG obtained from the original graph by inserting delays equal to the processing time of the operations, removing the original computational delays and inserting shimming delays when needed. This method does not try to optimize the number of resources leading, in general, to a non-optimal processor assignment.

The optimum unfolding technique presented by Parhi and Messerschmitt [55] unfolds the given graph by an unfolding factor such that each circuit in the resulting graph contains an ideal delay of unity. The strong point of this technique is that it enables rate-optimal scheduling even when some operations have computational delays higher than t'.e iteration bound. The major drawback of this technique is the increase in the size of the programs, since the unfolding factor could be very large, in the general case.

The most efficient technique existing in the literature for producing fully static rate-optimal schedules is the range-chart heuristic. It is efficient in the sense that, in most cases, it is capable of producing an optimal processor assignment, requires no unfolding and has a reasonable polynomial time-complexity. Further, Heemstra de Groot et al. [68] have shown that their range-chart technique outperforms those of others, as shown in Table 5.1.

In view of the above, we believe that it is adequate to compare our technique with the range-chart technique to show how efficient our algorithm is. Considering the issues in the above table, there is no doubt as can be seen from the preceding three chapters that our algorithm is optimal, that is, in terms of throughput, memory requirement, and guarantee of a solution. Since our algorithm and the

Technique	Number of processors	Iteration period	Memory space	Guarantee of solution
Single-iteration	poor	poor	minimal	yes
Direct blocking	efficiency increases with the blocking factor		increases with the blocking factor	yes
Maximum spanning tree	not optimized	optimal	minimal	yes
Search of cyclo static schedules	optimal	optimal	minimal	no
Scheduling range chart	optimized	optimal	minimal	yes

Table 5.1: Comparison of several scheduling algorithms for cyclic DFGs [68].

range-chart technique both perform well with regard to these issues, our comparison between these techniques will be limited to two issues, time complexity, and processor minimization.

The algorithm used for finding the iteration period in [68] requires a time complexity of $O(d \cdot M + d^3 \cdot \log d)$, where d is the number of delay elements and M the number of edges [74]. However d in the worst case is $O(M)$, and hence finding the iteration period has a worst case complexity of $O(M^3 \log M)$. Therefore, in the worst case, the time complexity for the range-chart algorithm is $O(M \cdot N + M^3 \log M + N^3 + N \cdot T \cdot P)$. It is assumed in [68] that $O(M) = O(N)$ and hence the time complexity of the range-chart algorithm is $O(N^3 \log N + N \cdot T \cdot P)$. The time complexity of the technique described in this thesis is $O(N^2 + N \cdot L + N \cdot T \cdot P)$ as given by (5.12), where again it has been assumed that $O(M) = O(N)$. It is clear that the time complexity of our algorithm is the $\max\{O(N^2), O(N \cdot L)\}$. Hence, our algorithm will have less time complexity than the range-chart technique, provided $O(N \cdot L) < O(N^3 \log N)$, which implies $O(L) < O(N^2 \log N)$, a valid assumption.

As far as the processor minimization is concerned our algorithm gives results

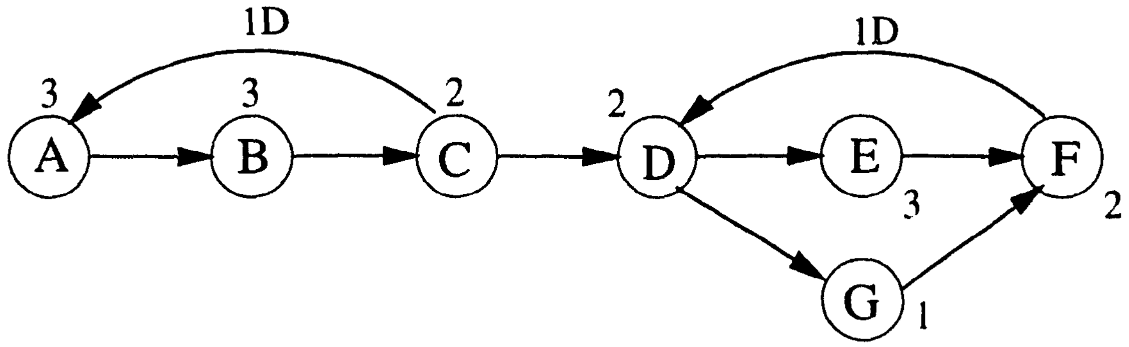


Figure 5.5: A DFG which may not be scheduled processor-optimally by the range-chart technique

similar to those in [68] (optimal solutions). However, for some class of problems, our algorithm does a better job than the range-chart algorithm in terms of processor minimization. To support this claim, we provide the following example. Consider the DFG shown in Figure 5.5, where A is the input node and D is the output node. The circuit $ABCA$ is critical with an iteration bound of 8. The circuits $DEFD$ and $DGFD$ have slack times of 1 and 3, respectively. The range-chart technique will first schedule the nodes A, B and C as they have no flexibility. The nodes D, E, and F have the same flexibility which is unity, but the algorithm will, respectively, choose nodes D, E and F because of the fixed-limit and the lowest index conditions. In a similar manner G 's scheduled. The output equivalent classes for this schedule is shown in Figure 5.6. Obviously this equivalent classes schedule requires three processors. On the other hand, the time schedule resulting from our algorithm is shown in Figure 5.7. The corresponding processor schedule as produced by the Processor Assignment Algorithm is shown in Figure 5.8. It is seen that our algorithm requires two processors (the processor bound) as compared to three required by the range-chart technique.

0	1	2	3	4	5	6	7
A	A	A	B	B	B	C	C
D^{-1}	D^{-1}	E^{-1}	E^{-1}	E^{-1}	F^{-1}	F^{-1}	
		G^{-1}					

Figure 5.6: The equivalent classes produced by the range-chart technique

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	A	A	B	B	B	C	C	D	D	E	E	E	E	F	F
										G	G	G	G		

Figure 5.7: The time schedule as produced by the Scheduling Algorithm

0	1	2	3	4	5	6	7
A	A	A	B	B	B	C	C
D^{-1}	D^{-1}	E^{-1}	E^{-1}	E^{-1}	G^{-1}	F^{-1}	F^{-1}

Figure 5.8: The processor schedule as produced by the Processor Assignment Algorithm

5.3 Summary

To evaluate the performance of our algorithm, we have first derived the time complexity needed for the computation of the different modules of the algorithm.. Then, we have compared our algorithm with the existing work in terms of the time complexity as well as the efficiency of the attainable processor assignment, taking into consideration the different optimality criteria. In order to find the time complexity of the different algorithms developed in this thesis, we have derived upper bounds on some of the parameters of the DFG, as the Contraction Algorithm is recursively applied.

In comparison with the previous work, we have shown through example that our scheduling technique can obtain an optimal processor assignment where other algorithms have failed. Further, the time complexity for our algorithm is shown to be lower than that of the range-chart technique (the best technique thus far) for a large class of applications.

Chapter 6

Conclusion

The problem considered in this thesis has been that of achieving an efficient processor assignment for a digital signal processing application program represented by a data flow graph model. The nodes of the given DFG represent operations and the edges represent the flow of operands. The schedule sought in this thesis is a static schedule, that is, it is completely specified at compile time. For an operation, both the processor that is going to execute it and the time at which it is going to be executed are specified at the compile time. To be able to produce such a schedule, the given program should have a predictable behavior, that is, it may not have data-dependent decision making operations.

The efficiency of a processor assignment is represented by three optimality criteria, namely, rate, delay, and processor optimality. In view of the optimality criteria, the scheduling problem can be thought of in two different ways: (1) Given the iteration period $T \geq T_0$ (T_0 being the iteration period bound), it is required to obtain a processor assignment with a minimum number of processors. (2) Given the number of processors, it is required to find a processor assignment to minimize the iteration period. This classification is required due to the fact that it is not always possible to achieve both the rate and processor optimality at the same time.

In this thesis, the scheduling problem has been divided into two separate tasks, namely, the time scheduling and the processor assignment. A time schedule is a schedule that shows the relative scheduling times of the different operations of the given algorithm without consideration to processor assignment. As a matter of fact, for the time scheduling purposes, the number of processors is assumed to be infinite. A heuristic is needed to map the time schedule thus obtained to a processor assignment with a view to minimizing the number of processors required.

We have proved that the rate-optimal time scheduling problem can be obtained analytically by a cyclic to acyclic transformation through a sequence of critical-circuit contractions. Unlike the usual cyclic to acyclic transformations discussed in the literature, the proposed method reduces the given graph without affecting the iterative nature of the graph. That is, the resulting acyclic graph contains all the information needed to schedule the acyclic graph rate-optimally. To accomplish this transformation, we have established the result that a critical graph can be contracted to a single node without altering the integrity of the data dependency. Further, an algorithm for contracting a critical circuit to one of its nodes has been given. For non-critical circuits, we have provided a simple algorithm of converting them into critical ones, so that they can be contracted. To schedule the resulting acyclic graph, we have used the critical path method. The circuits are then scheduled in an order reverse of that of their contraction. The steps which are needed to produce a time schedule for a DFG have been formalized in a scheduling algorithm. The resulting time schedule contains the firing time as well as the scheduling flexibility for each task in the given algorithm.

To incorporate delay-optimality in the time schedule resulting from the proposed scheduling algorithm, a technique to transform the given graph to a delay-optimal graph has been proposed. To accomplish this transformation, we have first proved that it is always possible to achieve both the rate and delay optimality at the same time. Then, we have proposed a simple (in terms of implementation and complexity) scheme to achieve the delay optimality. The idea behind this technique is that if the critical I/O path is a subset of a critical circuit, then a rate-optimal schedule is also delay optimal.

The mapping of the time schedule to a processor assignment has been performed with a view to minimizing the number of processors in an attempt to achieve the processor bound. The processor assignment has been represented by a $P \times T$ matrix, known as the processor assignment matrix, PAM, where P is the number of processors needed and T the iteration period. The basic operation to map a time schedule to a PAM is to shift the tasks to fit within a matrix of T columns. Using this basic operation and benefiting from some scheduling observations, we have established a processor assignment algorithm.

To reduce the number of processors required, which might be at the expense of delay-optimality, we have proposed an algorithm to exploit the flexibility of the acyclic nodes. In this algorithm, called the Clustering Algorithm, the DFG is divided into clusters and the PAMs for different clusters are shuffled in such a way that the number of processors is minimized. For further reduction in the number of processors used, a schedule refining technique has been presented. This technique is used when the Processor Assignment Algorithm fails to achieve the processor bound. The idea of this technique is to recursively increase the flexibilities assigned to the nodes which cause an increase in the number of processors needed until the processor optimality is achieved.

Finally, the performance of the proposed technique has been evaluated by finding an upper bound on its time complexity and comparing its performance with the existing techniques. This comparison is based on both the time complexity and the efficiency of the processor assignment in terms of different optimality criteria.

Unlike most heuristics developed to handle the scheduling problem, the scheduling technique proposed in the thesis guarantees a solution, as its correctness has been established analytically. It is known that the most efficient algorithm, presently available to deal with the scheduling problem, is the range-chart heuristic. The algorithm that has been proposed in this thesis outperforms the range-chart technique in two aspects: (i) it can produce a processor optimal schedule for some problems where the range chart fails, and (ii) for most DSP applications (when the number of circuits in the given DFG is less than $N^2 \log N$, N being the number of nodes in the DFG), the time complexity of our algorithm is of a lower order than that of the range-chart technique.

In this thesis, the multiprocessor system considered for the scheduling problem has been assumed to be a homogeneous system in which all processors are identical. For most DSP applications, the basic operations are addition and multiplication. For such applications, a processor can be as simple as an adder or a multiplier. It is well known in VLSI technology that a multiplier is much more expensive than an adder, and hence, it is more efficient to dedicate the multiplier units to the multiplication operations and the adder units to the addition operations than using processors combining both the units for all operations. This suggests investigating the problem of scheduling on heterogeneous computer systems in which processors are not identical. For example, in a system where there are two types of processors, with and without hardwired multipliers, multiplication may only be assigned to processors of the first type, while additions can be assigned to either.

In this thesis, each processor has been assumed to be capable of uniformly communicating with all other processors in the system. The topology of such an architecture can be represented by a complete graph in which there is an edge between any two nodes. Further research need to be conducted to deal with the problem of scheduling on a pre-defined architecture whose topological graph is not complete. With this constraint, it is no longer possible to neglect the communication delays associated with the edges of the DFG, since these delays are not fixed and may not be determined without the knowledge of the processor assignment. As a matter of fact, in this case, the communication delay associated with an edge is dependent as to where the operations associated with the end nodes of that edge are scheduled.

An algorithm has been proposed to convert a non-critical circuit to a critical one. This algorithm does not guarantee a shimming delay distribution that leads to an optimal processor assignment. Although with the Schedule Refining Technique presented in Chapter 4, the distribution of the shimming delays is altered so that the processor optimality may be achieved, further research is still needed to find an efficient heuristic to carry out this distribution.

Finally, future research should be directed to the development of a theory for static scheduling of large-grain data flow graphs on distributed systems in which the communication delay plays a major rule in the processor assignment. In this problem, other issues that need to be addressed are the topology of the interconnection network, the communication protocol, and the memory management.

REFERENCES

[1] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," *Communication ACM*, vol. 20, pp. 263 271, April 1977.

[2] O. G. Johnson, "Three-dimensional wave computations on vector computers," *Proceedings of the IEEE*, vol. 72, January 1984.

[3] H. M. Ahmed, J. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," *IEEE Computer*, vol. 15, January 1982.

[4] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI processor arrays," in *Mead and Conway, Introduction to VLSI systems*, Wesley Publishing Co, October 1980.

[5] S. Y. Kung and Y. H. Hu, "A highly concurrent algorithm and pipelined architecture for solving toeplitz systems," *IEEE Transactions on ASSP*, vol. ASSP 31, February 1983.

[6] K. Hwang, *Advanced Computer Architecture with Parallel Programming*. Mc-Graw Hill, New York, 1993.

[7] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, November 1980.

[8] I. Watson and J. Gurd, "A practical data flow computer," *Computer*, vol. 15, no. 2, pp. 51-57, 1982.

[9] J. B. Dennis, J. B. Fosseen, and J. Linderman, "Data flow schemes," in *Theoretical Programming, Springer-Verlag, Berlin*, pp. 187 216, 1972.

[10] L. Bic, "Execution of logic programs on a dataflow architecture," in *Proc. The 11th Annual International Symposium on Computer Architecture*, pp. 290 296, 1984.

[11] W. B. Ackerman, "Data flow languages," *Computer*, vol. 15, no. 2, pp. 15 25, 1982.

[12] A. L. Davis and R. M. Keller, "Data flow program graphs," *Computer*, vol. 15, no. 2. pp. 26-41, 1982.

[13] K. Gostelow and R. Thomas, "Performance of a simulated dataflow computer," *IEEE Trans. on Computers*, vol. C-29, no. 10, pp. 905-919, 1980.

[14] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proc. 2nd Annual Symposium on Computer Architecture, New York*, pp. 126-132, 1975.

[15] D. Culler and G. Papadopoulos, "The explicit token store," *Journal of Parallel and Distributed Computing*, pp. 289-308, 1990.

[16] L. Verdoscia and R. Vaccaro, "ALFA: A static data flow architecture," in *Proc. IEEE 4th Symposium on the frontiers of passively parallel computation*, pp. 318-325, 1992.

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

[18] K. P. Tan, T. S. Chua, and P. T. Lee, "AUTO-DFD: an intelligent data flow processor," *The Computer Journal*, vol. 32, no. 3, pp. 194-201, 1989.

[19] D. Dolev and M. Warmuth, "Scheduling flat graphs," *SIAM Journal on Computing*, vol. 14, pp. 638-657, August 1985.

[20] W. Zhao and J. A. Stankovic, "Preemptive scheduling under time and resource constraints," *IEEE Trans. on Computers*, vol. C-36, no. 8, pp. 949-960, 1987.

[21] Arvind and K. P. Gostelow, "The U-interpreter," *Computer*, vol. 15, no. 2, pp. 42-49, 1982.

[22] S. Komori, K. Shima, and S. Miyata, "The data-driven microprocessor," *IEEE Micro*, vol. 9, pp. 45-59, June 1989.

[23] E. J. Lerner, "Data-flow architectures," *IEEE Spectrum*, vol. 21, pp. 57-62, April 1984.

[24] M. Al-Mouhamed, "Analysis of macro-dataflow dynamic scheduling on nonuniform memory access architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 875-888, 1993.

[25] G. A. Uvieghara, W. W. Hwu, and Y. Nakagome, "An experimental single-chip data flow CPU," *IEEE Journal of Solid-State Circuits*, vol. 27, pp. 17-28, Jan. 1992.

[26] M. Sowa, "A method for speeding up serial processing in dataflow computers by means of a program counter," *The Computer Journal*, vol. 30, pp. 289-294, 1987.

[27] K. Kavi, B. Buckles, and N. Bhat, "A formal definition of data flow graph models," *IEEE Trans. on Computers*, vol. C-35, no. 11, pp. 940-948, 1986.

[28] R. G. Babb II, "Parallel processing with large grain data flow techniques," *Computer*, pp. 55-61, July 1984.

[29] I. Waston and J. Gurd, "A practical data flow computer," *Computer*, vol. 15, no. 2, pp. 51-57, 1982.

[30] J. Rumbaugh, "A data flow multiprocessor," *IEEE Trans. on Computers*, vol. C-26, no. 2, pp. 138-146, 1977.

[31] E. A. Lee, "Consistency in dataflow graphs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 2, pp. 223-235, 1991.

[32] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, vol. C-36, no. 1, pp. 24-35, 1987.

[33] E. A. Lee, *A coupled hardware and software architecture for programmable digital signal processors*. Ph.D dissertation, Univ. California, Berkeley, 1986.

[34] E. A. Lee and J. C. Bier, "Architectures for statically scheduled dataflow," *Journal of Parallel and Distributed Computing*, pp. 333-348, 1990.

[35] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235-1245, 1987.

[36] I. P. Radivojevic and J. Herath, "Executing DSP applications in a fine-grained dataflow environment," *IEEE Transactions on Software Engineering*, vol. 17, no. 10 pp. 1028-1041, 1991.

[37] S. H. Lee and T. P. Barnwell, "A topological sorting and loop cleansing algorithm for a constrained MIMD compiler of shift-invariant flow graphs," *ICASSP*, pp. 2927-2930, 1986.

[38] Y. M. Chong, "Data flow chip optimizes image processing," *Computer Design*, vol. 23, pp. 97-98, October 1984.

[39] J. Gaudiot, "Data-driven multicomputers in digital signal processing," *Proceeding of the IEEE*, no. 9, pp. 1220-1233, 1987.

[40] N. Halbwachs, F. Langnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE," *IEEE Transactions on Software Engineering*, vol. 18, pp. 785–793, Sept. 1992.

[41] P. D. Hoang and J. M. Rabaey, "Scheduling of DSP programs onto multiprocessors for maximum throughput," *IEEE Transactions on Signal Processing*, vol. 41, no. 6, pp. 2225–2235, 1993.

[42] K. Thulasiraman and M. N. S. Swamy, *Graphs: Theory and Algorithms*. New York: John Wiley, 1992.

[43] P. R. Gelabert and I. T. P. Barnewell, "Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs," *IEEE Trans. on Signal Processing*, vol. 41, no. 2, pp. 858–888, 1993.

[44] A. Shatnawi, M. O. Ahmad, and M. N. S. Swamy, "Rate-optimal static scheduling of dsp data flow graphs onto multiprocessors using circuit contraction," in *Proc. IEEE International Symposium on Circuits and Systems*, pp. 1360–1363, May 1995.

[45] A. Shatnawi, M. O. Ahmad, and M. N. S. Swamy, "Rate-optimal static scheduling for dsp data flow graphs onto multiprocessors," in *Proc. IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pp. 197–200, May 1995.

[46] A. Shatnawi, M. O. Ahmad, and M. N. S. Swamy, "Rate- and delay-optimal compile-time scheduling of data flow graphs of dsp programs onto multiprocessors," submitted for publication to *IEEE Trans. on Circuits and Systems - I*.

[47] S. Ha and E. A. Lee, "Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration," *IEEE Trans. on Computers*, vol. 40, no. 11, pp. 1225–1237, 1991.

[48] D. A. Schwartz and I. T. P. Barnwell, "Cyclo-static multiprocessor scheduling for the optimal realization of shift-invariant flow graphs," in *Proc. IEEE 1985 International Conference on Acoustics, Speech, Signal Processing*, pp. 1384–1387, 1985.

[49] D. A. Schwartz and T. P. Barnwell III, "Cyclo-static solutions: Optimal multiprocessor realizations of recursive algorithms," in *VLSI Signal Processing II*, ch. 11, IEEE Press, 1986.

[50] D. A. Schwartz, T. P. Barnwell III, and C. J. M. Hodges, "The optimal synchronous cyclo-static array: A multiprocessor supercomputer for digital signal processing," in *Proc. IEEE 1986 International Conference on Acoustics, Speech, and Signal Processing*, pp. 2891–2894, 1986.

[51] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, California, 1990.

[52] M. J. Gonzalez, "Deterministic processor scheduling," *Computing Surveys*, vol. 9, pp. 173–204, September 1977.

[53] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, CA: Freeman, 1979.

[54] M. Renfors and Y. Neuvo, "The maximum sampling rate of digital filters under hardware speed constraints," *IEEE Trans. on Circuits and Systems*, vol. CAS-28, no. 3, pp. 196–202, 1981.

[55] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Trans. on Computers*, vol. 40, no. 2, pp. 178–195, 1991.

[56] D. J. Wang and Y. H. Hu, "Rate optimal scheduling of recursive DSP algorithms by unfolding," *IEEE Trans. on Circuits and Systems-I*, vol. 41, no. 10, pp. 672-675, 1994.

[57] K. R. Baker, *Introduction to Sequencing and Scheduling.* New York: Wiley, 1974.

[58] J. Blazewicz, "Selected topics in scheduling theory," in *Surveys in Combinatorial Optimization*, pp. 1-59, P. L. Hammer, Ed. Amsterdam: Holland, 1987.

[59] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Trans. on Computers*, vol. C-24, pp. 1235-1238, December 1975.

[60] R. E. Crochiere and A. V. Oppenheim, "Analysis of linear digital networks," *Proc. IEEE*, vol. 63, pp. 581–595, April 1975.

[61] T. P. Barnwell, S. Gaglio, and R. M. Price, "A multi-microprocessor architecture for digital signal processing," in *Proc. 1978 Int. Conference on Parallel Processing*, pp. 115–121, August 1978.

[62] J. P. Brafman, J. Szczupak, and S. K. Mitra, "An approach to the implementation of digital filters using microprocessors," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-26, pp. 442–446, October 1978.

[63] R. Nouta and O. Simula, "On multiprocessor implementations of digital signal processing algorithms," in *Proc. IEEE International Symposium on Circuits and Systems*, pp. 1133–1137, April 1980.

[64] O. Simula, "On time-shiftedly operating multiprocessor realizations of digital signal processing algorithms," in *Proc. Fourth International Symposium on Network Theory*, pp. 85–90, September 1979.

[65] P. Evipidou and J. Gaudiot, "Block scheduling of iterative algorithms and graph-level priority scheduling in a simulated data-flow multiprocessor," *IEEE. Trans. on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 398–413, 1993.

[66] S. M. Heemstra de Groot and O. E. Herrmann, "Evaluation of some multiprocessor scheduling techniques of atomic operations for recursive DSP graphs," in *Proc. European Conference on Circuit Theory and Design*, pp. 400–404, September 1989.

[67] C. S. Burrus, "Block realization of digital filters," *IEEE Trans. Audio Electroacoust.*, vol. AV-20, pp. 230–235, October 1972.

[68] S. M. Heemstra de Groot, S. H. Gerez, and O. E. Herrmann, "Range-chart-guided iterative data-flow graph scheduling," *IEEE Trans on Circuits and Systems-I*, vol. 39, no. 5, pp. 351–364, 1992.

[69] K. K. Parhi and D. G. Messerschmitt, "Rate-optimal fully-static multiprocessor scheduling of data-flow signal processing programs," in *Proc. 1989 IEEE Int. Symp. Circuits and Systems*, pp. 1923-1928, 1989.

[70] M. Renfors and Y. Neuvo, "Fast multiprocessor realizations of digital filters," in *Proc. International Conference on Acoustics, Speech and Signal Processing*, pp. 916 919, 1980.

[71] S. M. Heemstra de Groot and O. E. Herrmann, "Rate-optimal scheduling of recursive DSP algorithms based on the scheduling range chart," in *Proc. IEEE Int. Symp. Circuits and Systems*, pp. 1805-1808, May 1990.

[72] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Proc. Third Caltech Conference on VLSI*, pp. 87–116, 1983.

[73] M. Granski, I. Koren, and G. Silberman, "The effect of operation scheduling on the performance of a data flow computer," *IEEE Trans. on Computers*, vol. C-36, no. 9, pp. 1019-1029., 1987.

[74] S. H. Gerez, S. M. Heemstra de Groot, and O. E. Herrmann, "A polynomial-time algorithm for the computation of the iteration-period bound in recursive data-flow graphs," *IEEE Trans. on Circuits and Systems-I*, vol. 39, pp. 49–52, January 1992.

[75] D. Y. Chao and D. Wang, "Iteration bounds of single-rate data flow graphs for concurrent processing," *IEEE Transactions on Circuits and Systems-I: Fundamental Theory and Applications*, vol. 40, no. 9, pp. 629-634, 1993.

[76] J. K. Lenstra and A. H. G. Rinnooy Kan, "Complexity of scheduling under precedence constraints," *Operation Res.*, vol. 26, pp. 22-35, January 1978.

[77] D. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM J. comput.*, vol. 4, pp. 77-84, March 1975.