IMPLEMENTATION OF RETIMING ALGORITHMS IN MATLAB

Dissertation submitted in partial fulfillment of the requirements
for the award of the degree of

Master of Technology

In

VLSI Design

Submitted by

Mehak Aggarwal

Roll. No. 601261017

Under the supervision of

Mrs. Manu Bansal

Assistant Professor, ECED

Department of Electronics & Communication Engineering

Thapar University, Patiala

INDIA

July, 2014

CERTIFICATE

I hereby declare that the work which is being presented in the dissertation entitled, **"Implementation of retiming algorithms in MATLAB"** in partial fulfillment of the requirement for the award of degree of Master of Technology (VLSI Design) at the department of Electronics and Communication Engineering, Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of Mrs. Manu Bansal, Assistant Professor, ECED.

The matter presented in this dissertation has not been submitted in any other University/Institute for the award of any other degree.

Date: 17 July 2017

Mehak Aggarwal

Mehak Aggarwal

Roll.No.601261017

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

16/7/17

Mrs. Manu Bansal
Assistant Professor
ECED,Thapar University

Counter signed by:

Dr. Sanjay Sharma
Professor & Head
ECED, Thapar University
Patiala-147004

Dr. S. K. Mahopatra
Dean of Academic Affa
Thapar University
Patiala-147004

ACKNOWLEDGEMENT

This thesis is completed with prayers of many and love of my family and friends. However, there are a few people that I would like to specially acknowledge and extend my heartfelt gratitude to, who have made the completion of this thesis possible. With the biggest contribution to this report, I would like to thank **Ms. Manu Bansal** who has given me her full support in guiding me with stimulating suggestions and encouragement to go ahead in all the time of the thesis..

I am also thankful to **Dr. Sanjay Sharma**, Professor and Head, Electronics and Communication Engineering Department, for providing us with the adequate infrastructure for carrying out the work.

I am also thankful to **Dr. Kulbir Singh**, P.G Coordinator, Electronics and Communication Engineering department, for the motivation and inspiration that triggered me for the work.

Lastly, I would also like to thank my parents for their years of unyielding love and encouragement. They have always wanted the best for me and I admire their determination and sacrifice.

MEHAK AGGARWAL

ABSTRACT

Data driven property and real time requirements are the two significant features of digital signal processing systems. Dataflow models of computation are widely used to represent digital signal processing applications. DSP algorithms are often repetitive. Execution of all the computations for the required number of times is called iteration. The iteration period of synchronous dataflow graph is the least time required for executing one iteration. Retiming is a graph transformation technique that only changes the delay distribution of the graph, while it has no effect on its functionality. Retiming is used to optimize the synchronous dataflow graphs by reducing the iteration period.

Here, the three algorithms to find the retiming vectors of synchronous dataflow graphs have been implemented and their execution times found. These algorithms are – byHSDF, ZHU10, and sdfFEAS. ByHSDF is the oldest method in which the synchronous dataflow graph is first converted to a homogeneous dataflow graph and then the retiming vector is found. ZHU10 doesn't require the conversion to a homogenous dataflow graph to find the retiming vector. sdfFEAS is the latest method. This method doesn't use backtracking as is employed in the previous method. Instead it employs the concept of critical walks. These algorithms are implemented in MATLAB and their execution times are compared and it is seen which algorithm gives the best results.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ABBREVATION	MEANING
DSP	Digital Signal Processing
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
SNR	Signal to Noise Ratio
SDFG	Synchronous Data Flow Graph
HSDFG	Homogeneous Data Flow Graph

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Digital signal processing (DSP) is the mathematical manipulation of an information signal to modify or improve it in some way. [1] It is characterized by the representation of discrete time, discrete frequency, or other discrete domain signals by a sequence of numbers or symbols and the processing of these signals. Digital signal processing and analog signal processing are subfields of signal processing.

Two important features that distinguish DSP from other general purpose computations are the real time throughput requirement and the data driven property. The hardware should be designed to meet the tight throughput constraint of the real time processing where new input samples need to be processed as they are received periodically from the signal source.

The goal of DSP is usually to measure, filter and/or compress continuous real-world analog signals. The first step is usually to convert the signal from an analog to a digital form, by sampling and then digitizing it using an analog-to-digital converter (ADC), which turns the analog signal into a stream of numbers. However, often, the required output signal is another analog output signal, which requires a digital-to-analog converter (DAC). Even if this process is more complex than analog processing and has a discrete value range, the application of computational power to digital signal processing allows for many advantages over analog processing in many applications, such as error detection and correction in transmission as well as data compression.

1.1 Key Advantages

1. Digital signals are more robust than analog signals with respect to temperature and process variations.
2. The accuracy in digital representations can be controlled better by changing the word length of the signal.
3. Furthermore, DSP techniques can cancel the noise and interference while amplifying the signal. In contrast, both signal and noise are amplified in analog signal processing.

4. Digital signals can be stored and recovered, transmitted and received, processed and manipulated, all virtually without error.
5. Many complex systems are realized digitally with high precision, high signal to noise ratio (SNR), repeatability and flexibility.

1.2 Applications

DSP includes subfields like: audio and speech signal processing, sonar and radar signal processing, sensor array processing, spectral estimation, statistical signal processing, digital image processing, signal processing for communications, control of systems, biomedical signal processing, seismic data processing, etc.

TABLE 1 Applications of digital signal processing

DSP Algorithms	**System applications**
Speech coding and Decoding	Digital cellular phones, personal communication systems, digital cordless phones, Multimedia computers, secure communications
Speech encryption and decryption	Digital cellular phones, personal communication systems, digital cordless phones, secure communications
Speech recognition	Advanced user interfaces, multimedia workstations, robotic and automotive applications , digital cellular phones, personal communication systems, digital cordless phones
Speech synthesis	Multimedia PC's, advanced user interfaces, robotics
Modem algorithms	Digital cellular phones, personal communication systems, digital cordless phones, digital audio broadcast, multimedia computers, wireless computing, navigation, data/facsimile modems, secure communications
Noise cancellation	Professional audio, advanced vehicular audio
Audio equalization	Consumer audio, professional audio, advanced vehicular audio

Image compression and decompression	Digital cameras, digital video, multimedia computers, consumer video
Echo cancellation	Speakerphones, modems, telephone switches
Beam forming	Navigation, radar/sonar, signal intelligence

1.3 DSP domains

In DSP, engineers usually study digital signals in one of the following domains: time domain (one-dimensional signals), spatial domain (multidimensional signals), frequency domain, and wavelet domains. They choose the domain in which to process a signal by making an informed guess (or by trying different possibilities) as to which domain best represents the essential characteristics of the signal. A sequence of samples from a measuring device produces a time or spatial domain representation, whereas a discrete Fourier transform produces the frequency domain information that is the frequency spectrum. Autocorrelation is defined as the cross-correlation of the signal with itself over varying intervals of time or space.

1.3.1 Time domain

The most common processing approach in the time or space domain is enhancement of the input signal through a method called filtering. Digital filtering generally consists of some linear transformation of a number of surrounding samples around the current sample of the input or output signal. There are various ways to characterize filters like linear, causal, time invariant, stable, finite impulse response. A filter can be represented by a block diagram, which can then be used to derive a sample processing algorithm to implement the filter with hardware instructions. A filter may also be described as a difference equation, a collection of zeroes and poles or, if it is an FIR filter, an impulse response or step response.

1.3.2. Frequency domain

Signals are converted from time or space domain to the frequency domain usually through the Fourier transform. The Fourier transform converts the signal information to

a magnitude and phase component of each frequency. Often the Fourier transform is converted to the power spectrum, which is the magnitude of each frequency component squared. The most common purpose for analysis of signals in the frequency domain is analysis of signal properties. The engineer can study the spectrum to determine which frequencies are present in the input signal and which are missing. In addition to frequency information, phase information is often needed. This can be obtained from the Fourier transform. With some applications, how the phase varies with frequency can be a significant consideration.

1.4 Implementation

Depending on the requirements of the application, digital signal processing tasks can be implemented on general purpose computers (e.g. supercomputers, mainframe computers, or personal computers) or with embedded processors that may or may not include specialized microprocessors called digital signal processors.

Often when the processing requirement is not real-time, processing is economically done with an existing general-purpose computer and the signal data (either input or output) exists in data files. This is essentially no different than any other data processing, except DSP mathematical techniques (such as the FFT) are used, and the sampled data is usually assumed to be uniformly sampled in time or space. For example: processing digital photographs with software such as Photoshop.

However, when the application requirement is real-time, DSP is often implemented using specialized microprocessors such as the DSP56000, the TMS320, or the SHARC. These often process data using fixed-point arithmetic, though some more powerful versions use floating point arithmetic. For faster applications FPGA's might be used. Beginning in 2007, multicore implementations of DSPs have started to emerge from companies including FreeScale and Stream Processors, Inc. For faster applications with vast usage, ASICs might be designed specifically. For slow applications, a traditional slower processor such as a microcontroller may be adequate. Also a growing number of DSP applications are now being implemented on Embedded Systems using powerful PCs with a Multi-core processor.

1.5 Meeting the real time requirement

The dataflow models of computation are widely used to represent architecture of DSP applications. One of the most useful dataflow models used to design multi-rate DSP algorithms is the synchronous dataflow graph (SDFG), also called multi-rate dataflow graph. Fig.1, for example, is an SDFG modeling a simplified spectrum analyzer [3].

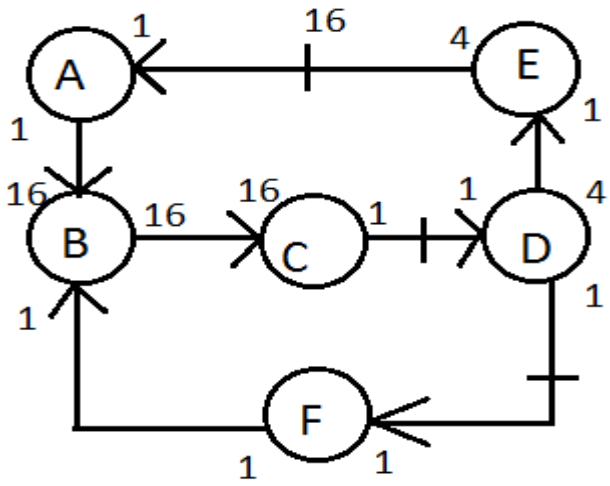

Fig 1. Example of a Synchronous data flow graph

DSP algorithms are often iterative. Execution of all the computations as required times is referred to as an iteration. The iteration period is the time required for execution of one iteration of the algorithm [3]. The less the iteration period of an algorithm is, the higher its throughput is; and performing with high enough throughput is usually a key real-time requirement of a DSP algorithm. It is therefore an important problem to decrease the iteration period to meet the real-time requirement of DSP applications. For dataflow graph the iteration period is limited by its topology and the computation time of nodes. Many graph transformation techniques have been used for solving this problem. Among them, retiming is notable for its simplicity and efficiency. Retiming is introduced originally applied to the homogeneous synchronous dataflow graph (HSDFG), which is a special case of the SDFG, to optimize application's performance by redistributing delays without changing its functionality.

Retiming is a transformation technique used to change the location of delay elements in a circuit without affecting the input/output characteristics of the circuit. Retiming has many applications in synchronous circuit design. These include reducing the clock period

of the circuit, reducing the power consumption of the circuit and logic synthesis. Retiming can be used to increase the clock rate of a circuit by reducing the computation time of the critical path.

Retiming has also been extended to SDFG's. different from the HSDFG, however in an iteration of an SDFG nodes maybe executed in different times in the SDFG in the above figure, for example, node A running 16 times per iteration but node B once. This disables many useful results derived for HSDFG's and complicates the analysis of retiming properties of SDFG's. When retiming is used to reduce the iteration period the method, traditionally, is to transform an SDFG to its equivalent HSDFG and then retime the HSDFG. However converting SDFG to HSDFG increases the problem space hugely.

CHAPTER 2

THE DATA FLOW PARADIGM

In data flow, a program is divided into pieces (nodes or blocks) which can execute (fire) whenever input data are available. An algorithm is described as a dataflow graph, a directed graph where the nodes represent functions and the arcs represent data paths, as shown in Fig.2[6]

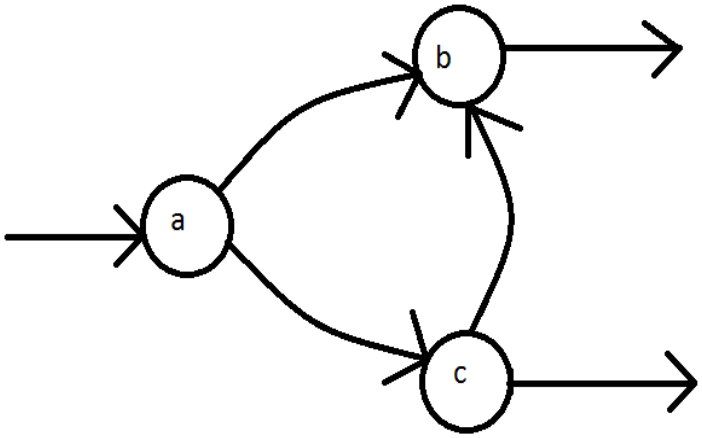

Fig. 2. A three node data flow graph with one input and two outputs.

The nodes represent functions of arbitrary complexity, and the arcs represent paths on which sequences of data (tokens or samples) flow. Signal processing algorithms are usually described in the literature by a combination of mathematical expressions and block diagrams. Block diagrams are large grain dataflow (LGDF)graphs, in which the nodes or blocks may be atomic (from the Greek atomos, or indivisible), such as adders or multipliers, or nonatomic (large grain), such as digital filters, FFT units, modulators, or phase locked loops. The arcs connecting blocks show the signal paths, where a signal is simply an infinite stream of data, and each data token is called a sample. The complexity of the functions (the granularity***)*** will determine the amount of parallelism available because, while the blocks can sometimes be executed concurrently, we make no attempt to exploit the concurrency inside a block. The functions within the blocks can be specified using conventional von Neumann programming techniques. If the granularity is

at the level of signal processing subsystems (second-order sections, butterfly units, etc.), then the specification of a system will be extremely natural and enough concurrency will be evident to exploit at least small-scale parallel processors. The blocks can themselves represent another data flow graph, so the specification can be hierarchical. This is consistent with the general practice in signal processing where, for example, an adaptive equalizer may be treated as a block in a large system, and may be itself a network of simpler blocks. LGDF is ideally suited for signal processing, and has been adopted in simulators in the past. Other signal processing systems use a data-driven paradigm to partition a task among cooperating processors and many block diagram languages have been developed to permit programmers to describe signal processing systems more naturally.

Although true asynchrony is rare in signal processing, multiple sample rates are common, stemming from the frequent use of decimation and interpolation. In addition to being natural for DSP, large grain data flow has another significant advantage for signal processing. As long as the integrity of the flow of data is preserved, any implementation of a data flow description will produce the same results. This means that the same software description of a signal processing system can be simulated on a single processor or multiple processors, implemented in specialized hardware, or even, ultimately, compiled into a VLSI chip.

2.1 SYNCHRONOUS DATAFLOW GRAPHS

A block is a function that is invoked when there is enough input available to perform a computation (blocks lacking inputs can be invoked at any time). When a block is invoked, it will consume a fixed number of new input samples on each input path. These samples may remain in the system for some time to be used as old samples **,** but they will never again be considered new samples. A block is said to be synchronous if we can specify a priori the number of input samples consumed on each input and the number of output samples produced on each output each time the block is invoked. Thus, a synchronous block is shown in Fig. 3(a) with a number associated with each input or output specifying the number of inputs consumed or the number of outputs produced. These numbers are part of the block definition. For example, a digital filter block would

have one input and one output, and the number of input samples consumed or output samples produced would be one. A 2:l decimator block would also have one input and one output, but would consume two samples for every sample produced. A synchronous data flow (SDF) graph is a network of synchronous blocks, as in Fig. 3(a).

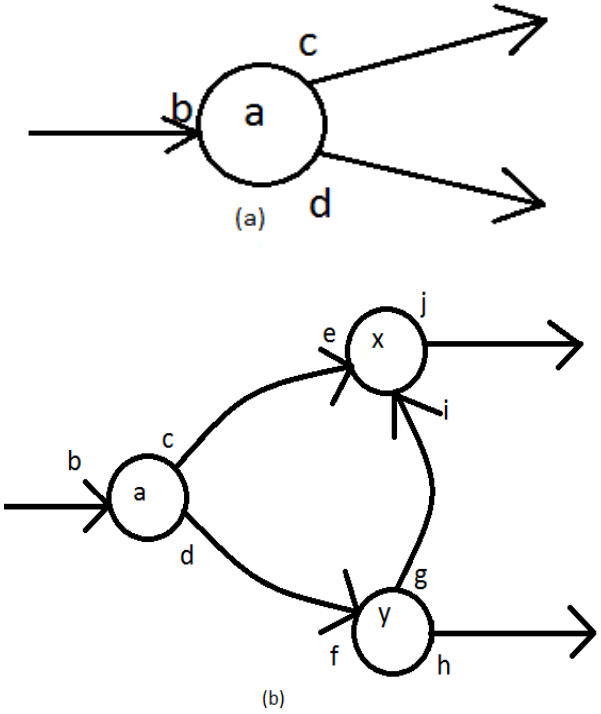

Fig 3 (a)synchronous node (b) synchronous data flow graph

SDF graphs are closely related to computation graphs. introduced in 1966 by Karp and Miller [10]and further explored by Reiter [11]. Computation graphs are slightly more elaborate than SDF graphs, in that each input to a block has two numbers associated with it, a threshold and the number of samples consumed. The threshold specifies the number of samples required to invoke the block, and may be different from the number of samples consumed by the block. It cannot, of course, be smaller than the number of samples consumed. For simplicity, it is assumed these two numbers are the same. Karp and Miller [10] show that computations specified by a computation graph are determinate, meaning that the same computations are performed by any proper execution. This type of theorem also underlies the validity of data flow descriptions. They also give a test to determine whether a computation terminates which is potentially useful because in signal processing we are mainly interested in computations that do not terminate. The assumption is that signal processing systems repetitively apply an algorithm to an infinite

sequence of data. To make it easier to describe such applications, the model is expanded slightly to allow nodes with no inputs. These can fire at any time. Computation graphs have been shown to be a special case of Petri nets [12]-[14]or vector addition system. These more general models can be used to describe asynchronous systems. There has also been work with models that are special cases of computation graphs. In 1971, Commoner and Holt described marked directed graphs**.** However, marked directed graphs are much more restricted samples produced or consumed on any arc to unity. This extensively restricts the sample rates in the system, reducing the utility of the model. In 1968, Reiter [15] simplified the computation graph model in much the same way (with minor variations), and tackled a scheduling problem.

However, his scheduling problem assumes that each node in the graph is a processor, and the only unknown is the firing time for the invocation of each associated function. Implementing the signal processing system described by a SDF graph requires buffering the data samples passed between blocks and scheduling blocks so that they are executed when data are available. This could be done dynamically, in which case a runtime supervisor determines when blocks are ready for execution and schedules them onto processors as they become free. This runtime supervisor may be a software routine or specialized hardware, and is the same as the control mechanisms generally associated with data flow. It is a costly approach, however, in that the supervisory overhead can become severe, particularly if relatively little computation is done each time a block is invoked.SDF graphs, however, can be scheduled statically (at compile time), regardless of the number of processors, and the overhead associated with dynamic control evaporates. Specifically, a large grain compiler determines the order in which nodes can be executed and constructs sequential code for each processor.

An SDF graph can be characterized by a matrix similar to the incidence matrix associated with directed graphs in graph theory.

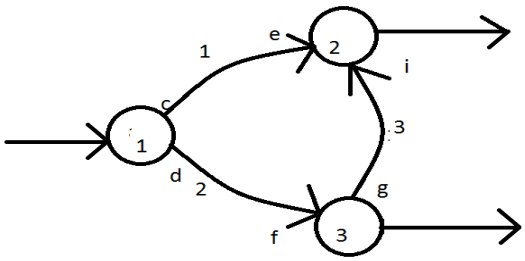

Fig 4

It is constructed by first numbering each node and arc, as in Fig. 4, and assigning a column to each node and a row to each arc. The (i, j)th entry in the matrix is the amount of data produced by node j on arc i each time it is invoked. If node j consumes data from arc i**,** the number is negative, and if it is not connected to arc i, then the number is zero. For the graph in Fig. 4 we get

$$t = \begin{bmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{bmatrix}$$

This matrix is called a topology matrix, and need not be square, in general. If a node has a connection to itself (a self-loop), then only one entry in topology matrix describes this link. This entry gives the net difference between the amount of data produced on this link and the amount consumed each time the block is invoked. This difference should clearly be zero for a correctly constructed graph, so the entry describing a self-loop should be zero.

Each arc can be replaced with a FIFO queue (buffer) to pass data from one block to another. The size of the queue will vary at different times in the execution. Define the vector b(n) to contain the queue sizes of all the buffers at time n. In Blosim [16], buffers are also used to store old samples (samples that have been consumed), making implementations of delay lines particularly easy. These past samples are not considered part of the buffer size here. For the sequential schedule, only one block can be invoked at a time, and for the purposes of scheduling it does not matter how long it runs. Thus, the index n can simply be incremented each time a block finishes and a new block is begun. The block invoked is specified at time n with a vector u(n), which has a one in the position corresponding to the number of the block that is invoked at time n and zeros for each block that is not invoked. For the system in Fig. 4, in a sequential schedule, v(n) can take one of three values, depending on which of the three blocks is invoked. Each time a block is invoked, it will consume data from zero or more input arcs and produce data on zero or more output arcs. The change in the size of the buffer queues caused by invoking

a node is given by b(n+1)=b(n)+ tau v(n) .The topology matrix characterizes the effect on the buffers of running a node program.

2.2 DEFINITION[2]

A SDFG is a finite directed multigraph G = <V, E, t, d, prd, cns>, in which:

1. V is the set of actors, modelling the functional elements of the system. Each actor $v \in V$ is weighted with its computation time t(v), a nonnegative integer;
2. E is the set of directed edges, modelling interconnections between functional elements. Each edge $e \in E$ is weighted with three properties: d(e), the number of initial tokens associated with e; prd(e), a positive integer that represents the number of tokens produced onto e by each execution of the source actor of e; cns(e), a positive integer that represents the number of tokens consumed from e by each execution of the sink actor of e. These numbers are also called the delay, production rate, and consumption rate, respectively. The source actor and sink actor of $e \in E$ are denoted as src(e) and snk(e), respectively.

The edge e is represented with source actor u and sink actor v mostly by e = <u, v>, and by e = <u, v, k>, where k numbers the edges connecting u to v, only when distinguishing different edges between two actors is necessary. The set of incoming edges to $v \in V$ is denoted by InE(v), and the set of outgoing edges from $v \in V$ by OutE(v). $v \in G$ is used to represent that v is an actor of G and $e \in G$ to represent that e is an edge of G .For technical reasons d(e) is allowed to be negative. If prd(e) = cns(e) = 1 for each $e \in E$, then we say that G is a homogeneous synchronous dataflow graph (HSDFG), also called a single-rate dataflow graph. An HSDFG is represented as G_h = <V, E, t, d>.

A delay distribution of a SDFG is a vector containing delays on all edges of the SDFG G, denoted as d(G). For example, the delay distribution of SSA in Fig. 1 is d(SSA) = [0, 0, 1, 0, 16, 1, 0] corresponding to the edges <A,B>, <B,C>, <C,D>, <D,E>, <E,A>, <D, F>, and <F,A>.

Applications for signal processing are usually nonterminating. Memory used must be bounded no matter how many times they are executed. In order that a SDFG G has a well-defined meaning, according to [6],[2] restrictions are placed on it.

1. d(e) $\geq$ 0 for each e $\in$ G.
2. G is bounded. Boundedness means that, if infinite execution sequences exist, then there are some for which the number of tokens on every edge is always finite.
3. G is live. Liveness means that there is no execution sequence leading to a deadlock.

A SDFG that satisfies conditions 1, 2 and 3 above is a valid SDFG. A SDFG *G* is sample rate consistent if there exists a positive integer vector q(V) such that for each edge e $\in$ G

$$q(src(e)) \times prd(e) = q(snk(e)) \times cns(e) \quad (1)$$

where (1) is called a balance equation. The smallest q is called the repetition vector [6].[2] q is used to represent the repetition vector directly. One iteration of a SDFG G is an execution sequence in which each actor v in G occurs exactly q(v) times.

In the SDFG SSA in Fig. 1for each edge e, its prd(e), cns(e) that are not equal to 1 and its d(e) are labelled on e. The computation time vector t = [1, 1, 2, 1, 1, 2]. A balance equation can be constructed for each edge. By solving these balance equations, SSA's repetition vector q = [16, 1, 1, 1, 4, 1] is achieved.

A SDFG is sample-rate inconsistent if there is no nonzero solution for its balance equations. Any execution of an inconsistent SDFG will result in deadlock or unbounded memory. A live SDFG is bounded if and only if it is sample-rate consistent. This is the necessary and sufficient condition for boundedness.

2.3 EQUIVALENT HSDFG

A sample-rate consistent SDFG can always be converted to an equivalent HSDFG, which captures the data dependences among firings of actors in the original SDFG in an iteration. In an iteration of a SDFG G, each actor *v* fires q(v) times. the *i*th firing of v can be mapped to an actor (v, i) in its equivalent HSDFG. Each edge e = <u, v> in *G* has q(v)cns(e) instances in this HSDFG. For each actor (u, i) in the equivalent HSDFG, the (v, j)s to which it is connected are determined by the production and consumption rates and delays on e. If there are delays on e, they are always consumed by initial firings of v. Algorithms for transforming a SDFG to its equivalent HSDFG appear in the literature [17],[18].

Definition [17]: Let G = <V, E, t, d, prd, cns> be a sample rate consistent SDFG and q its repetition vector. H maps G to its equivalent HSDFG H(G) = <V',E't',d'> as follows

1. For each v ∈ V and i ∈ [1, q(v)], there is an actor (v, i) ∈ V' with t'(v, i) = t(v).
2. For each e = <u, v> ∈ E, i ∈ [1, q(u)], k ∈ [1, prd(e)], and j=floor(((i − 1)prd(e) + (k − 1) + d(e)) / cns(e)) mod cns(e)q(v)+1

   there is an edge e'= <(u,i),(v,j),k> ∈ E' with

   d'(e') = floor((*i* − 1)prd(e) + (k − 1) + d(e)/cns(e)q(v)).

For (v, i) ∈ H(G), it is called the ith copy or the ith firing of v ∈ G; i is called the label of (v, i). d' is non decreasing with k. Therefore, for an equivalent HSDFG of a valid SDFG,if there exist edges between the actors (u, i) and (v, j) with zero delays, then the edge <(u, i), (v, j), k0>, where k0 is the lowest value among the edges between (u, i) and (v, j), has zero delays.

CHAPTER 3

RETIMING

Retiming is a transformation technique used to change the location of delay elements in a circuit without affecting the input/output characteristics of the circuit.[1] Retiming has many applications including reducing the clock period of the circuit, reducing the number of registers in the circuit, reducing the power consumption of the circuit and logic synthesis. Retiming can be used to increase the clock rate of a circuit by reducing the computation time of the critical path. The critical path is defined to be the path with the longest computation time among all paths that contain zero delays, and the computation time of the critical path is the lower bound on the clock period of the circuit.

The iteration period of a dataflow graph is limited by its topology, the computation time of its actors, and the delay distribution. The delay distribution indicates the numbers of initial tokens on edges of the SDFG. To decrease the iteration period and speed up a DSP algorithm, we need therefore to change any of these three factors. Restructuring a SDFG or rewriting functions of actors to adjust the computation time involves redesign of the whole model, which may need much effort, e.g., to verify the correctness of each actor and the functional correctness of the restructured model. If the new model still has a high iteration period, designers have to repeat the procedure again. iteration period, designers have to repeat the procedure again. An optimization method that preserves the functionality of the original SDFG is a preferred choice. Redistributing delays of a SDFG provides an opportunity to do so. Retiming is a graph transformation that redistributes the graph's delays while its functionality remains unchanged. Retiming can be defined either in a forward fashion, by which retiming an actor once means firing this actor once, or in a backward fashion, by which retiming an actor once means reversed firing this actor once. The two approaches to retiming are equivalent, in the sense that any retimed graph can be obtained through both a forward retiming and a backward retiming.

Given a SDFG G = <V, E, t, d, prd, cns>, a retiming of *G* is a function r : V → Z, specifying a transformation r of *G* into a new SDFG r(G) = <V, E, t, d_r , prd, cns>, where the delay function d_r is defined for each edge u −→v by the equation

$d_r(e) = d(e) + cns(e)r(v) − prd(e)r(u)$

CHAPTER 4

SOFTWARE USED

The software used for implementing the algorithms is MATLAB.

In the previous work that has been done in the field , the retiming vector has been calculated manually on HSDFG's or a new tool SDF3 has been used. To the best of my knowledge, MATLAB has never been used to find the execution times. And the tool previously mentioned SDF3 , is used in a limited scientific community and not as widely available or used as MATLAB.

4.1 ABOUT MATLAB

MATLAB is short for matrix laboratory. It is a special purpose computer software designed to perform engineering and scientific calculations. It was basically designed to perform matrix mathematics. But presently , it is flexible computing system capable of solving a wide range of technical problems.

The MATLAB program implements the MATLAB programming language. It provides a very exhaustive library of predefined functions to simplify any technical programming tasks.MATLAB is a huge program, with a very rich variety of functions. The most basic version of MATLAB without any tool kits is much richer than other programming languages. The toolkits extend the capability of basic MATLAB with many more functions in various specialities.

4.2 FEATURES OF MATLAB[19]

1. Its very easy to use. It is an interpreted language. The program can be used as a scratch pad to evaluate expressions typed at the command line , and it can also be used to execute large pre-written programs. Programs maybe easily written and modified with the built in integrated development environment and debugged with the MATLAB debugger. Since the language is so easy to use , it is ideal for the rapid prototyping of new programs. Many program development tools are provided to make the program easy to use. They include an integrated editor/debugger,online documentation and manuals, a workspace browser and extensive demos.

2. It is platform independent. It is supported on many different computer systems, providing a large measure of platform independence. Programs written on any platform will run on all of the other platforms and data files written on any platform may be read transparently on any other platform. Hence, it can be seen that programs written in MATLAB can migrate to new platforms when the needs of the user change.
3. It has an extensive library of pre-defined functions that provide tested and pre-packaged solutions to many basic technical tasks. There are hundreds of functions built into the MATLAB language , making the job of technical calculations much easier as compared to other languages. Also, there are many special purpose toolboxes available t help solve complex problems in specific areas. There is an extensive collection of free user contributed MATLAB programs that are shared through the mathworks website.
4. MATLAB has many integral plotting and imaging commands. The plots and images can be displayed on any graphical output device supported by the computer on which this language is running. It is an outstanding tool for visualizing technical data.
5. It includes tools that allow a programmer to interactively construct a graphical user interface of the program that is to be implemented. With this capability, the programmer can design sophisticated data analysis programs that can be operated by relatively inexperienced users.
6. Its flexibility and platform independence is achieved by compiling the programs into a device independent p-code, and then interpreting the p-code instructions at run time. A special MATLAB compiler is available. It can compile a program into a true executable that runs faster than the interpreted code.

4.3 MATLAB FUNDAMENTALS

The fundamental unit in this language is an array. An array is a collection of data values organized into rows and columns and known by a single name. Individual data values within an array may be accessed by including the name of the array followed by subscripts in parentheses that identify the row and a column of the particular value. Even scalars are treated as arrays-they are arrays with only one row and one column.

When MATLAB executes, it can display several types of windows that accept commands or display information. The three most significant windows are command windows ,where we enter commands; figure windows, which display plots and graphs and edit windows which permit a user to create and modify the programs . Also, MATLAB can display other windows that provide help and that allow the user to examine some of the additional windows given here.

4.4 ENTERING DATA

We have to implement digital signal processing algorithms in MATLAB. The algorithms involve applying retiming functions on synchronous data flow graphs so as to reduce the execution time of the iteration of the graph. Or in other words, we want the synchronous data flow graph to implement faster.

The input to the retiming algorithms that are to be implemented are synchronous data flow graphs. Now, MATLAB is a language that accepts inputs in array form. So we have to find a way to enter the various parameters of the synchronous data flow graphs in array form. The various parameters of the synchronous data flow graph are:

1. The set of actors modelling the functional elements of the system V.
2. The set of directed edges, modelling interconnections between functional elements, denoted by E.
3. The set of computation time of each node denoted by t(v), which is a non negative integer.
4. The set of delays d where d(e) is the number of initial tokens associated with e where is an edge that is a subset of E.
5. The set of produced values onto each edge , where prd(e) is a positive integer that represents the number of tokens produced onto e by each execution of the source actor of e.
6. The set of consumed values from each edge , where cns(e) is a positive integer that represents the number of tokens consumed from e by each execution of the sink actor of e.

4.5 REPRESENTATION

We enter the above six parameters by the following arrays.

1. A two dimensional topology matrix where (i, j)th entry in the matrix is the amount of data produced by node j on arc i each time it is invoked. If node j consumes data from arc i**,** the number is negative, and if it is not connected to arc i, then the number is zero. Hence by entering the topology matrix, we are entering four of the above parameters that is the nodes, edges and the produced and consumed values.
2. A one dimensional delay distribution matrix that gives the number of delay tokens on each edge.
3. A one dimensional computation time vector that gives the computation time needed to execute each node.
4. A one dimensional repetition vector that gives the number of times each node of a synchronous data flow graph needs to be executed in one iteration of the graph.

CHAPTER 5

RETIMING ALGORITHMS

5.1 ByHSDF[20][2]

This is the oldest algorithm to find out the retiming vector of a synchronous data flow graph. This algorithm can be directly applied to single rate data flow graphs. But it is not directly applicable to multirate synchronous data flow graphs. Hence, if we want to find the retiming vector of a multirate dataflow graph using this algorithm, we have to first convert it to a single rate data flow graph and then apply the algorithm. Converting the multirate graph to a single rate graph increases the problem space hugely as each node in the multirate dataflow graph will have multiple nodes in the single rate graph ,equal to that node's repetition vector. This is a very inefficient method as a lot of time is required to first convert the graph to the single rate graph and then applying it on the huge single rate graph that we get. Consequently, this algorithm gives the worst speed .

5.1.1 ALGORITHM 1: FEAS(G_h, dip)

INPUT : A valid HSDFG G_h= < V,E,t,d> and a non negative integer dip

OUTPUT: A retiming r of G_h such that r(G_h) is a valid SDFG with IP(r(G_h)) <=dip if such a retiming exists.

for all v ϵ V , let r(v)=0
G_r= G_h
get T and IP from CP (G_r)
if IP<=dip then
    return r
end if
for i = 1 to |V|-1 do
    for all vϵV do
        if (T(v)) > dip) then
        r(v)= r(v)+1
        end if
end for

G_r= r(G_h)

Get T and IP of G_r from CP(G_r)

if IP<=dip then

return r

end if

end for

return

5.1.2 ALGORITHM 2: CP[21]

INPUT: an HSDFG G= <V,E,d,t>

OUTPUT: the T and IP

Topologically sort the vertices of G with u preceding v if there is a zero delay edge from u to v in G

For all v in order of the sorted list do

if v has no zero delay incoming edge in G then

t(v) = t(u)

else

t(v)=t(v) + max (t(u)) |∃ e: u→v in G with d(e)=0

end if

end for

IP= max (t)

return T and IP

5.1.3 ALGORITHM 3: SDF to HSDF conversion[1]

q: repetition vector

for each node u in the SDFG

for q(u)=0 to q_u-1

There is a node u_k in the HSDFG with the same computation time as u in the SDFG

end for

end for

for each edge u→v in the SDFG with delay tokens d

for j=0 to prd(e)* q(u)-1

there is an edge $u^{j/prd(e)} \rightarrow v^{((j+d(e))/cns(e))\%q(u)}$ in the HSDFG

with ((j+d(e))/cns(e)*q(u) delays

end for

end for

5.1.4 EXPLANATION:

1. Algorithm 3 is used to convert an SDFG into an HSDFG , which is the format in which it can be input to the algorithm byHSDF.
2. Algorithm 2 is used to find the iteration period of the SDFG
3. Algorithm 1 is used to find the retiming vector, if one exists that fulfils the requirements i.e. the retiming vector when applied gives us an iteration period less than or equal to the desired iteration period.
4. It works by relaxation. After initializing the retiming vector as the zero vector and an auxiliary variable G_r as $G_{h,}$ it computes the T and IP of the original graph.If IP<=dip already holds, then no further retiming is needed.
5. Otherwise, at each iteration of the outer loop, it increases each entry $r(v)$ of the retiming vector by 1 when T(v) is larger than dip (the inner loop), trying to shorten those zero-delay paths, to form a retiming step.
6. Then it transforms the current HSDFG to a new graph according to this retiming step and compute the T and IP for the new graph to check whether its IP is not larger than dip; if not, then the accumulation of the previous retiming steps forms a feasible retiming.
7. Otherwise, the algorithm goes on to the next iteration until the bound of the outer loop ($|V| - 1$, where $|V|$ denotes the number of actors in V) is reached; if at that moment a Gr with IP(Gr) ≤ dip yet has not been found then it concludes that dip is not feasible for $G_{h.}$
8. The largest cycle in a graph includes at most $|V|$ edges and each retiming step increases entry r(v) at most 1, so the worst case is that $|V| - 1$ retiming steps cause

delays to travel through the largest cycle and yet no feasible retiming is found. Any more retiming steps may only lead to a delay distribution that has already been tested.

5.2 ZHU10[3]

In the previous algorithm, the major hurdle we faced was that even though we got the retiming vector, the execution time of the algorithm was very high because conversion to a single rate graph was required. In some extreme cases the single rate synchronous data flow graph could be exponentially larger than the original SDFG. In this method a relationship between an SDFG and its equivalent HSDFG is explored and the properties that are apparent help in computing the iteration period directly on SDFG's. And it is followed by an algorithm that directly computes the retiming function on the SDFG without converting it to an HSDFG.

5.2.1 ALGORITHM 4: FIPtest

INPUT : A valid SDFG G= < V,E,t,d,prd,cns> and a desired iteration period dip

OUTPUT: A retiming r of G such that r(G) is a valid SDFG with IP(r(G)) <=dip if such a retiming exists.

for all v ϵ V , let r(v)=0

G_r= G; i=1; istrue=false;

while i<=$\sum_{v\epsilon V} q(v)$ and istrue =false do

  istrue=true;

  get T(V)and eofZD(V) of G_r from IP()

  for all v ϵV do

    if (T(v)>dip) and eofZD(v) =true then

      istrue=false;

      r(v)=r(v)+1;

      reprev(v);

    end if

  end for

 if istrue=false then

  G_r=r(G); i++

```
  end if
end while
if istrue=true then
return r
else
return null
end if
```

5.2.2 PROCEDURE 1: reprev()

```
For all e ϵ InE(v) do
    u=src(e)
    if (T(u)>dip) and eofZD(u)=false and prd(e)<=cns(e) +d(e)
    then
          if (for all e1 ϵ OutE(u)-{e}, d(e1) >=prd(c1)) then
            r(u) = r(u) +1;
            reprev(u);
          end if
     end if
end for
```

5.2.3 ALGORITHM 5: IP(G)

```
INPUT : A valid SDFG G= < V,E,t,d,prd,cns> and q
OUTPUT: T(V),eofZD(V), ip
V0 ={vϵV: for all eϵ InE(v),d(e)>=cns(e)}
for all vϵV do
   T(v)=T1(V) =t(v)
   eofZD(v)=false
end for
for all vϵ V0 do
    T1(v)= t(v)
```

```
    getnext(v,1)
end for
ip=max vϵV T(v)
```

5.2.4 PROCEDURE2: getnext(u,l)

```
isEof =true; curt= T1(u)
for all eϵ OutE(u) do
     l1= floor [((l-1)prd(e) +d(e))/cns(e)]+1
     unext= snk(e)
     if l1 <= q(unext) then
          isEof=false;
          if T(unext)< T1(u) +t(unext) then
             T(unext)=T1(u) +t(unext)
          end if
          T1(unext) =T1(u) + t(unext)
          getnext(unext,l1)
      end if
  T1(u)= currT
end for
if isEof= true then
   eofZD(u)=true;
end if
```

5.2.5 EXPLANATION:

1. The algorithm FIPtest is a method to find the retiming function directly on SDFG.
2. The procedure reprev(v) improves the efficiency of the algorithm above by checking previous nodes of v and increasing the value of r if only the retiming is still legal.
3. The algorithm IP calculates the iteration period of the SDFG. No conversion to an HSDFG is required.
4. Procedure getnext(u,l) is a variation of the depth first search algorithm. It is executed to find the longest path to a particular node that is without delays.

5.3 sdfFEAS[2]

This is the latest method for finding the retiming function of a synchronous data flow graph. It is the most efficient of the three. Unlike any previous method, backtracking is not used to find the retiming functions to determine the change of delays in the procedure, instead retiming function is computed according to part of the computation time of critical walks and their subwalks. A walk is a sequence of actors and edges which may be repeated. In the SDFG these walks may or may not have delays, but in the equivalent HSDFG these walks correspond to path without delays. They are referred to as critical walks. The critical walks ultimately determine the IP.

5.3.1 ALGORITHM 6: sdfFEAS(G,dip)

INPUT : A valid SDFG G_h= < V,E,t,d,prd,cns> and a non negative integer dip

OUTPUT: A retiming r of G such that r(G) is a valid SDFG with IP(r(G)) <=dip if such a retiming exists.

for all v ϵ V , let r(v)=0

G_r= G

get criT and IP of G_r from sdfIP (G_r)

if IP<=dip then

    return r

end if

for i = 1 to $\sum_{v\epsilon V}$ q(v)-1 do

    for all vϵV do

        get lcriTdip(v) from criT

        nTdip(v)=q(v) – lcriTdip(v)+1

        r(v)= r(v) + nTdip(v)

    end for

G_r= r(G)

Get criT and IP of G_r from sdfIP(G_r)

if IP<=dip then

    return r

end if

end for

return

5.3.2 ALGORITHM 7: sdfIP(G)

INPUT : A valid SDFG G= < V,E,t,d,prd,cns>

OUTPUT: criT and IP

V_0 ={vϵV: for all eϵ InE(v),d(e)>=cns(e)}

for all vϵG, iϵ[1,q(v)] let criT(v,i) = -1

for all vϵV_0 do

criT(v,1) =t(v)

getnextT(v,1)

end for

ip = max $_{v\epsilon V,\ i\epsilon[1,\ q(v)]}$ criT(v,i)

return IP and criT

5.3.3 PROCEDURE 3: getnextT(u,l)

for all eϵ OutE(u) do

l_1= floor [((l-1)prd(e) +d(e))/cns(e)]+1

unext= snk(e)

if l_1 <= q(unext) then

if criT(unext,l_1)< criT(u,l) +t(unext) then

criT(unext,l_1)=criT(u,l) +t(unext)

getnext(unext,l_1)

end if

end if

end for

5.3.4 EXPLANATION:

1. The algorithm sdfFEAS calculates the retiming vector of the synchronous data flow graph. It employs a vector criT which holds the computation time of all the walks and subwalks.
2. sdfIP calculates the iteration period of the synchronous data flow graph. This it does with the help of criT. No backtracking is used as has been previously used in the other two algorithms.
3. getnextT(u,l) subtly goes through the critical walks and finds the maximum computation time of each node.
4. nTdip(v) holds the number of copies of v in the HSDFG which have computation times of maximal zero delay paths greater than the desired iteration period.
5. lcriTdip(v)ϵ [1,q(v)+1] and it is the lowest label that has criT(v,i)>dip. If there is no such label , then lcriTdip(v) = q(v) +1.

CHAPTER **6**

THE TEST CIRCUITS USED

6.1 A simplified spectrum analyzer[2]

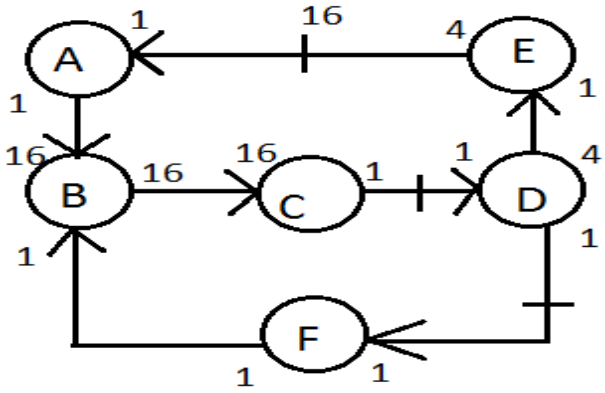

Fig 5 Simplified spectrum analyzer

The various blocks of this analyzer are:

TABLE 2 Function of each node

Actor	Function
A	Adoptive Low Pass Filter
B	FFT zoom
C	Peak Detector
D	Interpolator
E	Decision
F	Zoom Contol

Number of nodes: 6

Number of edges: 7

The computation time vector is [1 1 2 1 1 2]

The delay distribution vector is [16 0 0 1 0 0 1]

The repetition vector is [16 1 1 1 4 1]

6.2. Test graph 2 [2]

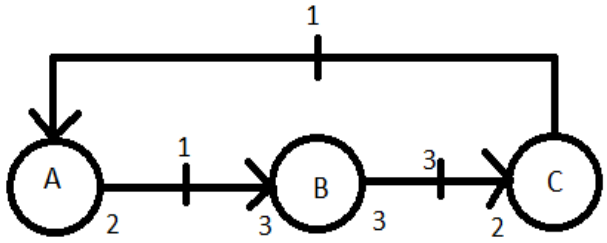

Fig 6. Test graph 2

Number of nodes: 3

Number of edges: 3

The computation time vector is [2 1 1]

The delay distribution vector is [1 3 1]

The repetition vector is [3 2 3]

6.3. Test graph 3[2]

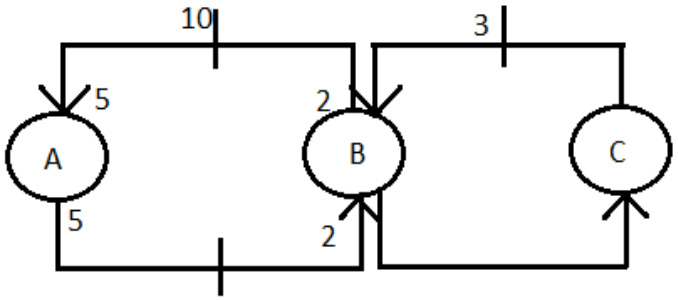

Fig 7. Test graph 3

Number of nodes: 3

Number of edges: 4

The computation time vector is [5 1 1]

The delay distribution vector is [4 0 3 10]

The repetition vector is [2 5 5]

6.4.Test graph 4[3]

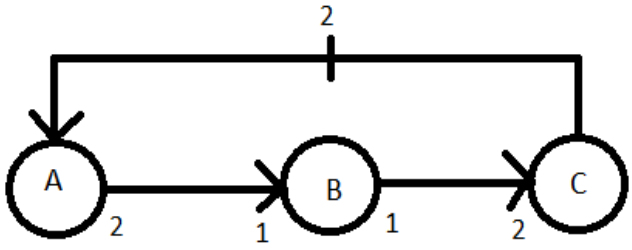

Fig 8. Test graph 4

Number of nodes: 3

Number of edges: 4

The computation time vector is [2 1 2]

The delay distribution vector is [0 0 2]

The repetition vector is [1 2 1]

6.5. MP3 playback application[22]

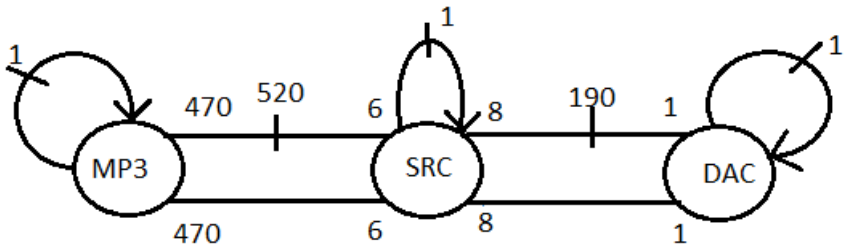

Fig 9 MP3 playback application

Number of nodes: 3

Number of edges: 7

The computation time vector is [1 1 1]

The delay distribution vector is [0 520 1 1 0 190 0]

The repetition vector is [2 235 1880]

This is a real life application. The MP3 playback application consists of three tasks: a block reader, an MP3 decoder and a sample rate converter. This is the description of the first block. The output of the sample rate converter is fed to a digital-to-analog converter that forms the time triggered interface with the environment. For analysis purpose, the

interface is considered as a task . In this application, the MP3 decoder consumes a number of bytes from its input buffer that depends on the processed data stream.

6.6. Audio echo canceller [22]

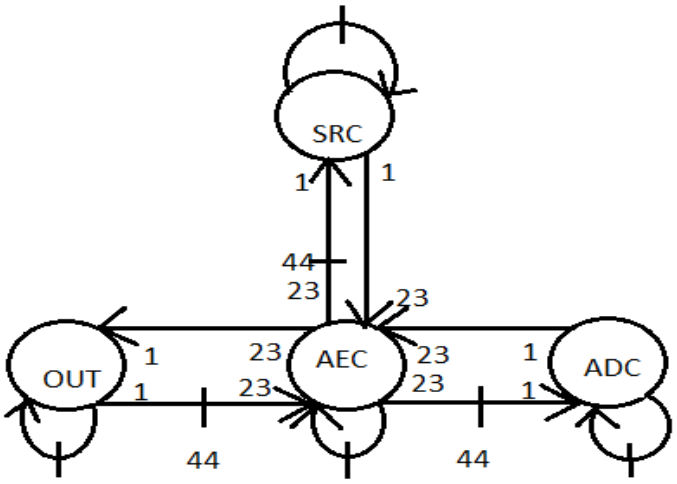

Fig 10 Audio echo canceller

Number of nodes: 4

Number of edges: 10

The computation time vector is [1 1 1 1]

The delay distribution vector is [0 44 44 0 44 0]

The repetition vector is [23 23 1 23]

This is a real life application.

6.7. Test graph 7[1]

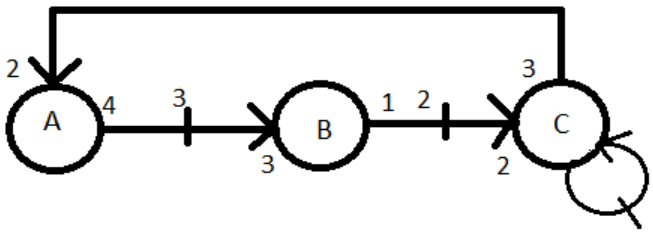

Fig 11. Test graph 7

Number of nodes: 3

Number of edges: 4

The computation time vector is [1 1 1]

The delay distribution vector is [3 2 0]

The repetition vector is [3 4 2]

6.8. Bipartite graph[22]

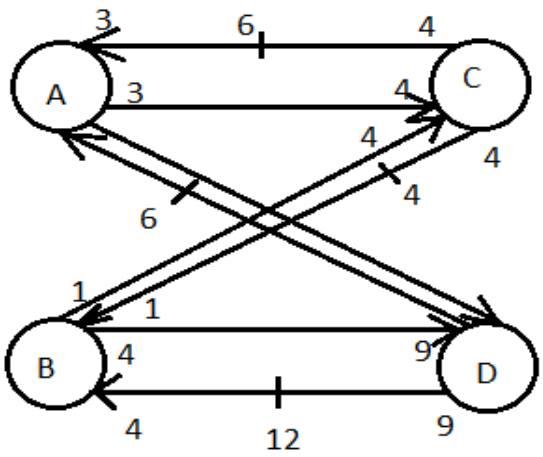

Fig 12 Bipartite graph

Number of nodes: 4

Number of edges: 8

The computation time vector is [1 1 1 1]

The delay distribution vector is [0 6 0 6 0 4 0 12]

The repetition vector is [12 36 9 16]

CHAPTER 7

RESULTS

The above three algorithms- byHSDF, ZHU10 and sdfFEAS have been implemented in MATLAB and the execution time of the above three algorithms, when applied to various test circuits has been observed.

7.1FUNCTION USED

MATLAB has two commands that help us to find the execution time of any algorithm.

1. tic : It starts a stopwatch timer.
2. toc: This function stops the stopwatch timer started by the tic function and displays the time elapsed in seconds.

It also has a run and time command that runs the central processing unit and tells the execution time. It tells the time elapsed to run every function so that performance can be optimized. We will also look into the processor utilization of each function for each test circuit so as to know how we get the execution time that we get.

Table3. Comparison of Execution times of sdfFEAS, Zhu10 and Byhsdf algorithms for various test circuits

Test circuit no.	SdfFEAS	Zhu10	Byhsdf
1	0.065235 sec	0.105737 sec	0.166021 sec
2	0.073846 sec	0.082835 sec	0.085296 sec
3	0.049687 sec	0.109300 sec	0.148024 sec
4	0.049687 sec	0.070477 sec	0.086840 sec
5	0.089520 sec	3.495350 sec	45 min
6	0.052383 sec	0.082654 sec	3.19322 sec
7	0.079635 sec	0.089133 sec	0.145123 sec

7.2 EXECUTION TIME OF VARIOUS SUB COMPONENTS

7.2.1 sdfFEAS

7.2.1.1 Spectrum analyzer

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
main_program_test	1	0.111 s	0.078 s	
sdflP1	3	0.028 s	0.019 s	
getnext	21	0.009 s	0.009 s	
retiming	2	0.004 s	0.004 s	
lcriT	12	0.001 s	0.001 s	

7.2.1.2 Test graph 2

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
main_program_test	1	0.088 s	0.053 s	
sdflP1	3	0.029 s	0.017 s	
getnext	14	0.012 s	0.012 s	
retiming	2	0.004 s	0.004 s	
lcriT	6	0.002 s	0.002 s	

7.2.1.3 Test graph 3

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
main_program_test	1	0.082 s	0.054 s	
sdflP1	2	0.023 s	0.014 s	
getnext	13	0.009 s	0.009 s	
retiming	1	0.004 s	0.004 s	
lcriT	3	0.001 s	0.001 s	

7.2.1.4 Test graph 4

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
main_program_test	1	0.086 s	0.054 s	
sdflP1	2	0.024 s	0.016 s	
getnext	6	0.008 s	0.008 s	
retiming	1	0.006 s	0.006 s	
lcriT	3	0.002 s	0.002 s	

7.2.1.5 MP3 playback application

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
main_program_test	1	0.134 s	0.055 s	
getnext	305	0.035 s	0.035 s	
sdflP1	2	0.058 s	0.023 s	
lcriT	3	0.017 s	0.017 s	
retiming	1	0.004 s	0.004 s	

7.2.1.6 Audio echo canceller

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
main_program_test	1	0.087 s	0.054 s	
sdflP1	2	0.027 s	0.016 s	
getnext	8	0.011 s	0.011 s	
retiming	1	0.005 s	0.005 s	
lcriT	4	0.001 s	0.001 s	

7.2.1.7 Test graph 7

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
main_program_test	1	0.090 s	0.054 s	
sdflP1	9	0.029 s	0.018 s	
getnext	58	0.011 s	0.011 s	
retiming	8	0.004 s	0.004 s	
lcriT	24	0.003 s	0.003 s	

7.2.2 ZHU10

7.2.2.1 Simplified spectrum analyzer

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
zhu10_main	1	0.121 s	0.076 s	
zhu10_getnext	52	0.018 s	0.018 s	
zhu10_IP1	6	0.035 s	0.017 s	
retiming	5	0.006 s	0.006 s	
zhu10_reprev	5	0.004 s	0.004 s	

7.2.2.2 Test graph 2

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
zhu10_main	1	0.113 s	0.075 s	
zhu10_IP1	4	0.027 s	0.014 s	
zhu10_getnext	22	0.013 s	0.013 s	
retiming	3	0.006 s	0.006 s	
zhu10_reprev	3	0.005 s	0.005 s	

7.2.2.3 Test graph 3

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
zhu10_main	1	0.110 s	0.068 s	
zhu10_getnext	33	0.016 s	0.016 s	
zhu10_IP1	4	0.030 s	0.014 s	
retiming	3	0.007 s	0.007 s	
zhu10_reprev	3	0.005 s	0.005 s	

7.2.2.4 Test graph 4

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
zhu10_main	1	0.098 s	0.067 s	
zhu10_IP1	2	0.023 s	0.012 s	
zhu10_getnext	6	0.011 s	0.011 s	
retiming	1	0.005 s	0.005 s	
zhu10_reprev	1	0.003 s	0.003 s	

7.2.2.5 MP3 playback application

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
zhu10_getnext	36579	4.185 s	4.185 s	
zhu10_main	1	4.372 s	0.115 s	
zhu10_IP1	81	4.231 s	0.046 s	
retiming	80	0.018 s	0.018 s	
zhu10_reprev	80	0.008 s	0.008 s	

7.2.2.6 Audio echo canceller

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
zhu10_main	1	0.142 s	0.074 s	
zhu10_getnext	168	0.028 s	0.028 s	
zhu10_IP1	24	0.055 s	0.027 s	
retiming	23	0.010 s	0.010 s	
zhu10_reprev	23	0.003 s	0.003 s	

7.2.2.7 Test graph 7

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
zhu10_main	1	0.117 s	0.073 s	
zhu10_getnext	66	0.019 s	0.019 s	
zhu10_IP1	9	0.033 s	0.014 s	
zhu10_reprev	9	0.005 s	0.005 s	
retiming	9	0.005 s	0.005 s	

7.2.3 byHSDF

7.2.3.1 Spectrum analyzer

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
byHSDF_conversion	1	0.166 s	0.115 s	
CP	1	0.037 s	0.020 s	
cp2_getnext	31	0.017 s	0.017 s	
byHSDF1	1	0.051 s	0.014 s	

7.2.3.2 Test graph 2

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
byHSDF_conversion	1	0.154 s	0.096 s	
cp2_getnext	64	0.018 s	0.018 s	
byHSDF1	1	0.057 s	0.017 s	
CP	3	0.032 s	0.014 s	
retiming	2	0.008 s	0.008 s	

7.2.3.3 Test graph 3

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
byHSDF_conversion	1	0.148 s	0.097 s	
byHSDF1	1	0.051 s	0.017 s	
cp2_getnext	32	0.014 s	0.014 s	
CP	2	0.026 s	0.012 s	
retiming	1	0.008 s	0.008 s	

7.2.3.4 Test graph 4

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
byHSDF_conversion	1	0.114 s	0.081 s	
byHSDF1	1	0.033 s	0.013 s	
CP	2	0.016 s	0.009 s	
cp2_getnext	9	0.007 s	0.007 s	
retiming	1	0.004 s	0.004 s	

7.2.3.5 MP3 playback application

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
byHSDF_conversion	1	0.497 s	0.323 s	
cp2_getnext	85	0.114 s	0.114 s	
CP	1	0.162 s	0.048 s	
byHSDF1	1	0.174 s	0.012 s	

7.2.3.6 Audio echo canceller

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
CP	70	4.535 s	2.397 s	
cp2_getnext	4803	2.138 s	2.138 s	
retiming	69	1.593 s	1.593 s	
byHSDF_conversion	1	6.322 s	0.175 s	
byHSDF1	1	6.147 s	0.019 s	

7.2.3.7 Test graph 7

Function Name	Calls	Total Time	**Self Time***	Total Time Plot (dark band = self time)
byHSDF_conversion	1	0.215 s	0.097 s	
cp2_getnext	702	0.078 s	0.078 s	
retiming	8	0.014 s	0.014 s	
CP	9	0.091 s	0.013 s	
byHSDF1	1	0.118 s	0.013 s	

CHAPTER 8

8.1 CONCLUSIONS

The following conclusions have been drawn after analyzing the results.

1. It is observed that the results for sdfFEAS are the best of the three algorithms. This is observed because in this algorithm, explicit conversion to a homogeneous synchronous data flow graph is not required as in byHSDF . Also, we do not implicitly explore the entire HSDF as is done in ZHU10. Here only critical walks are being explored which give us sufficient information so as to calculate the retiming vector.
2. For graph no. 5 , we notice that the time taken by byHSDF is very large. When we look at the repetition vector, we see that the values of repetition vector of nodes is as high as 1880. Hence, first the algorithm has to construct an HSDFG with such a huge number of nodes, and then go through the nodes in a depth first search manner. Such large times are hence unavoidable.
3. For test graph 6 also, we make a similar observation. Again, the repetition vector is as high as 23. and the number of delays is 44. Hence many permutations have to be checked to find the retiming vector.
4. The maximum time is consumed by the main module ZHU10_main in ZHU10 algorithm and the conversion procedure in byHSDF. This happens as graphs scale up i.e as the repetition vector increases. But main module of ZHU10 can be further optimized.

8.2 FUTURE WORK

Further improvements can be done in the retiming algorithms so as to reduce the execution time further, in order to meet the real time requirement.

1. Exploration of various transformation techniques such as unfolding in tandem with retiming to achieve the above stated goal.
2. Memory usage optimization can be explored using tools such as Valgrind.

CHAPTER 9

REFERENCES

[1] Keshab K. Parhi, "VLSI Digital Signal Processing, Design and Implementation," by John Wiley and sons, 2011.

[2] Xue-Yang Zhu, Twan Basten, Marc Geilen and Sander Stuijk, " Efficient Retiming of Multirate DSP Algorithms," in *IEEE Transactions on computer aided design of integrated circuits and systems*, vol. 31, issue no. 6, June 2012.

[3] Xue-Yang Zhu, "Retiming Multi-Rate DSP Algorithms to Meet Real-Time Requirement," in *Design, Automation & Test in Europe Conference & Exhibition*, pp. 1785- 1790, July 2010

[4] Xue-Yang Zhu, Marc Geilen, Twan Basten and Sander Stuijk, "Static Rate-Optimal Scheduling of Multirate DSP Algorithms via Retiming and Unfolding," in *IEEE 18th Real Time and Embedded Technology and Applications Symposium*,pp.109-118, 2012

[5] Vojin Zivojnovic, Sebastian Ritz and Heinrich Meyr, "Retiming of DSP programs for optimum vectorization," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, ICASSP-94., pp. II/465 - II/468 vol.2, 1994

[6] Edward Ashford Lee and David G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," in *IEEE Transactions on computers*, vol.C-36, issue no.1,pp. 24-35, January 1987

[7] Shenshen Gu, "Improved Reachability Analysis on Retiming of DSP Algorithms with an Efficient Cell Enumeration Method," in *International Conference on Information Science and Technology(ICIST)*,pp. 572-576, March 26-28, 2011

[8] Vojin Zivojnovic, Sebastian Ritz and Heinrich Meyr, " High Performance DSP Software Using Data-Flow Graph Transformations," in *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, vol. 1, pp.492- 496 ,1994

[9] Edward Ashford Lee and David G. Messerschmitt, "Pipeline Interleaved Programmable DSP's: Synchronous Data Flow Programming," in *IEEE Transactions on acoustics, speech, and signal processing*, vol.35, issue no. 9, pp. 1334 -1345, September 1987

[10] R.M. Karp and R.E. Miller, “Properties of a model for parallel computations: determinancy, termination, queueing” *SIAM J.*. vol 14, pp.1390-1411, Nov 1966

[11] R.Reiter, “A study of a model for parallel computations” *Ph. D dissertation. Univ Michigan.* Ann Arbor 1967

[12] J.L. Peterson, “Petri nets,”*Comput. Sur* vol 9.Sept 1977

[13] “Petrinet Theory and the Modelling of systems,” Engelwood Cliffs, NJ:Prentice Hall, 1981

[14] T. Agarwala, ”Putting Petri nets to work ” *Computer* p.85, Dec 1979

[15] R. Reiter, “Scheduling parallel computations”. *J Ass. Comput. Mach.* ,vol 14, pp. 590-599. 1968

[16] D. G. Messerschmitt, “A tool for structured functional simulation,” *IEEE J. Select Areas Commun*. Vol. SAC-2, Jan 1984

[17] S. Sriram and S.S. Bhattacharyya, “Embedded Multiprocessors: Scheduling and Synchronization,” Boca Raton, FL: CRC Press ,2009

[18] G. Sih, “Multiprocessor scheduling to account for interprocessor communication,” *Ph. D dissertation, Electron. Res. Lab., Univ*. California, Berkeley,CA,1991

[19] Stephan J. Chapman, “MATLAB Programming for Engineers,” by Thomson Learning

[20] E. A Lee, ”A coupled hardware and software architecture for programmable digital signal processors,” *Ph. D. dissertation, Electron. Res. Lab., Univ. California,* Berkeley,CA 1986

[21] T. O’Neil and E.H.M. Sha,”Retiming synchronous dataflow graphs to reduce execution times,” IEEE Trans. Signal Process., vol 49,no. 10, pp 2397-2407, Oct 2001

[22] Waheed Ahmed,Robert de Groote, Philip K. F.Holzenspies, Marielle Stoelinga and Jaco van de Pol,”Resource Constrained Optimal Scheduling of SDF Graphs via Timed Automata,” *University of Twente,* The Netherlands