

Component Selection and Pipelining using Stochastic Evolution

by

Mohammad Farook

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

June, 1996

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

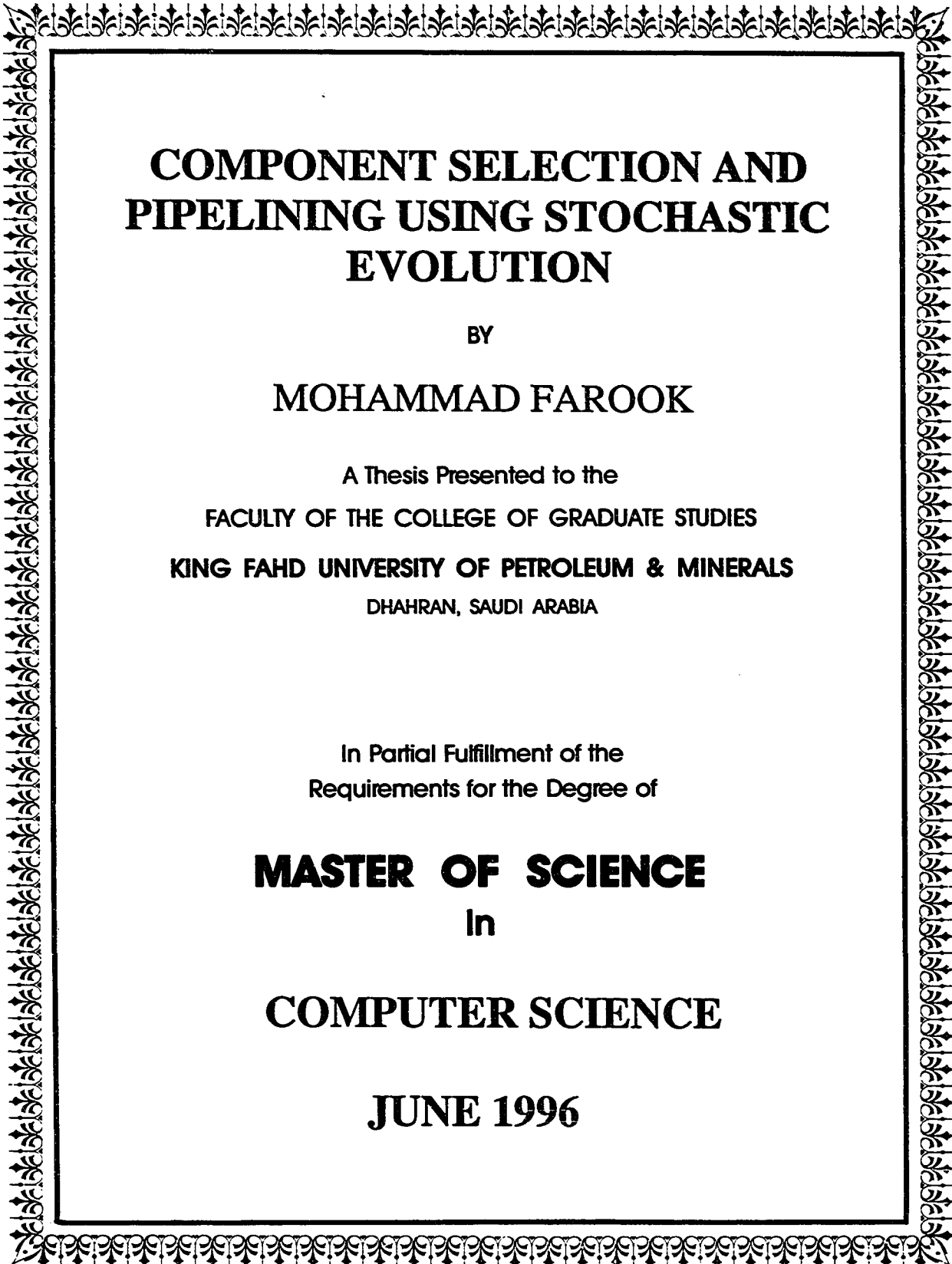
In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600



COMPONENT SELECTION AND PIPELINING USING STOCHASTIC EVOLUTION

BY

MOHAMMAD FAROOK

A Thesis Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

JUNE 1996

UMI Number: 1380769

UMI Microform 1380769
Copyright 1996, by UMI Company. All rights reserved.
This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

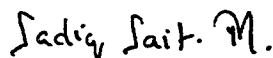
COLLEGE OF GRADUATE STUDIES

This thesis, written by MOHAMMAD FAROOK under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in COMPUTER SCIENCE.

THESIS COMMITTEE



Dr. Talal H. Maghrabi (Chairman)



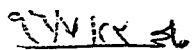
Dr. Sadiq M. Sait (Co-Chairman)



Dr. Khalid Al-Tawil (Member)



Dr. Jarallah Al-Ghamdi (Member)



Department Chairman



Dean, College of Graduate Studies

Date: 24.7.96



Dedicated to

My Parents and Brothers

Acknowledgments

First and foremost I thank Almighty Allah who gave me the opportunity, courage and patience to carry out this work.

Acknowledgement is due to King Fahd University of Petroleum & Minerals for providing support to this work.

I am indebted to my thesis chairman, Dr. Talal H. Maghrabi and Co-chairman Dr. Sadiq M. Sait for their help and advice. I acknowledge them for their encouragement and support. I would like also to place on record my appreciation for the cooperation and guidance extended by my committee members, Dr. Khalid Al-Tawil and Dr. Jarallah Al-Ghamdi. I am also thankful to the department chairman, Dr. Mulhem and other faculty members and staff for their cooperation.

I wish to express my gratitude to my parents, brothers, sisters-in-law, niece and nephews for their support, encouragement and motivation.

To all my friends, who have made my stay at KFUPM a memorable one, I thank you for your wonderful company and for being there when I needed the most.

Contents

Acknowledgements	iv
List of Figures	viii
List of Tables	x
Abstract (English)	xi
Abstract (Arabic)	xii
1 Introduction	1
1.1 Problem Definition	7
1.2 Literature Survey	10
1.2.1 Allocation techniques	14
1.2.2 Scheduling techniques	16
1.2.3 Pipelining	21
1.3 Solution techniques	24
1.3.1 Iterative techniques	24

1.3.2	Constructive techniques	26
1.4	Summary	28
2	Stochastic Evolution and Simulated Annealing	29
2.1	Stochastic Evolution	30
2.1.1	Algorithm	30
2.1.2	PERTURB	33
2.1.3	UPDATE	34
2.1.4	Issues Concerning Stochastic Evolution Algorithm	35
2.2	Simulated Annealing	36
2.2.1	Background	37
2.3	Algorithm	40
2.4	Problem-specific decisions	42
2.5	Summary	43
3	Component Selection and Pipelining using Stochastic Evolution	44
3.1	Initial Solution Representation	45
3.2	Cost Function	48
3.3	PERTURB Strategy	49
3.3.1	Random Moves	49
3.3.2	Fixed Random Moves	50
3.3.3	Top-Down Strategy	52
3.3.4	Bottom-up strategy	54
3.3.5	Perturb Function	56

3.4	Post Processing	59
3.5	Update Function	60
3.6	Parallelizing Stochastic Evolution Algorithm	61
3.7	Summary	64
4	Component Selection and Pipelining using Simulated Annealing	65
4.1	Initial, Current and Best Solution	66
4.2	Cost Function	66
4.3	Generation of Moves	66
4.4	Metropolis Function	67
4.5	Tuning of Parameters	69
4.6	Parallelizing simulated annealing	70
4.7	Summary	72
5	Experimental Results	73
6	Conclusions and Future work	84
6.1	Conclusions	84
6.2	Future Work	85
	Bibliography	86
	Vita	

List of Figures

1.1	Relationship of HLS with logic and layout synthesis.	3
1.2	HLS in design hierarchy.	4
1.3	Steps involved in HLS.	6
1.4	A sample data flow graph.	9
1.5	One possible solution to the example shown in Figure 1.4.	11
1.6	Another solution for the example shown in Figure 1.4.	11
1.7	Different states of the solution space.	13
1.8	Example of ASAP scheduling [4].	18
1.9	Example of list scheduling [4].	18
1.10	Example of FDS [4].	20
2.1	HLS system.	31
2.2	Stochastic Evolution algorithm.	32
2.3	PERTURB Function.	34
2.4	UPDATE procedure.	35
2.5	Simulated Annealing algorithm.	39
2.6	Metropolis Procedure.	41

3.1	Random move strategy.	51
3.2	Fixed random moves.	52
3.3	Top-Down strategy.	53
3.4	Bottom-Up Strategy.	55
3.5	PERTURB algorithm.	57
3.6	PERTURB with post processing.	60
4.1	The Metropolis Algorithm.	68
5.1	Plot showing the comparison between SA and MSE during the initial stages.	77
5.2	Barchart showing the results of SA and MSE during the initial stages.	78
5.3	A barchart showing the comparisons between SA, SE and MSE (post-processing).	79
5.4	Plots for different DFG's using SA and SE techniques: (a) EWF, (b) Graph 1, and (c) Graph 2 (contd).	80
5.5	Plots for different DFG's using SA and SE techniques: (d) Graph 3, (e) Graph 4, and (f) Graph 5.	81
5.6	Plots for different DFG's using SA and MSE (post-processing) techniques: (a) EWF, (b) Graph 1, and (c) Graph 2 (contd).	82
5.7	Plots for different DFG's using SA and MSE (post-processing) techniques: (d) Graph 3, (e) Graph 4, and (f) Graph 5.	83

List of Tables

1.1	Example of a component library.	10
3.1	Component Library.	47
5.1	Characteristic of input graphs.	74
5.2	Stochastic Evolution results.	75
5.3	Stochastic Evolution and Simulated Annealing results.	76
5.4	Modified Stochastic Evolution (post-processing) and Simulated Annealing results.	76
5.5	Comparisons between SA, SE and MSE (post-processing).	79

ABSTRACT

Name: Mohammad Farook
Title: Component Selection and Pipelining
by stochastic evolution:
Degree: Master of Science
Major Field: Information & Computer Science
Date of Degree: June 1996

High-level synthesis is the process of translating a high-level program like specification of the behavior of a digital circuit into a structural design in terms of interconnected set of Register-Transfer level components. Component selection and pipelining is one of the important problems in HLS. We investigate the application of Stochastic Evolution (SE) for solving component selection and pipelining and compare it with Simulated Annealing (SA) for the same computation time. The inputs are a Data Flow Graph (DFG), a realistic component library with multiple implementations of operators and Latency and Pipe stage delay constraints. Component selection involves replacing components of the DFG by slower components to minimize the cost. The cost function is the sum of costs (in gates) of all the components of the DFG and the pipeline registers. Pipelining is done based on the constraints of latency and pipe stage delay specified. A new method of improving the results in SE, called post-processing is proposed. This is called Modified Stochastic Evolution (MSE) technique. In post-processing, after obtaining a valid state the DFG is scanned to see if there is a possibility of replacing one or more components by slower components of the same type without violating the constraints. Experiments were carried out on different types of DFGs. The performance of SA is better than SE without post-processing, while SE performs better than SA in some cases when post-processing is introduced.

Master of Science Degree
King Fahd University of Petroleum and Minerals, Dhahran.
June 1996

خلاصة الرسالة

الاسم : محمد فاروق
عنوان الرسالة : اختيار المكونات و خط التوصيل التسلسل العشوائى
التخصص : علم الحاسب و المعلومات
تاريخ الشهادة : يونيو ١٩٩٦م

التصميم عالى المستوى هو عملية ترجمة برنامج عالى المستوى مثل تحديد وصف دائرة رقمية الى تصميم هيكلى باستعمال مجموعة متصلة من المكونات ذات مستوى المسجل الناقل. اختيار المكونات و خط التوصيل من المواضيع المهمة فى تصميم عالى المستوى. نبحت استخدام التسلسل العشوائى لحل اختيار المكونات و خط التوصيل و منه ثم نقارنه مع محاكاة التلدية لنفس وقت التحسين. والمدخلات هي رسم تدفعة المعلومات و مكتبة مكونات واقعية تحوى تنفيذات متعددة للعوامل شروط تاخير. مرحلة التوصيل و وقت الانتظار. اختيار المكونات تشمل تبديل بعض مكونات رسم تدفعة المعلومات بأخرى أبطأ لتقليل التكلفة. ان دالة التكلفة هي مجموع تكاليف جميع المكونات رسم تدفعة المعلومات و مسجلات خط التوصيل. خطوط الاتصال تبني اعتمادا على وقت الانتظار و تاخير مرحلة التوصيل. و في هذا البحث نفقح طريقة تسلسل العشوائى معدلة و ذلك باعادة النظر فى امكانية تبديل أحد المكونات أو أكثر بمكونات ابطأ من نفس النوع دون مخالفة الشروط. وقد تم تنفيذ تجارب على أنواع مختلفة من رسوم تدفعة المعلومات. ان أدا محاكاة التلدية أفضل من التسلسل العشوائى ولكن التسلسل العشوائى المعدل يودى أفضل من محاكاة التلدية فى بعض الحالات عند ادخال مرحلة ما بعد التنفيذ.

درجة ماجستير علوم
جامعة الملك فهد للبترول و المعادن
الظهران ، المملكة العربية السعودية

Chapter 1

Introduction

Design automation is the automatic synthesis of a physical design from some higher-level behavioral specification [1]. Automatic synthesis is much faster than manual design as it reduces the design cycle considerably. It also gives the designer a scope for experimentation. Synthesis is used for commercial implementations of systems which are widely used for production-level design of digital circuits. Synthesis can be divided into the following main categories.

- High-level synthesis converts a high-level program-like specification of the behavior of a circuit into a structural design in terms of an interconnected set of Register-Transfer (RT) level components, such as ALUs, registers and multiplexors.
- Logic synthesis converts a structural design, an interconnected set of RT level components, into optimized combinatorial logic, and maps that logic onto the library of available cells.

- Layout synthesis converts an interconnected set of cells, which describes the structure of a design, into the exact physical geometry (layout) of the design. It involves both, the placement of the cells as well as their connection (routing).

Figure 1.1 shows the relationship of high-level synthesis with the logic synthesis and layout synthesis in design automation.

High-level synthesis raises the level of abstraction to the algorithmic level, allowing a more behavioral style specification. Behavioral specification aims at describing only the functionality of a circuit representing the way the system or its components interact with the environment. Structure refers to the set of interconnected components that make up the system. Usually there are different structures that can be used to realize a given behavior.

The place of HLS in design hierarchy is illustrated in Figure 1.2. The level of detail increases from left to right in Figure 1.2. Also, five levels of abstraction shown in synthesis can take place, namely system, algorithmic, register-transfer, logic and circuit levels. Going from the algorithmic behavior to register-transfer level structure is HLS as shown in Figure 1.2. High-level synthesis bridges the gap between behavioral specifications and their hardware structure, automatically generating circuit descriptions that can be used for logic synthesis [2]. As opposed to logic synthesis which optimizes any combinational logic, high-level synthesis also deals with the memory elements, the interconnection structure (buses and multiplexors), and the sequential aspects of the design.

High-level synthesis is a formidable task. Although the main problems have been addressed, further optimizations involving area, cycle time, and sequential behav-

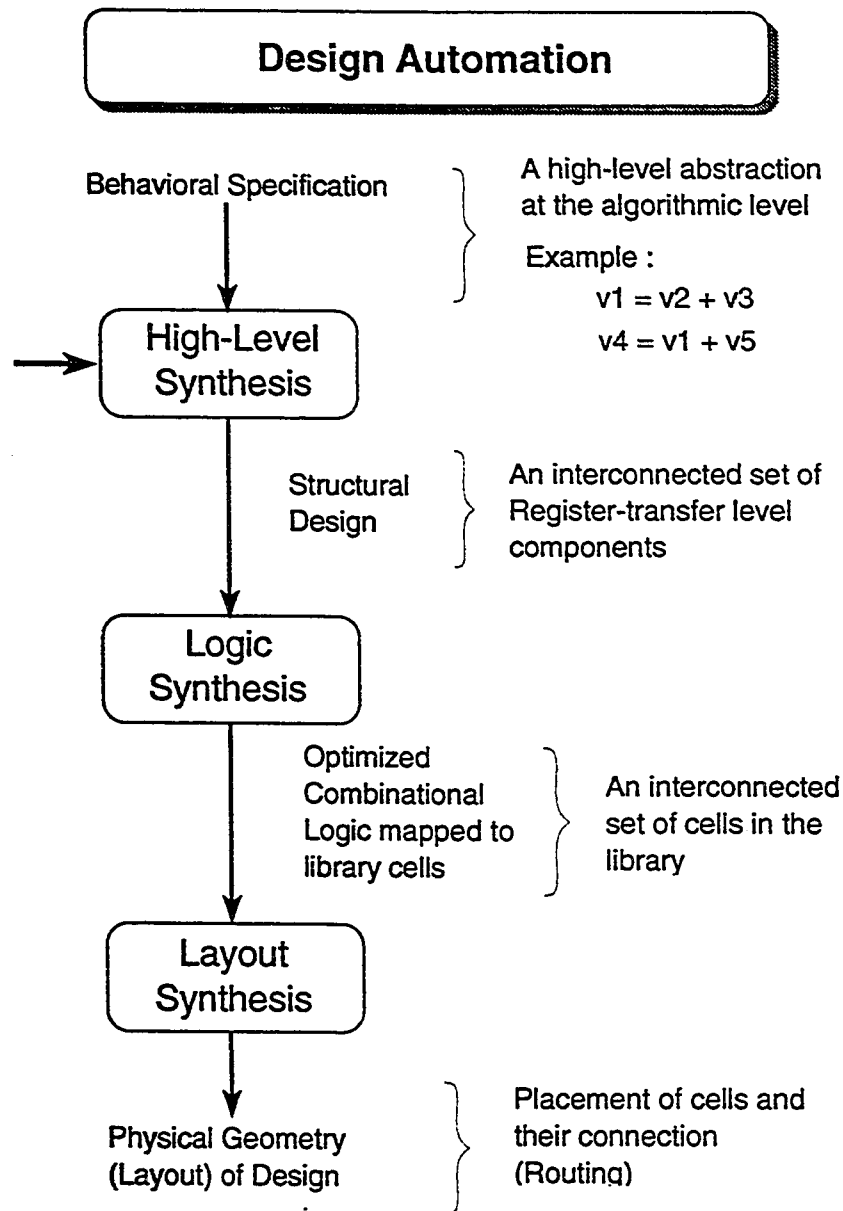


Figure 1.1: Relationship of HLS with logic and layout synthesis.

LEVEL	DOMAINS		
	Behavior	Structure	Physical
System	Communicating processes	Processors Memories Switches	Cabinets Cables
Algorithmic	Input-Output	Memory, Ports Processors	Board Floorplan
Register-Transfer	Register Transfers	ALUs, REGs, Muxes, Buses	ICs Macro Cells
Logic	Logic Equations	Gates Flip flops	Standard Cell Layout
Circuit	Network Equations	Transistors, Connections	Transistor Layout

Figure 1.2: HLS in design hierarchy.

ior are necessary to obtain high quality designs. It is also difficult to describe the hardware problem in terms of behavioral models. This makes it necessary to develop synthesis systems that allow specifications at various levels. There is a lack of effective high-level verification techniques that makes the application of high-level synthesis difficult. Since correctness by construction cannot be expected from a software system with several hundred lines of code, verification methods are necessary.

The main steps involved in high level synthesis are:

- Compilation of the HDL (High level description language is a sequential language similar to a programming language) source into an internal representation, usually a data flow graph and/or a control data flow graph.
- Transformations of the internal representation into a form more suitable for high-level synthesis. These transformations involve both compiler like and hardware-specific transformations.
- Scheduling, involves assigning each operation to a time step. It is also called *control synthesis* or *control step scheduling*.
- Allocation, which involves assigning each operation to a piece of hardware. It consists of selection of the type of hardware modules from a library and mapping each operation to the selected hardware. Allocation is also referred to as *data path synthesis* or *data path allocation*.
- Partitioning, which consists of dividing the design into smaller pieces. Partitioning can aim at obtaining a collection of concurrent hardware modules, or

can simply be used to generate smaller hardware pieces that may be easier to synthesize further.

- Output generation which produces the design is passed to logic synthesis and finite state machine synthesis.

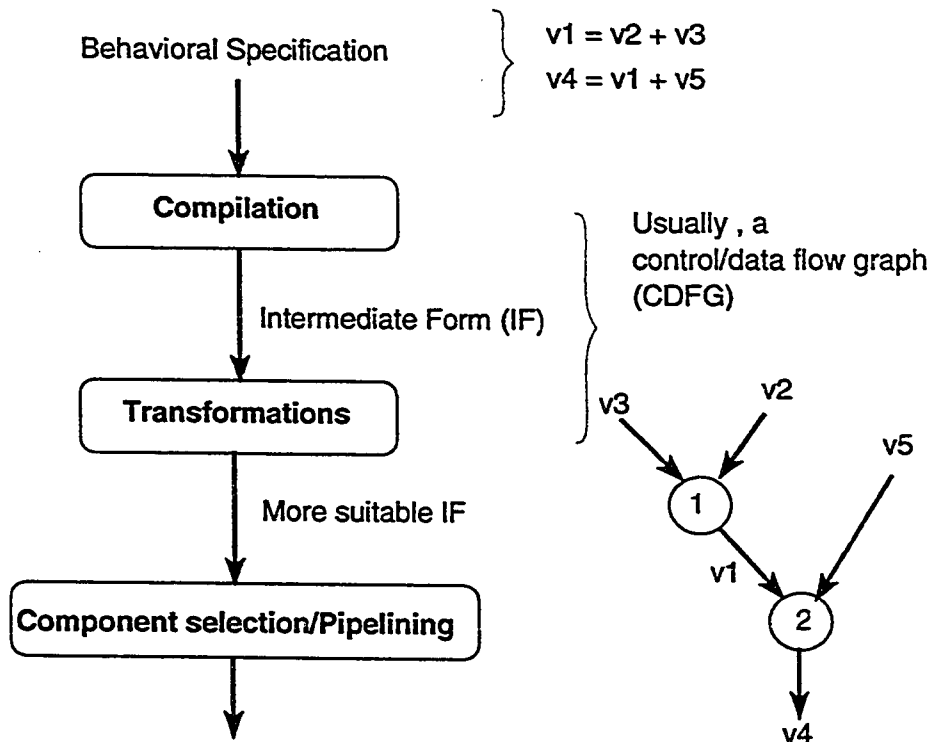


Figure 1.3: Steps involved in HLS.

The steps involved in high-level synthesis are illustrated in Figure 1.3. In order to extract the structure, the algorithmic specifications are first converted to an intermediate form such as Control/Data Flow Graphs (CDFGs), in which nodes correspond to operations (for example addition, multiplication, etc.,) and edges correspond to data values (for example variables and constants) and control flow dependencies.

Some compiler like high-level transformations (such as dead-code elimination, common subexpression elimination, etc.,) are applied to optimize the behavior of the design resulting in a more suitable intermediate form. This intermediate form is then used for allocation and pipelining.

Component selection and pipelining are some of the important steps in the synthesis of circuits from behavioral descriptions [3]. Component selection involves determining the best selection of components from a realistic library containing many different implementations per operator, and pipelining involves providing pipeline registers in the circuit such that the delay of each pipe stage is as close as possible to the given constraints [4].

1.1 Problem Definition

An important factor in obtaining cost effective designs is the ability to use multiple operator implementations in the datapath. Delay paths can then be balanced by using slow components where possible and using faster components only when necessary. For high performance applications it is necessary to combine pipelining with the use of a multiple-implementation library to satisfy performance requirements at a reasonable cost while satisfying certain constraints.

Given a data flow graph $DFG(V, E)$ where V represents a set of vertices, and $E \subseteq V * V$, a set of directed edges, a component library CL consisting of a set of three tuples (*Component_type*, *Area*, *Delay*), and constraints on pipe stage (*PS*) delay and latency, find an assignment of vertices to components and a partition so as to minimize the cost (given by the sum of the area of the datapath components).

The terms *PS delay*, *Assignment* and *Partition* are defined as follows:

- **PS delay:** Is the sample inter-arrival delay, that is, the delay between the arrival of two consecutive input samples. This is also the clock cycle of the design. Throughput, which is often the prime constraint on DSP systems, is the inverse of the PS delay.
- **Assignment:** If we associate a type (such as $*$, \div , $+$, $-$) called $Vertex_type(v)$, with every vertex v then an assignment is defined as a function from $V \rightarrow CL$ such that if $Assignment(v) = c$ then $Vertex_type(v) = Component_Type(c)$. This states that vertices can only be mapped to components of the same type.
- **Partition:** A *Partition* is a collection of subset of vertices, such that the union of all subsets is the complete vertex set V , and the intersection of any two subsets is the empty set.

Partitioning in the problem of component selection and pipelining refers to the pipeline stages in the DFG.

The DFG shown in Figure 1.4 along with the constraints is an example of the problem. The DFG consists of three multipliers and two adders. The PS delay constraint is 10 *ns* and Latency is 20 *ns*. The DFG consists of vertices which correspond to different operators depending on the circuit and directed edges that determine the interconnections between vertices and also the flow of operations between the input and output stages. The objective is to map the components from the multiple component library, shown in Table 1.1, onto the vertices to obtain a cost-optimal solution. Cost-optimal solution for this problem is obtained by minimizing the area

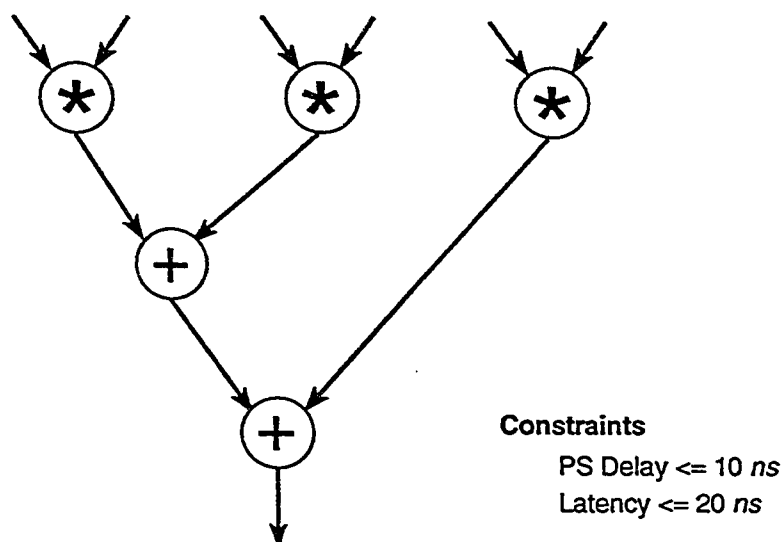


Figure 1.4: A sample data flow graph.

occupied in terms of the number of gates without violating the constraints and also pipelining the DFG. Pipelining is done by traversing the DFG from the inputs towards the output. When the sum of all the delays in the path exceeds the specified PS delay constraint then a register is placed. This is done for all the branches until the entire DFG satisfies the PS delay constraint.

The multiple component library shows different types of operators along with the area and delay of each of the operators. The availability of a large number of operators of each type is assumed.

A cost-optimal solution is obtained by an appropriate mapping of components from the component library onto the vertices of the DFG. Different types of iterative/constructive techniques may be used to obtain a cost-optimal solution. Figure 1.5 shows one possible solution. This consists of two multipliers of type *Mpy4*,

Component Type	Component Name	Area Gates	Delay ns
*	Mpy1	100	30
*	Mpy2	200	20
*	Mpy3	250	10
*	Mpy4	300	5
+	Add1	50	20
+	Add2	70	8
+	Add3	80	5
+	Add4	100	2
Register	Reg	50	

Table 1.1: Example of a component library.

one multiplier of type *Mpy3* and two adders of type *Add3* and *Add2* respectively. This solution requires three registers. The total cost is 1150 gates. Figure 1.6 shows another solution. In this figure three multipliers of type *Mpy3* and two adders of type *Add2* and *Add4* are used. This solution requires a total of four registers and it's total cost is 1120 gates. Both the solutions satisfy the PS delay and the latency constraints.

1.2 Literature Survey

The existing optimization techniques [5] can be classified into two types: iterative/constructive and global. Iterative/constructive techniques assign elements to vertices which represents the behavior of the circuit (operations, values or data transfers), one at a time, while global techniques find simultaneous solutions to a number of assignments at a time. More specifically, iterative/constructive techniques select

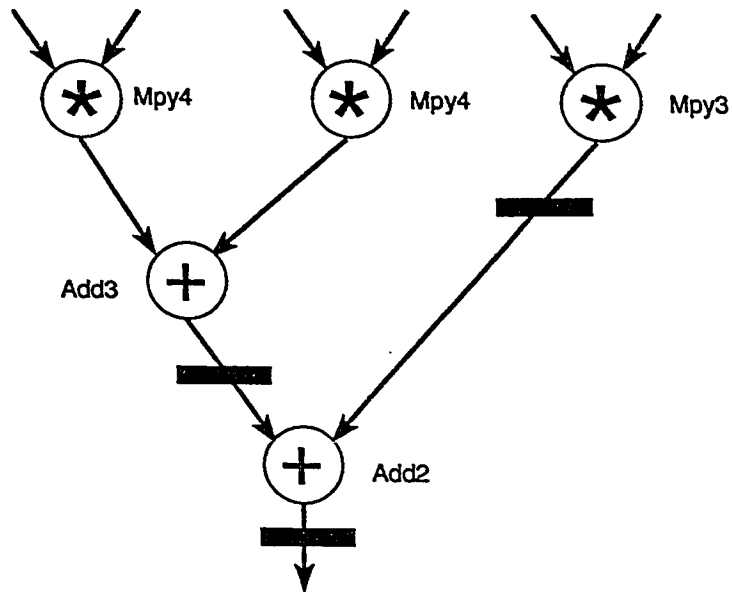


Figure 1.5: One possible solution to the example shown in Figure 1.4.

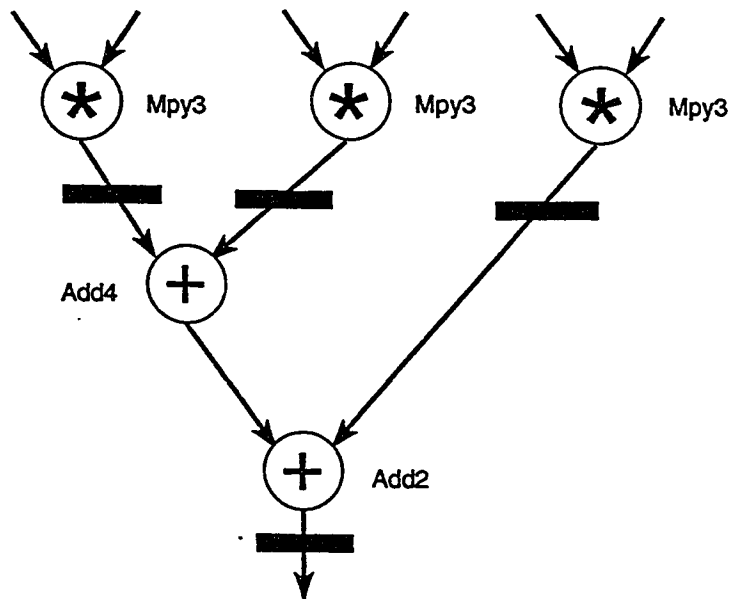


Figure 1.6: Another solution for the example shown in Figure 1.4.

an operation, value or interconnection to be assigned, make the assignment, and then iterate until all the assignments are made. These techniques generally look at a restricted window in the search space than global techniques, therefore are more time efficient, but are less likely to find optimal solutions. The optimization techniques are used in order to obtain cost-effective solutions within a limited time period. The size of the solution space for component selection and pipelining can be calculated as follows. If there are n number of vertices, m number of edges in the DFG, and k are the number of components in the component library, then the number of possible solutions are $k^n * 2^m$. A number of iterative/constructive techniques have been developed for solving problems having a large solution space. Some of these iterative techniques uses the hill-climbing approach. The hill climbing approach is to accept both solutions having higher and lower costs. Figure 1.7 shows the different states in the solution space. This figure shows a plot of cost versus states. The label L represents a local minima while G represents the global minima. The state G is reached by moving from the current state to the state at point G through a number of intermediate states. A new state is obtained by disturbing one or more components of the current state. It is seen that from the local minima L , we accept both costs that are higher than those that are present at L and those that are lesser than it. Finally we reach a point G , the global minima.

Examples of systems using iterative/constructive techniques are found in [6, 7]. Scheduling and allocation are two important phases in the synthesis of circuits from behavioral descriptions. Component selection and pipelining are also important steps in HLS. These are described in detail in the following sections.

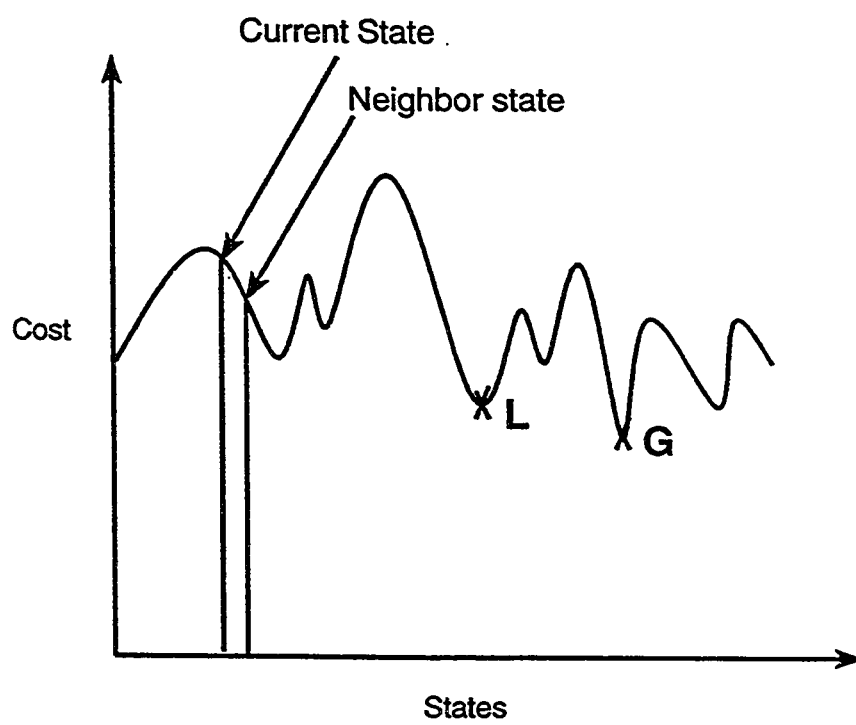


Figure 1.7: Different states of the solution space.

1.2.1 Allocation techniques

Allocation assigns each operation, variable and communication path to a piece of hardware. It naturally falls into three parts: Functional unit (FU) allocation, register allocation, and connection allocation. In high-level synthesis, the main aim in allocation is to share hardware units, i.e., operations can share functional units (ALUs, adders, etc.), variables can be mapped onto common registers and memories can share buses and multiplexors. The goal of allocation is to optimize the overall hardware.

A problem closely related to allocation is module assignment. Whenever there is more than one type of functional unit to perform a particular operation, that operation has to be assigned to one specific functional unit type. Such problems are solved using heuristic techniques. The different types of allocation algorithms that can be used are:

1. Heuristic allocation, e.g., greedy and sequential allocation.
2. Linear programming approaches.
3. Graph-based algorithms.

Heuristic approaches usually select one element (operation or variable) at a time to allocate and assign it to the hardware, with the selection done according to different criteria. Heuristic approaches yield reasonable results, are fast and have the potential to mix FU, register and communication path allocation.

Linear programming (LP) approaches formulate allocation (or allocation and scheduling) as a linear programming problem. In the past, however, since linear

programming required extensive computational resources, it could only be used for small examples. Recently, the method has reappeared, among other reasons because modern large LP systems can solve problems with tens of thousands of variables.

The third group of algorithms formulate allocation as clique covering (or partitioning) of a compatibility graph or node coloring.

Global allocation techniques include graph theoretic formulations, Branch and Bound algorithms and mathematical programming techniques. Trickey [8] used a graph theoretic approach in which the elements to be assigned to hardware, whether they are operations, values or interconnections, are represented by nodes, and there is an arc between two nodes if and only if the corresponding elements can share the same hardware. The problem then becomes one of finding sets of nodes in the graph, all of whose members are connected to one another, since all the elements in such a set can share the same hardware without conflict. An example of a system using Branch and Bound technique is SPLICER [9]. Formulations of allocation and component selection as a mathematical programming problem involves creating a variable for each possible assignment of an operation, variable or interconnection to a hardware element. The variable is one if the assignment is made and zero if it is not. Constraints must be formulated which guarantee that each operation must be assigned to one and only one hardware element, and so on. The objective is to find a valid solution that minimizes some cost function. STAR allocation system [10] uses an iterative improvement technique. It uses a rip-up reconstruct approach to the allocation problem. The data path is refined globally by evaluating the binding quality of each object, probabilistically selecting a cluster of heavily correlated objects (which may consist of variables, operations, and data transfers),

and rebinding them to form a better design or determine that there can be no more cost improvement.

Several optimization problems are computationally intractable, i.e., their decision versions are NP complete. There have been several deterministic heuristics suggested in the past for solving specific NP-complete problems. However most of these heuristics are essentially descent algorithms with respect to the cost function, hence they are unable to escape the local minima with respect to the underlying neighborhood structure. The simulated annealing (SA) algorithm introduced by Kirkpatrick *et al.*, [11] is an iterative stochastic procedure for solving combinatorial optimization problems. Proofs of convergence to a global minima and successful experimental implementations have led to the widespread use of SA and its acceptance as a viable general method for combinatorial optimization. However in general SA suffers from two major drawbacks. The first one being that an actual implementation of SA requires careful tuning of some of its parameters to achieve good results. The second one is that it uses excessive computation time and it's often less effective when compared with some well designed deterministic heuristics for the specific problem being solved.

1.2.2 Scheduling techniques

Scheduling assigns each operation in the behavior to a point in time. In synchronous systems, time is measured in control steps. Scheduling aims at optimizing the number of control steps needed for completion of a function, given certain limits on hardware resources and cycle time. A scheduling algorithm must take into account the

control constructs, such as loops and conditional branching, the data dependencies expressed in the data flow graph, and constraints on the hardware. In synchronous hardware the basic constraints are that every unit of the hardware can be used only once during the control step, i.e., registers can be loaded only once, combinational logic may evaluate once (feedback is forbidden) and buses may carry only a single value. Other constraints on a design may restrict the size, the delay, and the power.

The first approach to scheduling in high-level synthesis was probably the exhaustive search. Since then, many scheduling algorithms have been proposed for high-level synthesis have been proposed, some relying on methods known from microprogram optimization. Davidson et. al., [12] discuss an exhaustive search using branch-and-bound techniques, as-soon-as-possible (ASAP) scheduling and scheduling the critical path first.

In an ASAP schedule, all operations are assigned to the earliest possible control step, corresponding to a topological sort of the graph in the depth-first order. An example of CDFG with its ASAP schedule under the constraint of one adder and one multiplier is shown in Figure 1.8 [5].

List Scheduling schedules operations into control steps, one control step at a time. For the current control step, a list of data ready operators is constructed, containing those operators whose inputs are produced in earlier control steps and that do not violate any resource constraints. This list is then sorted according to some *priority function*, the highest-priority operator is placed into current control step, the list is updated, and the process continues until no more operators can be placed into the control step. This process is then repeated on the next control step, until the entire design is scheduled. List scheduling is illustrated in Figure 1.9 [5].

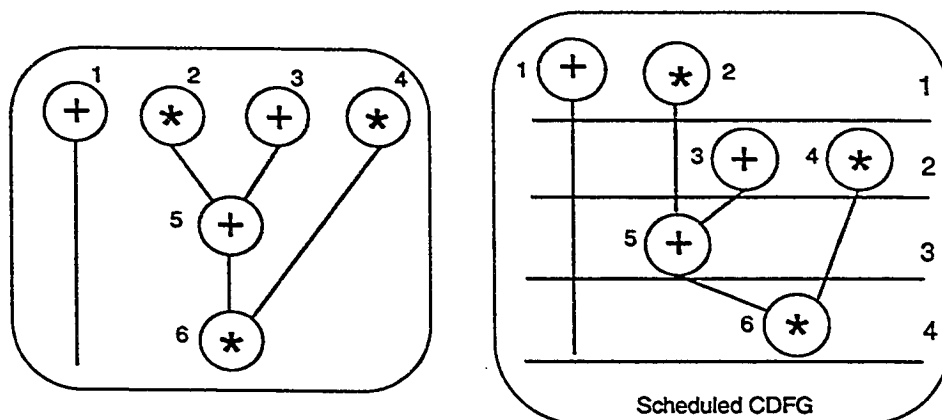


Figure 1.8: Example of ASAP scheduling [4].

The priority function is the path length from the node to the end of the block which is shown in Figure 1.9a and the list schedule is shown in Figure 1.9b. Variations of list scheduling are used in many high-level synthesis systems, for example, CMU's System Architect's Workbench [13].

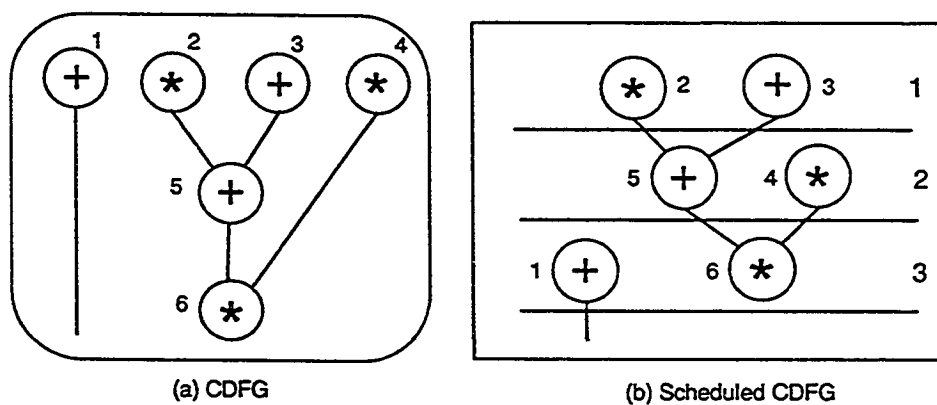


Figure 1.9: Example of list scheduling [4].

A more complex scheduling method is *force - directed scheduling* [14], which uses a global criterion that indicates how crowded a control step is compared with

others to decide where to schedule an operation. The probability of operations being in a given control step can be calculated by using *mobility* which is the difference between the ASAP and the as-late-as-possible (ALAP) schedules. For example, if an operation can be scheduled in three steps, it has a mobility of 3. Thus, the probability of the operation to be scheduled in each of these steps is $1/3$. Adding all the probabilities of any control step gives a measure of how crowded that control step is. This measure is called the *distribution* because it tells how much hardware will be required. After determining the distribution for all control steps, the effect for each possible assignment of an operation v to a control step s can be calculated. Then the operation/control-step pair can be scheduled to minimize the distribution differences among control steps. The quantitative measure of scheduling v in s is calculated on the distribution using an equation that is analogous to the force in a spring (spring constant times displacement, where the constant is the original distribution for a control step and the displacement is the change in the distribution value). Thus, this scheduling algorithm is called forced-directed scheduling (FDS). FDS is illustrated in Figure 1.10 [5]. A CDFG with three add operations labeled as $a1$, $a2$ and $a3$ is shown in Figure 1.10a. Figure 1.10b shows time frames for add operations, that is the probability of each operation being in a given control step. Distribution is shown in Figure 1.10c. Calculation of force involved in assigning $a3$ to control step 2 is shown in Figure 1.10. As we see in Figure 1.10c the control step 2 is heavily loaded, and thus the positive force indicates that $a3$ should not go into control step 2.

All of the above scheduling techniques, except exhaustive search and branch and bound techniques come under iterative/constructive techniques. Another approach

to scheduling by transformation is to use heuristics to guide the process. Starting with an initial schedule, transformations that promise to move the design closer to the given constraints or to optimize the objective are chosen. In the former, we first assign each operation to a separate control step and then merge control steps iteratively without violating any constraints. In the latter, we first assign all operations to a single control step and then divide this control step until we have no constraint violations.

1.2.3 Pipelining

Pipelining is a common technique to enhance the circuit performance. In a pipelined implementation, the circuit is divided into stages. Each stage executes concurrently and feeds its results to the following stage.

Pipelining has been applied to instruction set to support efficient execution of different instruction streams and to signal/image processors. Conversely pipelined digital signal processing (DSP) design may be simpler, because often the processor is dedicated to an application.

Few techniques for synthesizing pipelined data paths have been proposed under some limiting assumptions such as constant data rates. Unfortunately efficient instruction set processor design requires handling variable data rates as well as a variety of other issues, such as stage bypasses, hazard analysis and support for controlling the pipeline by allowing stalling and flushing. As a result, present synthesis techniques are not yet applicable to the design of competitive instruction set processors, although they have been applied successfully to some DSP designs. To achieve

pipelining, the input task (process) must be subdivided into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. Consecutive tasks are initiated at an interval which is the integer multiple of a clock cycle and is shorter than pipeline latency. Some *ad hoc* representation paradigms have been developed for pipelined circuits. For example, pipelined circuits can be specified by modeling each stage independently as well as the stage interfaces and the synchronization mechanisms. This corresponds to providing a mixed structural/behavioral circuit model that may preclude architectural optimization of the circuit as a whole. In particular, the number of stages is prescribed. Circuits are modeled by pipelining sequencing graphs where the source vertex is fixed at the throughput rate. The time interval between two successive inputs is called the data introduction interval. Generally the data introduction interval is smaller than latency. If the data introduction interval equals latency, then the operations are performed without using any pipeline registers. If the latency is greater than the data introduction interval then pipelining can be introduced at different intervals in the circuit, so that delay between consecutive pipeline registers is less than or equal to the data introduction interval.

The design objectives of pipelined circuits are four: *area*, *latency*, *cycle – time* and *throughput*. Architectural exploration and optimization relate to determining the tradeoff points in the corresponding four-dimensional space. Most approaches consider the sequence of problems derived by choosing particular values of the data introduction interval and of the cycle-time (hence determining implicitly the throughput) and a search for optimal (area, latency) tradeoff points. Several optimization techniques can be used to obtain a cost-optimal solution. Some of them are

stochastic evolution, simulated annealing, genetic algorithm and integer programming.

A scheduling and hardware sharing (allocation) algorithm for synthesizing both pipelined and non-pipelined data paths is presented in [15]. In this the scheduling algorithm tries to distribute operations equally among partitions to maximize hardware sharing. Multiplexer delays are explicitly considered to produce a more accurate scheduling. In hardware sharing the structural parameters such as the size of the multiplexers, interconnect overhead, the size of smallest sharable operator etc., are employed to control the amount of sharing globally and produce a heuristically optimized RTL structure. The scheduling algorithm is iterated until a satisfactory structure is obtained. This algorithm could also be used for synthesizing pipelined data path from a graphics process description that contains about 1000 components.

Jun and Hwang [16] describe the SODAS-DSP system, a pipelined datapath synthesis system targeted for application-specific DSP chip design. Through facilitated user interaction, the design space of pipelined datapaths for given design descriptions are explored to produce an optimal design. Taking a signal flow graph as the input SODAS-DSP generates pipelined datapaths through scheduling and module allocation processes. New scheduling and module allocation algorithms have been proposed for efficient synthesis of pipelined hardware. The proposed scheduling algorithm is of iterative/constructive nature, where the measure of equidistribution of operations among pipeline partitions is adopted as the objective function. This DSP system generates efficient pipelined datapaths compared to Sehwa [17].

A method for pipelining VLSI/ULSI systems for effective communication is proposed in [18]. Propagation delays of data signals have been known to severely impair

performance of VLSI/ULSI interconnection networks. A simple but effective way has been proposed to increase the performance. The basic idea of the technique relies on the fragmentation of the wires and in reconnecting them with a special device called a repeater in order to form a bidirectional pipeline. By employing this technique the transmission speed is improved by 150% for 32-byte messages when a 10 cm 8-bit bus is used. It is seen that the improvement increases for longer messages and for larger skews.

1.3 Solution techniques

Iterative and constructive techniques have been used to solve a wide range of problems. The iterative techniques such as tabu search, simulated annealing, simulated evolution, genetic algorithm, stochastic evolution etc., have been used.

1.3.1 Iterative techniques

Amellal and Kamenska [19] have proposed one such iterative technique that describes a tabu search synthesis system for functional synthesis. The functional synthesis of a digital system is the realization of a register-transfer level description from the functional specification of the system. Synthesizing a digital system from a functional description is a complex process requiring the solution of various different problems. A control and data flow graph for representation is developed and this model generates a single graph representing both the data and the control flow of a VHDL behavioral description. The use of conditional dependency edges in the graph gives a better implementation of the control constructs. A new mathematical

formulation of the scheduling problem using a approach based on penalty weights is developed. Penalty weights include the real costs of the hardware units available in a given technology. The penalty weights take into consideration different area parameters of the design to be generated. The number of functional and storage units as well as the number of interconnections is optimized by the minimization of an objective function including penalty weights.

Sait et. al., [20, 21] solve the problem of scheduling and allocation using two iterative techniques namely genetic algorithm (GA) and tabu search. The problems of allocation and scheduling are formulated as an optimization problem. A new chromosomal representation for scheduling and allocation is proposed using GA technique. Apart from this, two new crossover operators to generate legal schedules has been developed. The problem of scheduling and allocation has been implemented using the tabu search technique. Using this technique a good initial solution, a neighborhood generation strategy, formulation and maintenance of tabu lists, a proper aspiration level criteria and a good tabu list size has been developed.

The related research based on pipelining and component selection can be categorized into two classes. The first class consists of tools like Sehwa [17], the tools from the GS Corporation R&D laboratories [15], and PLS a pipelined scheduler [22]. These tools pipeline a given DFG so as to optimize area and performance for given constraints, usually on the throughput or latency of the design. However they all assume a single implementation for functional units which force them to use the same component on non-critical and critical paths, resulting in designs that are inefficient and costly. SLIMOS [23] and MOSP [24] differ slightly from the above approach - they start from a multiple implementation library and then select one

single implementation per operator. Hence their final design also contains single implementations, leading to the same design inefficiencies.

The second category contains algorithms such as Tabu search (TBS) [4] and the module selection algorithm in [25]. Though these tools use unrestricted libraries that allow multiple physical implementations for the same operator, they combine component selection with non-pipeline scheduling, rather than pipelined scheduling.

Stochastic evolution (SE) is an iterative technique that has been applied to problems of certain complexities like network bisection and traveling salesman problems. We wanted to investigate the application of SE to problems whose complexities are similar to those of component selection and pipelining. This problem was also solved by simulated annealing (SA) and the two techniques were compared. The reason for comparing SE with SA is due to the similarities between the two techniques. Both the techniques are stochastic in nature, both accept downhill moves and they have control parameters that govern the probability of accepting the uphill moves. The basic difference between these two techniques is in the determination of the range of magnitude of the negative gains and in their method of acceptance.

1.3.2 Constructive techniques

Constructive techniques search a small part of the solution space. The heuristics employed by these techniques generally do not produce an optimal solution, but these produce effective results within a short duration. Many constructive techniques have been implemented in HLS systems. Sait et. al., [26] describes a loop-based scheduling algorithm. In this algorithm a subset of some high-level programming

language is used to describe the behavior of the intended design. An intermediate form is generated by using the programming language compiler in the transformation process. The use of the compiler results in optimization and avoids restricting the language to certain data types or control constructs. In order to eliminate machine dependency and complexity the intermediate form is converted into another form which is machine independent and has simpler syntax and semantics, called Pseudo Assembly language (PAL). PAL descriptions are used by the system components to produce the intended hardware in an RTL description language. Scheduling in this HLS system is done in a constructive manner using Loop Based Scheduling (LBS). In LBS the control flow graph is partitioned into subgraphs, then each of the subgraphs are scheduled individually and finally the individual schedules of all the subgraphs are combined.

A constructive heuristic has been used to solve the problem of component selection and pipelining [3]. The objective is to maximize the use of slow components and minimize the use of faster components while satisfying the constraints of PS delay and latency. The components are selected from a realistic component library that contains multiple implementations of operators. The key to the constructive heuristic lies in judiciously selecting vertices to be slowed down in each iteration, since slowing down one vertex may prevent slowing down others due to graph dependencies. Thus, the desirability of slowing down a vertex is evaluated with respect to all the vertices that would be affected by its slow down. With every vertex is associated a value, called *vertex weight* which gives a measure of its desirability or priority in the selection process. The vertex with the highest weight is one that is selected to slow down. This heuristic gives cost-effective results within a small

period of time.

1.4 Summary

The chapter described the various iterative techniques for solving combinatorial optimization problems in HLS and their role in design automation. The problem of component selection is one such problem in HLS. The following chapter describes the SE and SA techniques. The solution strategies and the algorithms of SE and SA that were used in solving the problem of component selection and pipelining are described in chapters 3 and 4 respectively. The experimental results are presented in chapter 5 and the conclusions and future work are presented in chapter 6.

-

Chapter 2

Stochastic Evolution and Simulated Annealing

Stochastic Evolution (SE) and Simulated Annealing (SA) are two techniques that can be used to solve a wide range of combinatorial optimization problems. The SA algorithm introduced by Kirkpatrick et al. [11] is an iterative improvement technique for solving combinatorial optimization problems. SE resolves the two main drawbacks of SA. They are:

- An actual implementation of SA requires a careful tuning of some of its control parameters to achieve good results and it uses excessive computation time.
- It does not have a suitable stopping criteria [27].

An important factor in obtaining cost-effective designs is the ability to use multiple operator implementations in the data path. Delay paths can then be balanced by using slow components where possible and the faster components only when neces-

sary. For high performance applications such as Digital signal processing (DSP) systems, designers often combine pipelining with the use of a multiple-implementation library so as to satisfy performance requirements at a reasonable cost. In this research SA and SE techniques were used to solve the problems of component selection and pipelining of a DFG to give a cost-effective solution. This chapter gives an introduction to these two techniques.

2.1 Stochastic Evolution

The inputs for component selection and pipelining are a set of constraints, a component library, and a DFG. The output is a valid pipelined DFG with components assigned to the nodes and the constraints being satisfied. The relationship of these elements with the stochastic system is shown in Figure 2.1. The stochastic system in our problem is the SE or the SA algorithm. Stochastic evolution is based on the concept of state model [27]. A state model is described as a finite set M of movable elements, a finite set L of locations and the state is defined as $M \rightarrow L$ satisfying certain constraints. The idea behind the SE algorithm is that the suitability of each movable element $m \in M$ in its current location $S(m)$ leads to a lower cost of the state S . SE algorithm is a special instance of a more general class of adaptive heuristics by Nahar et. al. [28].

2.1.1 Algorithm

The input to SE is an initial state S_0 , an initial value of the control parameter p_0 , and parameter R used in the stopping criterion. The initial state S_0 is a valid

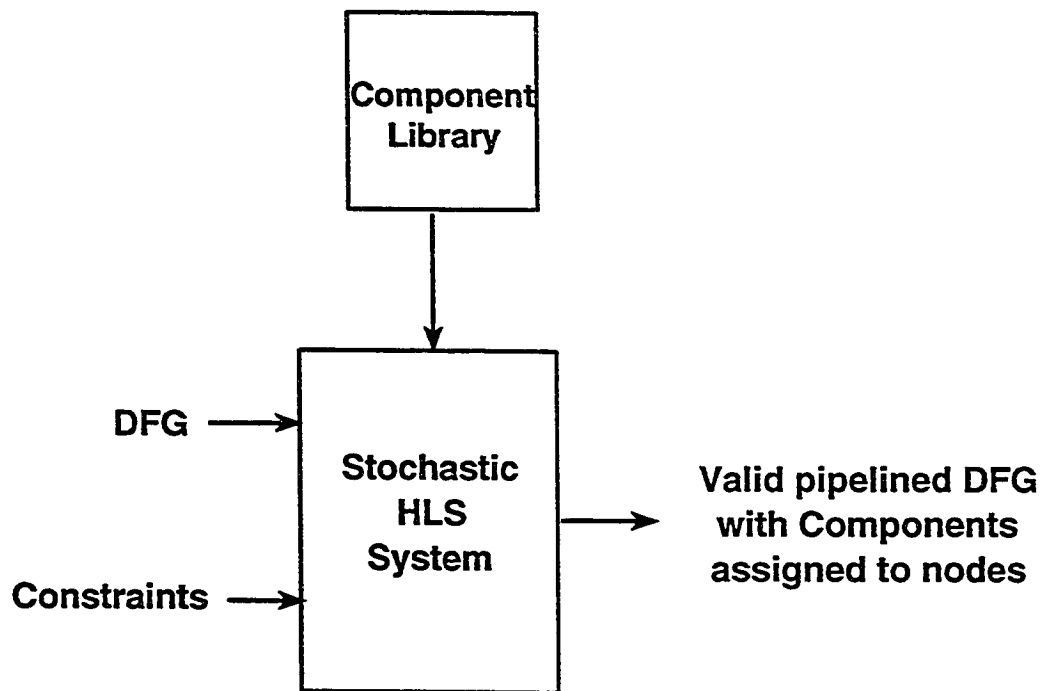


Figure 2.1: HLS system.

ALGORITHM SE

```

 $S = S_0$                                 /* initial state */
 $S_{Best} = S$                             /* save initial state */
 $p = p_0$                                 /* initialize control parameter */
 $\rho = 0$                                 /* initialize counter */
REPEAT
     $C_{pre} = COST(S)$ 
     $S = PERTURB(S, p)$ 
     $C_{cur} = COST(S)$ 
    UPDATE( $p, C_{pre}, C_{cur}$ )
    IF ( $COST(S) < COST(S_{Best})$ ) THEN
         $S_{Best} = S$                     /* save best state */
         $\rho = \rho - R$                   /* decrement counter by R */
    ELSE
         $\rho = \rho + 1$                 /* increment counter */
    ENDIF
UNTIL  $\rho > R$                           /* stopping criteria */
RETURN ( $S_{Best}$ )                        /* report best state */

```

Figure 2.2: Stochastic Evolution algorithm.

state satisfying all the constraints specified by the problem under consideration. The initial state is assumed to be the best state on invocation of the algorithm. This algorithm is shown in Figure 2.2. The cost function depends on the type of the problem being solved. In a network bisection problem, cost is the number of hyperedges cut, where as in a traveling salesman problem, it is the sum of the distances covering all the cities.

The SE algorithm retains the state of lowest cost among those produced by the function PERTURB. Each time a state is found which has a lower cost than the best state so far, SE decrements the counter by R . The UPDATE procedure is used to update the control parameter p .

The iteration bound R acts as the expected number of iterations the SE algorithm needs until $C_{cur} < COST(S_{best})$, i.e., an improvement in cost takes place. If such an improvement occurs at $p < R$ iterations, then the remaining $R - p$ iterations are added to the next R iterations to be performed. Therefore, if I is the total number of iterations performed by the algorithm, then I/R is the number of improvements encountered. Consequently the quality of the final state obtained increases with the running time of the SE algorithm. If R is set too large, then SE algorithm wastes time during the last set of iterations because it cannot find better states. However if R is chosen too small, the SE algorithm might not have enough time to improve the initial state.

2.1.2 PERTURB

During each call to *PERTURB* the elements are scanned in a particular order. The choice of this ordering is problem specific. The *PERTURB* function is shown in Figure 2.3.

Let $S : M \rightarrow L$ be the existing function that may or may not satisfy the constraints of a state and assume that the cost function has been defined. Let element $m \in M$. If a unique move is associated with m from S then it generates a new function $S' : M \rightarrow L$ such that $S'(m) \neq S(m)$. The move associated with m could itself be a simple move or a compound move depending upon the problem being solved. The gain is calculated as $GAIN(m) = COST(S) - COST(S')$ which gives the change in costs after the move is performed. The function *PERTURB* stochastically decides whether or not to accept the move associated with the element m

```

Function PERTURB(S, p)

FOR EACH (m ∈ M) DO                                /* m:vertex of the graph */
S' = MOVE(S, M)                                  /* perform move associated with m */
  GAIN(m) = COST(S) - COST(S')                  /* compute gain of move */
  IF (GAIN(m) > RANDINT(−p, 0)) THEN
    S = S'                                           /* accept move */
  ENDIF
ENDFOR
S = MAKESTATE(S)                                   /* make sure S is a state */
RETURN (S)                                           /* return a neighboring state */

```

Figure 2.3: PERTURB Function.

being scanned with the help of a non-negative control parameter p .

The value of $GAIN(m)$ is compared to an integer r randomly generated in the interval $[-p, 0]$. If $GAIN(m) > r$, then the move to S' is accepted, otherwise it is rejected. If $r \leq 0$, moves with positive gains are always accepted. The algorithm then goes on to scan the next element in M . After scanning all the elements of M , the final state is accepted if it satisfies all the constraints. If the state S does not satisfy the constraints then the latest moves are retraced backwards until a state is obtained which satisfies the constraints.

2.1.3 UPDATE

The *UPDATE* function shown in Figure 2.4 is mainly responsible for updating the value of the control parameter p . p is used to determine the range of the negative gains that are to be accepted. This has to be selected carefully. Initially p is set to a non-negative value close to zero. Such a choice for p means that only moves

```

Procedure UPDATE( $p$ ,  $C_{pre}$ ,  $C_{cur}$ )

  IF ( $C_{pre} = C_{cur}$ ) THEN
     $p = f(p)$ 
  ELSE
     $p = p_0$ 
  ENDIF

```

Figure 2.4: UPDATE procedure.

with small negative gains are accepted. The value of p is increased only when costs for two consecutive iterations are the same. If both the costs are same, then p is increased to a new value $f(p) \geq p$. $f(p)$ is obtained by increasing p by a certain positive value. The value that is added to p depends on the type of the problem being solved. The value is chosen such that a cost effective solution is obtained. Otherwise p is reset to its initial value. The parameter p is increased to give the algorithm a chance to escape a local minimum via an uphill climb. Depending on the problem more than one control parameter may be used.

2.1.4 Issues Concerning Stochastic Evolution Algorithm

The successful implementation of SE to achieve cost-effective results is based on four factors.

1. An appropriate modeling of the state of the problem.
2. The notion of the move to be associated with the “movable” elements of the state has to be carefully designed.

3. An initial value of the control parameter p and a method for updating it has to be devised.
4. A value for the stopping criterion parameter R .

The above issues are interrelated and depend on the problem as well. The basic requirement is a good representation of the given problem. This enables in designing effective and efficient strategies. The move strategy inturn depends on the problem. The choice of parameter p and it's update method depends on the moves adopted. The choice of stopping criteria depends on the second, third and the fourth choices. In the state model, if the moves and the control parameter p are appropriately chosen then a near optimal solution could be obtained.

2.2 Simulated Annealing

Simulated annealing is a technique for solving combinatorial optimization problems [11, 28, 29]. It is not an algorithm with a prescribed sequence of operations to solve a problem but a paradigm for constructing algorithms to solve optimization problems of a particular character. It belongs to a class of iterative improvement schemes. It has been applied to several combinatorial optimization problems from various fields like traveling salesman problem, graph partitioning, quadratic assignment, matching, linear arrangement and scheduling [30]. Resource constraint problems [31] have also been solved using SA. In the areas of engineering, simulated annealing has been applied to VLSI design [1] (placement [32, 33], routing [34]), image processing, code design, facilities layout, network topology design etc.

2.2.1 Background

Simulated annealing was derived using an analogy between the physical annealing process of the solids and combinatorial problems. The term annealing refers to heating a solid to a very high temperature and then slowly cooling the molten material in a controlled manner until it crystallizes [1, 35].

A combinatorial optimization problem is one in which we seek to find some configuration of parameters $\bar{X} = (X_1, X_2, X_3, \dots, X_n)$ that minimizes some function $f(\bar{X})$. This function is usually referred to as the *cost* or *objective* function. The objective function is a measure of goodness of a particular configuration of parameters. Realistic design problems may require many parameters and a complex cost function. Consider for example, deciding the placement of components on a surface of an integrated circuit in an optimal way. We may seek to maximize the ability to route wires to interconnect these components [34, 36], minimize the overall chip area, minimize the manufacturing yield of the chip, minimize the deviation from specified timing constraints and so forth. The cost function for such a problem may be very sophisticated with a large number of parameters.

Iterative strategies attempt to perturb some existing suboptimal solution in the direction of a better, lower cost solution. An obvious approach is to explore easily reachable neighboring configurations and to select the one with the least cost, i.e., the one giving the most improvement. In practice the current solution is randomly perturbed. This process is continued until no further improvements are obtained, at which point the process terminates. In such iterative techniques improvement is only downhill and the solution gets stuck in a local minima [35]. In order to overcome this,

random initial configurations can be tried, improving each one of them and using the best answer. However, for very large problems, the computational expense is high, the number of random starts needed to adequately sample the cost surface is unreasonable, and still there is no guarantee of finding a good solution.

Simulated annealing offers a strategy very similar to iterative improvement, with one major difference that annealing allows perturbations to move uphill in a controlled fashion. Because each move can transform one configuration into a worse configuration, it is possible to jump out of a local minima and potentially fall into a downhill path. Designing an annealing algorithm for a problem consists of five major parts:

1. **Configuration Space:** The set of allowed configurations of the system must facilitate easy representation of each state and easy generation of perturbations.
2. **Move Set:** The set of feasible moves (eg., pair swaps) must be rich enough so that all reasonable solutions can be found by applying a sequence of moves from this set. In addition, these moves must be relatively inexpensive to compute, since a large number of moves will be used.
3. **Cost Metric:** The metric must be incrementally computable so that the time to evaluate each move is minimal.
4. **Annealing Schedule:** The manner in which the temperature T is lowered during annealing, also known as the temperature schedule is crucial. Starting too cold, stopping too hot, or cooling too quickly all produce suboptimal

Algorithm `Simulated_annealing($S_0, T_0, \alpha, \beta, M, Maxtime$)`

```

/*  $S_0$  is the initial solution */
/*  $T_0$  is the initial temperature */
/*  $\alpha$  is the cooling rate (a constant) */
/*  $\beta$  a constant */
/*  $M$  represents the time until the next parameter is updated */
/*  $Maxtime$  is the total allowed time for the annealing process */

```

```

BEGIN
   $T = T_0$ 
   $S_c = S_0$ 
  Time = 0
  REPEAT
    Call Metropolis( $S_c, T, M$ )
    Time = Time + M
     $T = \alpha * T$ 
     $M = \beta * M$ 
  UNTIL ( $Time \geq Maxtime$ )
  Output best solution
END

```

Figure 2.5: Simulated Annealing algorithm.

solutions. Starting too hot or cooling too slowly wastes CPU time [1].

5. **Data Structures:** The ability to propose and evaluate moves efficiently hinges on a good representation for the basic objects in the problem.

Although the simulated annealing framework is conceptually straightforward, design of a successful annealing-based algorithm involves considerable engineering judgement in the process of designing the five components described.

2.3 Algorithm

The core of the simulated annealing algorithm shown in Figure 2.5 is the *Metropolis* procedure, which simulates the annealing process at a given temperature T [37]. The procedure *Metropolis* is named after the scientist who devised a similar scheme to simulate a collection of atoms in equilibrium at a given temperature.

Simulated annealing procedure starts with a initial solution S_0 , initial temperature T_0 , cooling rate α , a constant β which controls the time spent in annealing at a particular temperature, and *Maxtime* which is the total time allowed for the annealing process and M that represents the time until the next parameter is updated.

The *Metropolis* procedure shown in Figure 2.6 receives as input the current temperature T , and the current solution S_c which it improves through local search. *Metropolis* is also provided with the value M , which is the amount of time for which annealing must be applied at a temperature T . The procedure simulated annealing simply invokes *Metropolis* at decreasing temperatures. Temperature is initialized to a value T_0 at the beginning of the procedure, and is slowly reduced in a geometric progression; the parameter α is used to achieve cooling. The amount of time spent in annealing at a temperature is gradually increased as temperature is lowered [11]. This is done using the parameter $\beta > 1$. The variable *Time* keeps track of the time being expended in each call to the *Metropolis*. The annealing procedure halts when *Time* exceeds the allowed time.

In simulated annealing the current state is disturbed to obtain a neighboring state. The neighboring state may be obtained by performing a random perturbation,

Algorithm Metropolis(S_c, T, M)

```

BEGIN
  REPEAT
     $S_n = \text{neighbor}(S_c)$ 
     $\Delta c = (\text{cost}(S_n) - \text{cost}(S_c))$ 
    IF  $((\Delta c < 0) \text{ or } (\text{random} < e^{-\Delta c/T}))$  THEN
       $S_c = S_n$ ; /* accept the solution */
    M = M - 1
  UNTIL (M = 0)
END

```

Figure 2.6: Metropolis Procedure.

such as moving a component or part to a new location or replacing a component or a part by another one. After obtaining a new state the *cost* of the state is computed. If the cost of the new solution S_n is better then the cost of the current solution S_c , then the new solution is accepted and S_c is set to S_n . If the new solution has a higher cost in comparison to the original solution S_c , *Metropolis* will accept the new solution on a probabilistic basis. This is to accept uphill moves. At higher temperatures the probability of large uphill moves is high and at lower temperatures the probability is small. If this random number is smaller than $e^{-\Delta c/T}$, where Δc is the difference in costs, ($\Delta c = c(S_n) - c(S_c)$), and T is the temperature, the uphill solution is accepted. The probability that an inferior solution is accepted is given by $P(\text{random} < e^{-\Delta c/T})$. The random number generation is assumed to follow a uniform distribution. At very high temperatures, (when $T \rightarrow \infty$), $e^{-\Delta c/T} = 1$, and hence the above probability approaches 1. When $T \rightarrow 0$, the probability $e^{-\Delta c/T}$ falls to zero [1].

2.4 Problem-specific decisions

The problem-specific decisions are concerned with the neighborhood structure and the cost function. These have a significant effect on the success of an annealing algorithm. As with generic decisions, it is not always possible to set down a series of rules which will define the best choices for a given problem. However, it is possible to outline some properties which are desirable. In making decisions about these factors, two important objectives have to be considered.

- The validity of the algorithm is to be maintained.
- The computation time should be used effectively for as many iterations as possible.

If the available computing time is to be used efficiently, it is vital that frequently used routines should be as fast as possible. Generations of the neighborhood solutions must be done in an efficient manner such that it does not consume much time. It is sometimes complex if the neighborhood is large or if the solution space is constrained by stringent feasibility conditions. As the cost function has to be calculated between two states after every iteration, it is important that the cost function and the neighborhood structure be chosen in such a way that this calculation can be carried out quickly and efficiently. It is often the case that it does not necessitate recalculation of the complete cost function for the new solution, and such shortcuts should be considered when deciding on the forms of the costs and the neighborhood.

If the number of iterations are to be kept reasonably low, it is necessary to avoid neighborhoods which give rise to a spiky topography over the solution space. The

number of iterations that have to be carried out to obtain an optimal solution also depends on the size of the solution space. If the solution space is smaller, optimal solution can be obtained in lesser number of iterations. If the solution space is large, it is necessary to reduce the size of the solution space by eliminating part of the solution space based on certain criteria. This in turn is specific to a given problem. In addition to keeping the solution space small, it is also useful to aim for reasonably small neighborhoods. This enables a neighborhood to be searched adequately in fewer iterations, but conversely means that there is less opportunity for dramatic improvements to occur in a single move. But all the conditions cannot be satisfied for a given problem, therefore compromises have to be made.

2.5 Summary

This chapter described the SE and the SA techniques. The various strategies that were used to perturb the states have been described. The acceptance criteria for both the techniques has also been discussed. The next chapter describes the application of SE technique for component selection and pipelining.

Chapter 3

Component Selection and Pipelining using Stochastic Evolution

This chapter describes the application of Stochastic Evolution (SE) technique for component selection and pipelining. The inputs for this problem are a Data Flow graph (DFG), a multiple-component library and constraints. A DFG consists of vertices that represents operators and edges that show the interrelationship between the vertices. Each DFG consists of a set of inputs and a set of outputs. Mapping components from a realistic component library (CL) on to the vertices of the DFG is component selection. Pipelining involves placing registers on the DFG such that *PS* delay constraint is satisfied, i.e., delay between any two consecutive registers in the DFG should not exceed the specified *PS* delay. To design the problem of component selection and pipelining using the SE technique the following issues have

to be addressed:

- Designing an initial representation of the solution.
- Determining the cost function.
- Designing appropriate perturb strategies.
- Defining appropriate acceptance criteria.

A careful design of these critical parts results in obtaining a cost-optimal solution. This chapter describes the various design issues that were considered and the effect of each one of them.

3.1 Initial Solution Representation

A $DFG(V, E)$ consists of a set of vertices V , and a set of directed edges E . Each of the vertices are connected to one or more vertices through edges. In order to keep track of the edges that are coming into a particular vertex and going out of the vertex, i.e., the sum of the indegree and outdegree of the vertex, all the possible paths between the inputs and the outputs were stored. This type of representation of the DFG keeps track of the successors and the predecessors of all the vertices in the DFG. This design method helps in calculating the cost easily and efficiently and also determines whether the current state satisfies the *PS* delay and latency constraints which is explained in detail in later sections.

Each of the vertices is assigned a unique number. If the DFG consists of n vertices then the vertices are labeled 1 through n . Apart from containing the unique

ID number each vertex also stores the information about the component type, component name, delay, and the cost in terms of the number of gates. The realistic CL that is considered is shown in Table 3.1. The CL consists of eight different types of multipliers and six different types of adders and subtractors. The various fields of each component of the CL includes component type, component name, delay in *ns* and cost in terms of the number of gates.

PS delay and latency are also given as inputs along with the DFG. Any initial representation of the solution should be a valid initial solution. A solution is said to be valid if its latency of the initial solution is less than or equal to the given latency and the delay of each of the components mapped onto the DFG is less than or equal to the specified *PS* delay. It is necessary to start with a valid initial solution because in the worst case, when repeated perturbations of the current state does not produce a valid solution, then, the initial solution will be the best solution. If an invalid initial solution is used then the final solution that is obtained may not be a valid solution. The initial solution consisted of mapping the fastest components onto the DFG. In this method components with the smallest delays were mapped onto the DFG. The multipliers corresponding to the *Mpy8* and adders/subtractors corresponding to *Add6/Sub6* of the CL shown in Table 3.1 were used. As the components with the smallest delays have the highest cost (in terms of number of gates), the initial solution will start with the highest cost and lowest latency. The advantage of using this method is that we always start with a valid initial solution satisfying the constraints of *PS* delay and the latency. The initial solution also consists of pipelining the DFG. This is done by traversing the DFG from the inputs towards the output. The delays of each of the components is accumulated. At the

Component types	Component Name	Delay (ns)	Cost (Gates)
*	Mpy1	57.97	2368
*	Mpy2	44.21	2400
*	Mpy3	36.21	2600
*	Mpy4	32.98	2710
*	Mpy5	28.57	2978
*	Mpy6	25.00	3500
*	Mpy7	22.00	4000
*	Mpy8	20.50	4500
+/-	Add1/Sub1	25.80	62
+/-	Add2/Sub2	20.00	125
+/-	Add3/Sub3	13.50	187
+/-	Add4/Sub4	10.00	250
+/-	Add5/Sub5	5.50	375
+/-	Add6/Sub6	3.00	500
Register	Reg		200

Table 3.1: Component Library.

point where the accumulated cost exceeds the specified *PS* delay a register is placed. Therefore the combined delay of the components between two successive registers is less than or equal to the *PS* delay.

3.2 Cost Function

The cost function of the DFG is the cost (number of gates) of all the components of the DFG and the number of pipeline registers. Total cost of the components is :

$$Total \ cost = Cost \ of \ Registers + \sum_{i=1}^n Cost_i \quad (3.1)$$

where n is the number of vertices in the DFG. The cost of the DFG after disturbing the current state changes. This cost is computed such that the time needed to calculate the new cost is minimal.

In order to calculate the cost, an array of size n is maintained where n is the number of vertices in the DFG. The costs of each of the components of the current state are stored in the array. When the current state is perturbed, then one of the components is replaced by another component. Suppose that a component with ID i whose current cost is C_i is replaced by component with cost C_r . If T is the current total cost then the new cost is calculated by the formula shown by

$$New \ cost = T - C_i + C_r + Cost \ of \ Registers \quad (3.2)$$

The cost of the registers is obtained by pipelining the DFG. The cost of the current component being perturbed is obtained from the array at the position corresponding

to the the *ID* number. After computing the new cost, the cost of the component at position i in the array is replaced by the cost of the new component.

3.3 PERTURB Strategy

Each call to the *PERTURB* function involve disturbing all the elements of the DFG. This is done by replacing each component by another component of the same type from the CL. Initially fastest components are mapped onto the DFG. Then subsequent perturbations involve replacing the components of the DFG by other components of the same type from the CL which is chosen at random.

Whenever a multiplier has to be replaced then one of the eight multipliers is chosen at random. Similarly if an adder/subtractor has to be chosen, one of the six different adders/subtractors is chosen. Other different methods can be adopted to select the components from the library, but we have implemented it for the random method only. On replacing a component the corresponding data about the delay and the cost of the component is updated. The *COST* array that keeps track of the costs of each of the components of the current state is also updated.

The components of the DFG are scanned according to some *priori* ordering. Four different types of *PERTURB* strategies were tried. They are discussed in the following subsections.

3.3.1 Random Moves

In this strategy the components are selected at random at every call to the *PERTURB* function. Therefore the order in which the components are scanned in every iter-

ation differs. But in every iteration each of the components is scanned only once. The overhead involved with this strategy is that during every iteration, on choosing a component at random, a check has to be made to see if the component chosen has been perturbed during the same iteration.

Figure 3.1 shows the Random move strategy. The DFG shown in this example consists of four multipliers, and two adders. The vertices that represent the components are labeled 1 through 6. The labeling of the vertices is done randomly. Figure 3.1a represents the initial state or the current state. On the next call to the *PERTURB* function a different scanning order is chosen at random. This is shown in Figure 3.2b. Similarly for every successive call to the *PERTURB* function different random orders are generated for scanning the components. In this strategy it has to be ensured that the components of the DFG are disturbed only once during each iteration.

3.3.2 Fixed Random Moves

In this strategy a fixed random order is used. The same random order is maintained on every call to the *PERTURB* function. The order to be fixed is chosen at random. Figure 3.2 shows the fixed random strategy. In this figure the numbers associated with each of the vertices shows the order in which the vertices are scanned. On every call to the *PERTURB* function, the same order of scanning the nodes is used. The ordering is such that every node of the DFG is disturbed just once whenever *PERTURB* function is called. Different fixed move random strategies were used for experimentation.

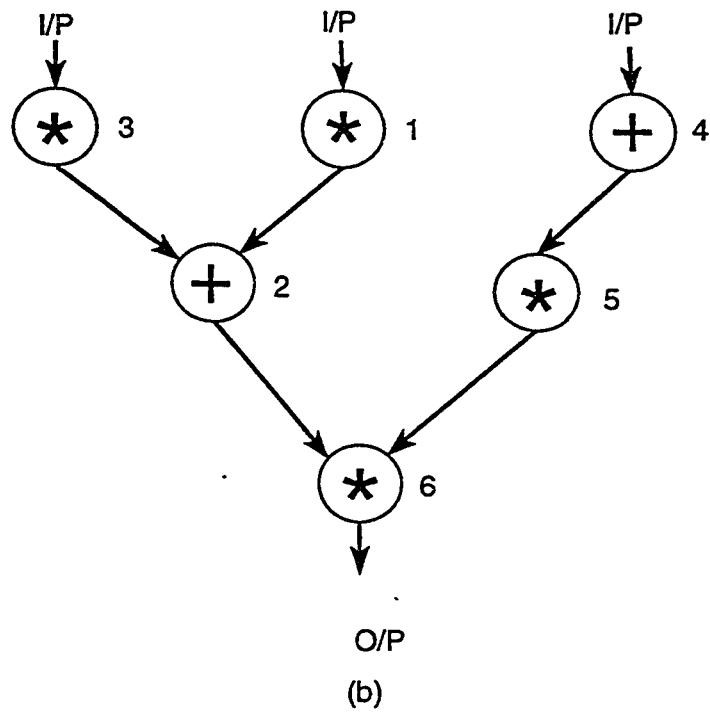
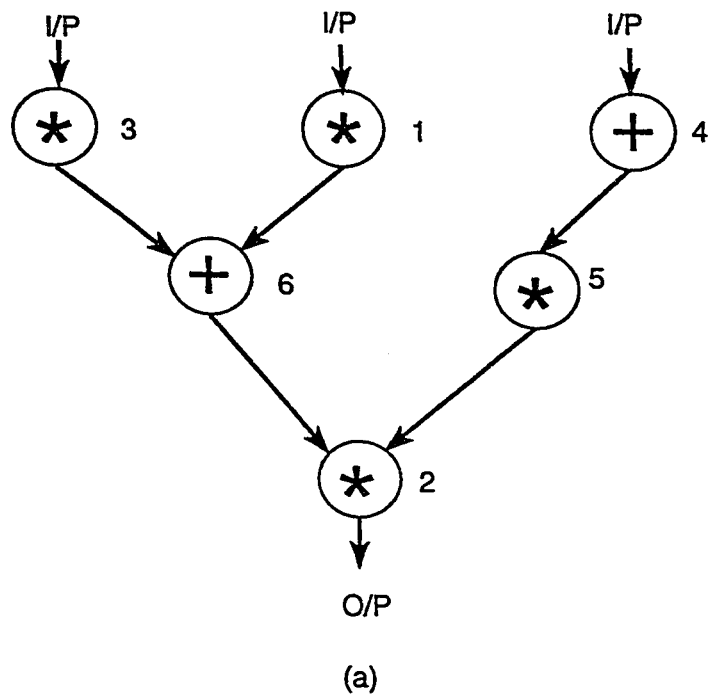


Figure 3.1: Random move strategy.

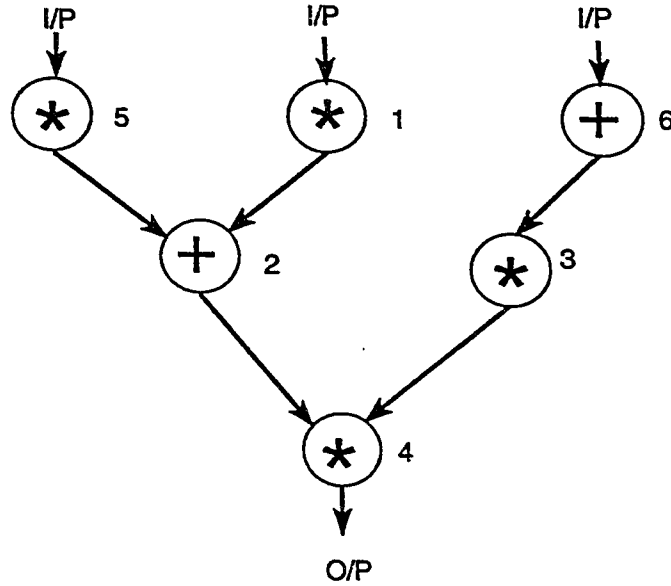


Figure 3.2: Fixed random moves.

3.3.3 Top-Down Strategy

In Top-Down strategy the vertices that are closer to the inputs are perturbed first. This is followed by the next set of vertices at the lower levels. Hence the bottom most vertex, i.e., the vertex preceding the output will be the last one to be disturbed. In Figure 3.3 the vertices are labeled in the manner described above. In the figure shown, vertices {1,2,3} are the first ones to be perturbed, followed by {4,5} and then vertex {6}.

At any particular level of the DFG if there is more than one vertex, then three different approaches can be used

1. **Left to right approach:** In this approach the order in which the elements are scanned are from left to right. The same method is followed at all the

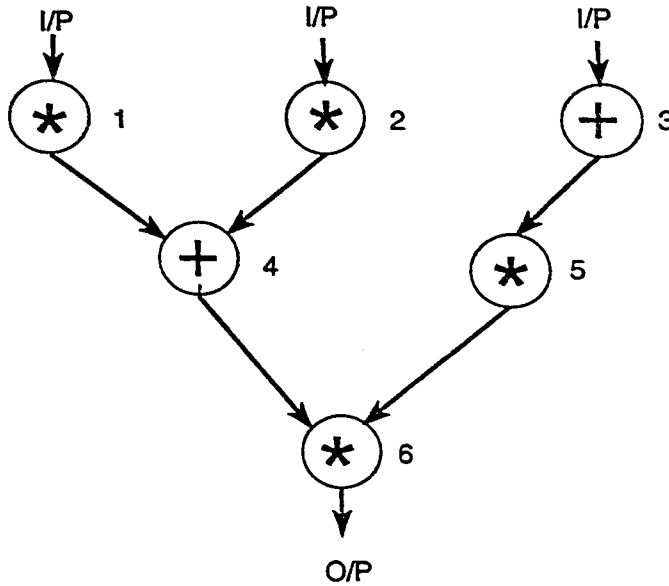


Figure 3.3: Top-Down strategy.

levels of the DFG.

2. **Right to left approach:** In the right to left approach the components of the DFG are scanned from right to left at every level.
3. **Random approach:** The random approach does not follow any particular pattern. The components are chosen at random. In this method the random approach could follow two approaches, namely
 - (a) **Total random approach:** In this approach the elements are disturbed in a random order. The order in which the elements are scanned in subsequent iterations is different. This is done by choosing a particular vertex at random at the particular level being scanned. This process is repeated until all the components at that particular level have been

scanned and it is repeated for the other levels in the DFG. The order in which the elements are chosen at each level in subsequent iterations also differs.

- (b) **Fixed random approach:** In fixed random approach the order in which the components are chosen at a particular level is defined. Hence in subsequent iterations the same order is maintained for scanning the elements.

3.3.4 Bottom-up strategy

In this approach the components nearest to the output will be the first ones to be scanned, while the components near the inputs will be the last one to be scanned. In Figure 3.4 the vertex labeled {1} will be the first one to be disturbed, followed by {2,3} and then finally by {4,5,6}. Within a particular level, the following approaches could be used.

1. Scanning the elements from left to right at every level of the DFG.
2. Scanning the elements from right to left at every level of the DFG.
3. Random method
 - (a) The vertices of the DFG are scanned in a particular order. The same scanning order is maintained for every perturbation.
 - (b) A random order is chosen for scanning the elements. The order is changed randomly for every perturbation.

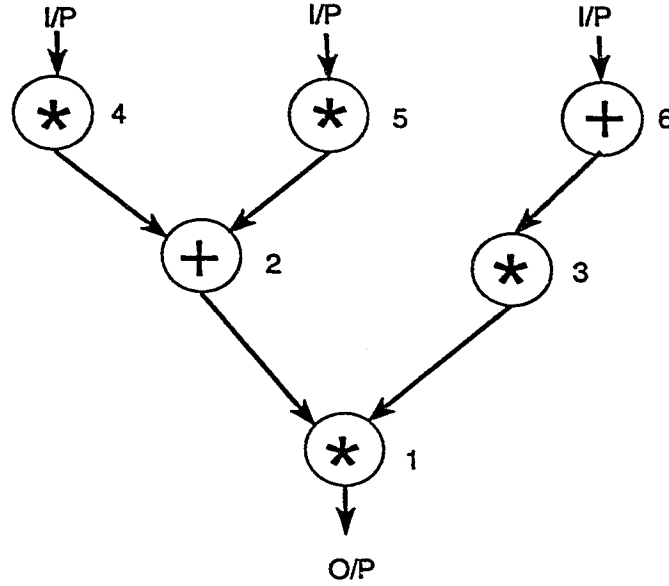


Figure 3.4: Bottom-Up Strategy.

The order in which the vertices of the DFG are scanned does not matter as the components from the component library are chosen at random. For example, if any of the methods suggested for perturbation is used and if a multiplier or an adder is to be replaced, then it is replaced by another adder or multiplier which is chosen at random from the CL. It is due to the randomness in selecting a component from the CL, the four methods mentioned namely random approach, fixed random approach, top-down approach or bottom-up approach did not affect the nature of the results. For the top-down approach and the bottom-up approach, the order of scanning the components at every level was done using all the three methods specified, namely left-to-right approach, right-to-left approach, and random approach. The results did not differ in all the three cases.

3.3.5 Perturb Function

The first time the *PERTURB* function (Figure 3.5) is invoked the initial solution is the current solution. A new solution is obtained by replacing one of the components of the DFG by another component that is selected at random. The cost of this state S_{new} is calculated. The difference between S_{cur} and S_{new} determines the gain. If the gain is less than the random number generated between $-p$ and 0, then the solution is rejected and the *PERTURB* function is again executed with S_{cur} state as the current state. If the gain is greater than the random number then the state S_{new} is accepted and the *PERTURB* function is invoked with S_{new} as the current state S_{cur} . Whenever a component is replaced by another component then the replaced component is of the same type as the previous one. It is ensured that the delay of the replaced component is less than or equal to the specified PS delay. Once a component is disturbed the number of registers and the cost of the new state is calculated. The algorithm is repeated until all the vertices of the DFG are perturbed. At this point it is checked to see if the final state obtained satisfies the constraints. If the state satisfies the constraints then the current state is returned else it is necessary to trace back the moves until a state that satisfies the given constraints is obtained. But as tracing back the moves is complicated and time consuming, a new method was used to keep track of the last valid state that satisfied the constraints. The problem of tracing back is further compounded because the components are disturbed in a random manner.

A new method of keeping track of the last valid state that has to be returned at the end of the execution of the *PERTURB* function was developed. In this

Algorithm *PERTURB*(S, p, lt, pd)

```

     $S \rightarrow$  Current state
     $p \rightarrow$  Non-negative control parameter
     $lt \rightarrow$  latency constraint
     $pd \rightarrow$  PS-Delay constraint

    For  $i = 1$  to  $n$  Do
         $S_{new} = \text{Neighbor}(S_{cur}, i)$ 
        PIPELINE( $S_{new}$ )                                /* pipeline the DFG */
        Gain = Cost( $S_{cur}$ ) - Cost( $S_{new}$ )
        If (Gain > RANDINT(-p,0)) Then
            If ((LATENCY_TEST( $S_{new}$ )) = TRUE) &&
                (PS_DELAY_TEST( $S_{new}$ ) = TRUE)) Then
                 $S_t = S_{new}$ 
            Endif
             $S_{cur} = S_{new}$ 
        Endif
    Endfor
    Return  $S_t$ 
End

```

Figure 3.5: PERTURB algorithm.

method whenever the new state that is obtained by disturbing the current state is accepted it is also checked to see if it satisfies the given constraints. If it satisfies the given constraints then the state is stored in S_t . Hence after disturbing all the components the state S_t is the last valid state, which is returned. Therefore by using this method it is not necessary to reverse the last few moves till a valid state is obtained. A state is valid if the *PS* delay constraint and the latency constraints are satisfied. The latency constraint is checked by traversing the DFG from the inputs towards the output. If on any path between the input and the output the latency of the DFG exceeds that of the specified latency then the state is rejected. The *PS* delay constraint is checked by scanning all the components of the DFG. If the delay of any of the components of the DFG is greater than the specified *PS* delay then the state is rejected. For a particular state to be accepted as S_t it is necessary that both the constraints should be satisfied.

The number of iterations for which the *SE* algorithm is made to run depends on the parameter R , which specifies the reward criteria as explained in the previous chapter. The algorithm was initially carried out with small values of R at first. It was found that the algorithm ran for few hundred iterations and the results obtained were poor. On increasing the the value of R the results improved and the algorithm ran for larger number of iterations. It was found that best results were obtained when the value of R was between 20 and 25. In general if the value of R is increased then the chances of obtaining better results also improve.

3.4 Post Processing

Post-processing was introduced in the *PERTURB* function to check if it improved the results. It is the last step of the *PERTURB* function. The *PERTURB* function is made to disturb the components of the DFG once. After this is done the final state that is obtained satisfies the PS Delay and the latency constraint. The reason for introducing post-processing is that the cost of the DFG could further improve because the combined PS delays of the components between two consecutive registers maybe lesser than the specified PS delay constraint. The *PERTURB* function with post-processing is shown in Figure 3.6.

In post-processing the entire DFG is scanned starting from the inputs towards the output. The delays of the components between two consecutive registers is calculated. If the combined delay of the components between two consecutive registers is less than the specified PS delay then the possibility of replacing one or more components by slower components of the same type is calculated. If this is possible then the components are replaced. The replacement of the components should be done such that the the PS delay and latency constraints are satisfied. By doing this the number of slower components that are used increases and the cost of the solution decreases. Post-processing is done once every time the *PERTURB* function is called. The results obtained by using post-processing showed that there was considerable improvement compared to the results of SE without post-processing.

Algorithm *PERTURB*(S, p, lt, pd)

$S \rightarrow$ Current state
 $p \rightarrow$ Non-negative control parameter
 $lt \rightarrow$ latency constraint
 $pd \rightarrow$ PS-Delay constraint

```

For  $i = 1$  to  $n$  Do
   $S_{new} = \text{Neighbor}(S_{cur}, i)$ 
  PIPELINE( $S_{new}$ )                                /* pipeline the DFG */
   $\text{Gain} = \text{Cost}(S_{cur}) - \text{Cost}(S_{new})$ 
  If ( $\text{Gain} > \text{RANDINT}(-p, 0)$ ) Then
    If (( $\text{LATENCY\_TEST}(S_{new}) = \text{TRUE}$ )) &&
      ( $\text{PS\_DELAY\_TEST}(S_{new}) = \text{TRUE}$ )) Then
       $S_t = S_{new}$ 
    Endif
     $S_{cur} = S_{new}$ 
  Endif
Endfor
POST\_PROCESSING()
Return  $S_t$ 
End

```

Figure 3.6: PERTURB with post processing.

3.5 Update Function

The update function is responsible for updating the value of the control parameter p . Tuning of this parameter is important as it affects the results. Careful tuning of this parameter is necessary for obtaining good results.

In our problem p is the cost associated with the operators. A value equal to the least difference between two components of the component library (CL) was assigned to p and $f(p)$ was assigned a value equal to the maximum difference between two

components in the *CL*. While experimenting with these values of p and $f(p)$ it was observed that the initial drop in the cost was less because only adders/subtractors could be disturbed. As the iterations increased the changes in the costs were high as the value of $f(p)$ was high.

On keeping the value of $f(p)$ large it was observed that the variations in costs were large. Therefore $f(p)$ had to be reduced. This was done by decreasing $f(p)$ in multiples of 100. This was done until the range of variations were lessened. Further reduction of $f(p)$ produced poor results.

As the value of p is equal to the least difference between any two components of the *CL*, only adders/subtractors could be disturbed. Therefore the value of p had to be increased so that multipliers could also be disturbed at the initial value of p . Hence p was increased keeping $f(p)$ constant. By increasing the value of p better results were obtained. On increasing p a certain point was reached beyond which poor results were obtained. At these values of p and $f(p)$ the best results were obtained.

3.6 Parallelizing Stochastic Evolution Algorithm

A common feature of all software tools for circuit design is the excessive demand they place for CPU-time. Optimization problems can be solved to give near-optimal solutions in polynomial time by searching the entire solution space. However the size of the problem prohibits the use of this method due to time constraint. There is hence a need to accelerate computation by using parallel processing techniques.

A *move* or a *trial* in SE for component selection and pipelining consists of the

following tasks:

1. perform a perturbation of the current solution to create a new solution;
2. compute the difference in the cost between the new solution and the current solution;
3. accept the new solution if it's cost is less than the current solution, else accept it with a probability;
4. if the new solution is accepted, then replace the current solution by the new solution. Also test if the solution satisfies the *PS* delay and the latency constraints. If it satisfies then store the solution as the temporary best state. This is to ensure that we do not have to reverse the last number of moves after executing the *PERTURB* function to obtain a valid state that has to be returned as the last temporary state will be the last valid state that has to be returned;
5. replace the new solution as the current solution and repeat the entire process till all the components of the DFG have been disturbed once.

In SE, the parameter R controls the number of the iterations the algorithm is supposed to run. Each time a state is found which has a lower cost than the current best state so far, SE decrements the counter by R , thereby rewarding itself by increasing the number of iterations. One of the important issues that have to be addressed while parallelizing SE is to determine how R is to be decremented.

One of the simplest method of parallelizing SE involves running SE on different processors with different initial solutions. Each of processors runs SE serially and

works on different solution spaces, without overlapping. After completion the best solution among the processors is chosen. The major problem associated with this method is that it is necessary to divide the solution space and it also has to be ensured that the solution does not overlap with those of the other processors, or there is minimal overlap. The division of the solution space such that overlapping does not occur is a difficult task.

For component selection and pipelining, the SE algorithm can run on different processors with the same initial solution. The value of the parameter R is initially the same for all the processors. Then each of the processors runs the algorithm serially. After completing the *PERTURB* function, the best state among all the processors is chosen as the next state. The current value of ρ is chosen from the processor that gave the best state. The value of ρ and the best state is communicated to all the processors and the algorithm is executed. This entire process is repeated until the end of execution. Another variation to this method could be to start with different solutions.

In order to carry out the above process one processor is chosen as the master processor. The master processor is the one that decides which processor has the best current solution after every execution of the *PERTURB* function. It then communicates the current best solution and the value of parameter R to all the other processors.

In the above process some of the processors have to remain idle after every cycle of *PERTURB* function as some of the processors may complete it faster. Hence these processors are forced to wait until all the processors complete. After their completion the master processor decides which processor has the best solution and then

communicates it to the other processors. Some communication protocols have to be designed for communicating between the processors and the master processor.

3.7 Summary

This chapter described the application of SE technique for component selection and pipelining. It also describes the various perturb strategies used to generate neighborhoods, defines the acceptance criteria, and also a method of parallelizing this technique. Post-processing has also been introduced in the SE technique and this has resulted in an improvement in results. The next chapter shows the application of SA for component selection and pipelining.

Chapter 4

Component Selection and Pipelining using Simulated Annealing

This chapter will discuss the Simulated Annealing (SA) technique for component selection and pipelining. An introduction to SA was given in Chapter 2. The main tasks necessary to formulate component selection and pipelining are:

- Finding a proper initial representation.
- Finding an appropriate cost function.
- Generating moves.
- Defining a proper cooling schedule

We will discuss these tasks in the remaining sections of this chapter.

4.1 Initial, Current and Best Solution

The initial solution consists of mapping components from the component library (CL) onto the DFG. This was done by mapping the fastest components from the component library to the DFG. The method used is the same as the one for Stochastic Evolution (SE). This method always produces a valid initial solution. This initial solution will be the current solution before the algorithm is executed and will also be the initial best state.

4.2 Cost Function

In SA the cost is calculated in the same way as it is calculated for SE. This method of calculating cost reduces the time needed to calculate the new cost as it is done incrementally, taking into account only the differences due to local disturbances. In SA, in each iteration only valid solutions are accepted as neighboring solutions, and the cost is calculated only if the current state produced is a valid state.

4.3 Generation of Moves

Generating a new state is done in a manner similar to the one that was followed for SE. All these different methods of perturbations were carried out, namely:

- Random perturbation.
- Fixed random perturbation.
- Top-down approach

- Right-to-left approach
- Left-to-right approach
- Random approach
- Fixed random approach
- Bottom-Up approach
 - Right-to-left approach
 - Left-to-right approach
 - Random approach
 - Fixed random approach

4.4 Metropolis Function

The core of the SE algorithm is the *Metropolis* function which is shown in Figure 4.1. This is responsible for producing neighboring solutions and also defining the basis for accepting the current state. The acceptance of a particular state is inturn dependent on the following parameters: the cooling rate α , a constant β , the initial temperature T_0 , the total time allowed for the annealing process *Maxtime*, and M which represents the time until the next parameter is updated. The tuning of these parameters is explained in the next section.

The *Metropolis* function begins by disturbing the current state to obtain a new state. The new state's cost is calculated and it is accepted if the cost is less than the current state's cost or it is accepted with a certain probability. But in the

Algorithm Metropolis(S_{cur} , T , lt , pd)

```

 $S_{cur}$   $\rightarrow$  Current state
 $T \rightarrow$  Initial temperature  $T$ 
 $lt \rightarrow$  latency constraint
 $pd \rightarrow$  PS-Delay constraint

For  $i = 1$  to  $n$  Do                                /*  $n$  - Number of vertices */
     $S_{new} = \text{Neighbor}(S_{cur}, i)$ 
    PIPELINE(DFG)                                     /* pipeline the DFG */
    Gain = Cost( $S_{cur}$ ) - Cost( $S_{new}$ )
    If ((LATENCY TEST( $S_{new}$ ) = TRUE) &&
        (PS_DELAY TEST( $S_{new}$ ) = TRUE)) Then
        If (Gain > RANDINT() <  $e^{-\Delta c/T}$ ) Then
             $S_{cur} = S_{new}$ 
        Endif
    Endif
Endfor
Return  $S_{cur}$ 
End

```

Figure 4.1: The Metropolis Algorithm.

problem being considered it also has to be determined if the current state satisfies the constraints of *PS* delay and latency.

4.5 Tuning of Parameters

The set of parameters T_0 , α , β , and M specified in the previous sections constitutes the cooling schedule. The cooling schedule can be determined by a trial and error method or by a control mechanism that is problem independent.

The cooling schedule consists of choosing appropriate values for the parameters. The method adopted for choosing these appropriate values is:

1. **Initial Temperature T_0 :** The initial temperature was chosen such that all the transitions are accepted initially. That is, the initial acceptance ratio $\chi(T_0)$ must be close to unity.

$$\chi(T_0) = \frac{\text{Number of moves accepted at } T_0}{\text{Total number of moves attempted at } T_0} \quad (4.1)$$

Initially a very low value of T_0 was chosen. The acceptance ratio was then calculated by running the algorithm for a fixed number of iterations. At low values of T_0 it was found that $\chi(T_0)$ was less than 0.5. Then T_0 was increased in multiples of tens and hundreds. This procedure was repeated until the $\chi(T_0)$ was close to unity in the range between 0.9 and 1.0.

2. **Choosing α :** The choice of α should be such that temperature T_0 should be reduced at a uniform rate. Furthermore T_0 should not approach zero quickly. Therefore the temperature is reduced in geometric progression as shown in

Equation 4.2.

$$T_{k+1} = \alpha * T_k, \quad k = 0, 1, \dots, \quad (4.2)$$

In our problem it was noticed that for values of α , between the range of 0.8 and 0.94 resulted in variations in the cost only for 100 - 150 iterations after which the results remained constant and the results obtained were poor. At $\alpha = 0.98$ best results were obtained.

3. **Choosing *Maxtime*:** Initially the algorithm was executed for 100 iterations. At the end of the 100th iteration it was found that the cost was not stabilizing. In order to stop the algorithm it is necessary to run the algorithm until the result remains constant for a certain number of iterations. Then *Maxtime* was slowly increased in multiples of 50 until the cost remained constant for about 200 - 400 iterations.
4. **Choosing *M*:** This is equivalent to the number of times the *Metropolis* loop is executed at a given temperature. *M* was chosen to be equal to the number of vertices in the DFG and it was kept constant.
5. **Choosing β :** β was kept at a constant value of 1 so that every time *Metropolis* is invoked, all the components of the DFG are disturbed just once.

4.6 Parallelizing simulated annealing

Acceleration of simulated annealing has been a very important area of research since the invention of simulated annealing algorithm itself. Several acceleration techniques

have been reported, which can be classified into three general categories [38]:

1. Design of faster serial annealing namely by using faster schedule [39, 40, 41, 42].
2. Hardware acceleration which consists of implementing time consuming parts in hardware.
3. Parallel acceleration, where execution of the algorithm is partitioned on several concurrently running processors [43, 44].

Parallel computation offers a great opportunity for sizeable improvement in the solution of large and hard problems that would otherwise have been impractical to tackle on a sequential computer.

The main task that has to be taken into consideration for parallelizing is to determine the design, distribution and method of accepting the current solution that is obtained by disturbing the current state. The *PERTURB* or the function that disturbs the current state involves the following steps:

1. perform a perturbation to disturb the current solution to obtain a new solution,
2. test if the current solution satisfies the constraints of *PS* delay and latency,
3. compute the difference in costs between the current state and the new state,
4. if the cost of the new solution is lesser than the current state and it satisfies the constraints then accept the solution, else, accept it with a probability, and
5. if the new solution is the best one that has been obtained so far then replace the current best solution by the new solution.

The above method is similar to the SE process with the exception that in SE the parameter ρ also has to be communicated. Therefore the method of parallelizing SA is the same as the one for SE.

4.7 Summary

This chapter described the SA algorithm for solving the problem of component selection and pipelining. The various strategies to obtain neighborhood states, and the acceptance criteria, have been explained. The next chapter discusses the results of both the SE and SA algorithms that were obtained when they were applied to the problem of component selection and pipelining.

Chapter 5

Experimental Results

Stochastic Evolution (SE) and Simulated Annealing (SA) were tested on different types of Data Flow Graphs (DFG). They differed in the number of nodes, types of operators and the complexity of the interconnections between the nodes. The depth of the DFG's varied from 7 to 15. The characteristics of the DFGs are shown in Table 5.1. A total of six different graphs were used for experimentation. The stochastic evolution (SE) and simulated annealing (SA) techniques were applied. Each of the DFGs were run a number of times and the results were tabulated. Latency and PS delay for each of the DFGs were different as shown in the tables. The DFGs are labeled EWF (Elliptical wave filter) and Graph 1 through graph 5. The results of SE without post-processing for different values of R are shown in Table 5.2. The results of SE and SA are shown in Table 5.3. Table 5.4 shows the results of SA and MSE (post processing).

GRAPH	Depth	No. of Vertices
EWF	4	14
Graph 1	14	23
Graph 2	11	23
Graph 3	10	23
Graph 4	15	27
Graph 5	13	17

Table 5.1: Characteristic of input graphs.

Tables 5.3 and 5.4 show the average cost, best cost and time (in seconds) of both SA and SE, where the cost is indicated in terms of the number of gates. Improvement of SA over SE is shown as % reduction. Table 5.5 shows a comparison of the results of SA, SE and MSE with post-processing. It is also shown as a bar chart in Figure 5.3.

The variations in costs for SA are initially large and then it gradually decreases and remains constant after a certain period of time. This is because the cooling schedule in SA decreases gradually hence it accepts solutions with higher costs initially and then progressively accepts solutions with less variations in costs. Then a point is reached when it accepts only good solutions, i.e., it accepts solutions whose cost is lesser than the current solution.

In case of SE there is an initial drop in the cost in the first few iterations then the cost varies within a range. The initial drop is because the initial cost is the

DFG	R	Cost (gates)		Iterations	Latency	PS delay	Time
		Avg.	best				
EWF	10	23175	21100	27	86	37	7
	15	22277	21100	55	86	37	19
	20	21938	21100	118	86	37	27
	25	21278	21100	129	86	37	29
Graph 1	10	36592	36013	94	247	63	223
	15	36544	35641	185	247	63	431
	20	35301	34749	328	247	63	753
	25	33505	34958	234	247	63	621
Graph 2	10	72667	69360	342	203	63	223
	15	68362	67896	346	203	63	815
	20	68063	67889	622	203	63	1462
	25	67883	67742	892	203	63	2060
Graph 3	10	39016	37847	119	187	63	161
	15	38360	37101	175	187	63	234
	20	37209	36689	398	187	63	533
	25	37753	37447	328	187	63	434
Graph 4	10	59589	58912	136	307	63	591
	15	57266	57158	308	307	63	1173
	20	55841	55247	447	307	63	1719
	25	55123	54259	831	307	63	3183
Graph 5	10	40235	39147	148	286	63	318
	15	40492	39901	287	286	63	627
	20	39647	38440	475	286	63	1071
	25	39228	38926	515	286	63	1112

Table 5.2: Stochastic Evolution results.

DFG	Latency	PS Delay	Stochastic Evolution			Simulated Annealing			% Red.
			Avg. cost	Best cost	Time	Avg. cost	Best cost	Time	
EWF	86	37	21865	21807	241	20674	20674	239	-5.48 %
Graph 1	247	63	36309	36053	2396	34596	34569	2138	-4.29 %
Graph 2	203	63	69989	67717	2375	66281	66209	2237	-2.27 %
Graph 3	187	63	38432	38333	1380	37199	37198	1375	-3.04 %
Graph 4	307	63	56310	55820	3650	52901	52696	3189	-5.92 %
Graph 5	286	63	39916	39480	2241	39083	39042	2141	-1.12 %

Table 5.3: Stochastic Evolution and Simulated Annealing results.

DFG	Latency	PS Delay	Stochastic Evolution			Simulated Annealing			% Red.
			Avg cost	Best cost	Time	Avg cost	Best cost	Time	
EWF	86	37	21100	21100	241	20674	20674	239	-2.6 %
Graph 1	247	63	34743	34517	2451	34596	34569	2050	+0.15 %
Graph 2	203	63	67009	65931	2332	66281	66209	2237	+0.42 %
Graph 3	187	63	37221	36981	1388	37289	37199	1375	+0.59 %
Graph 4	307	63	55909	54457	3939	52901	52696	3189	-3.34 %
Graph 5	286	63	39183	38855	2283	39083	39042	2141	+0.48 %

Table 5.4: Modified Stochastic Evolution (post-processing) and Simulated Annealing results.

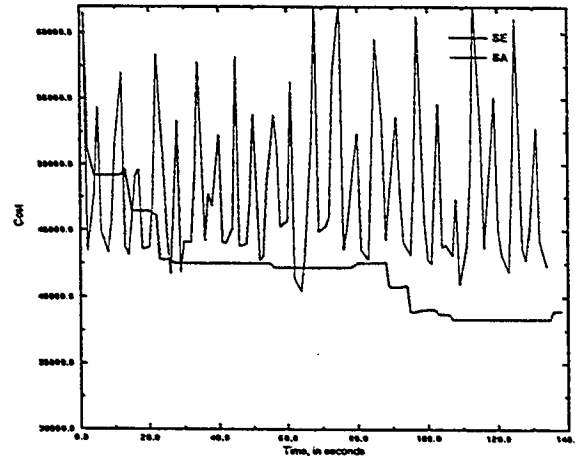


Figure 5.1: Plot showing the comparison between SA and MSE during the initial stages.

highest and when each of the components are disturbed then they are replaced by components with lesser cost, hence the total cost is much less.

It can be seen from the graphs that the performance of SA and SE is comparable during the initial stages while SA outperforms SE in the later stages. Figure 5.1 shows an example of the plot for SA and Modified Stochastic Evolution (MSE) during first few iterations. In this figure it is seen that MSE performs better than SA. The barchart shown in Figure 5.2 shows that MSE performs better than SA in four of the six DFGs.

This is because in case of SE, it allows hill-climbing throughout while SA allows hill-climbing such that it progressively decreases the range of costs for accepting poorer solutions and then a point is reached when it does not accept poor solutions, i.e., a point is reached when it accepts only solutions that are better than the current solutions thus preventing hill-climbing at later stages.

The performance of the DFGs for SE and SA are shown in Figures 5.4 and 5.5.

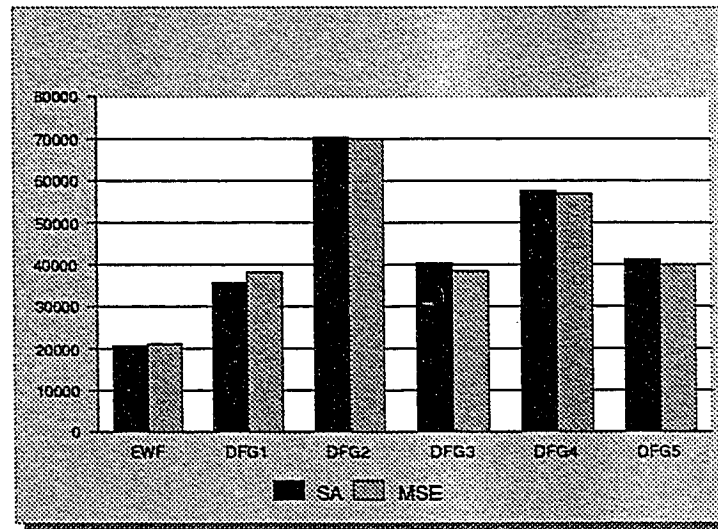


Figure 5.2: Barchart showing the results of SA and MSE during the initial stages.

The graphs are plotted to show cost (number of gates) versus time.

Figures 5.6 and 5.7 represents the plots of SA and SE with post-processing. It is seen that in four of the six DFG's SE performs better than SA. This is because in post-processing, after every iteration the entire DFG is scanned and if there is a possibility of replacing one or more components by slower components without violating the constraints then they are replaced. It is due to this that the cost decreases.

This chapter discussed the results of SA, SE, and MSE with post-processing. The conclusions and the future work are presented in the next chapter.

Simulated Annealing	Stochastic Evolution	Stochastic Evolution (post-processing)
20674	21807	21100
34569	36053	34517
66209	67717	65931
37198	38333	36981
52696	55820	54457
39042	39480	38855

Table 5.5: Comparisons between SA, SE and MSE (post-processing).

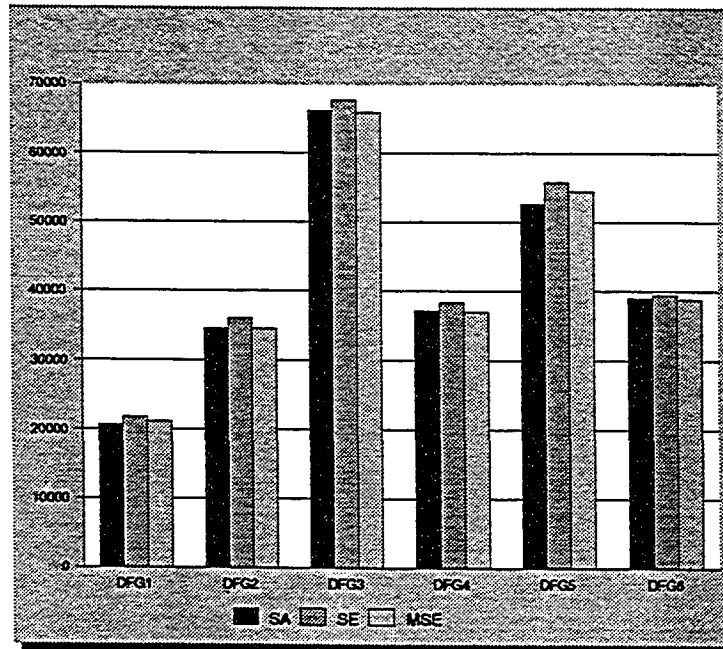


Figure 5.3: A barchart showing the comparisons between SA, SE and MSE (post-processing).

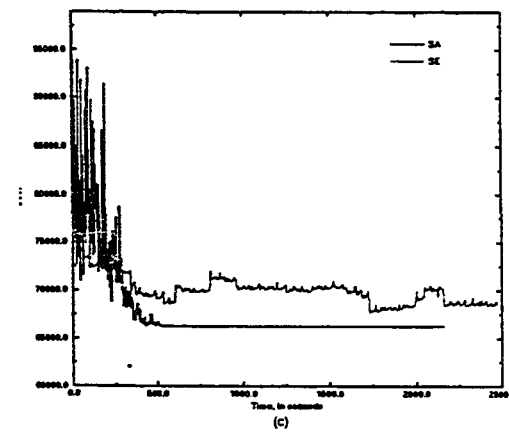
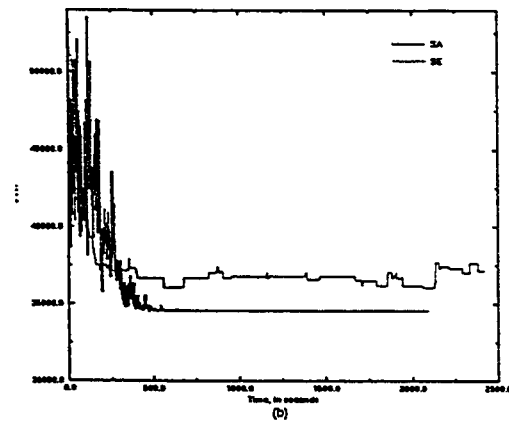
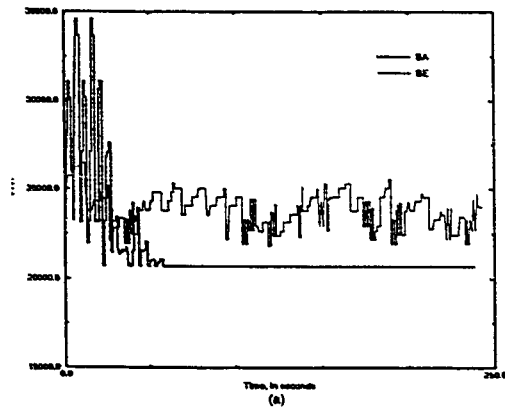


Figure 5.4: Plots for different DFG's using SA and SE techniques: (a) EWF, (b) Graph 1, and (c) Graph 2 (contd).

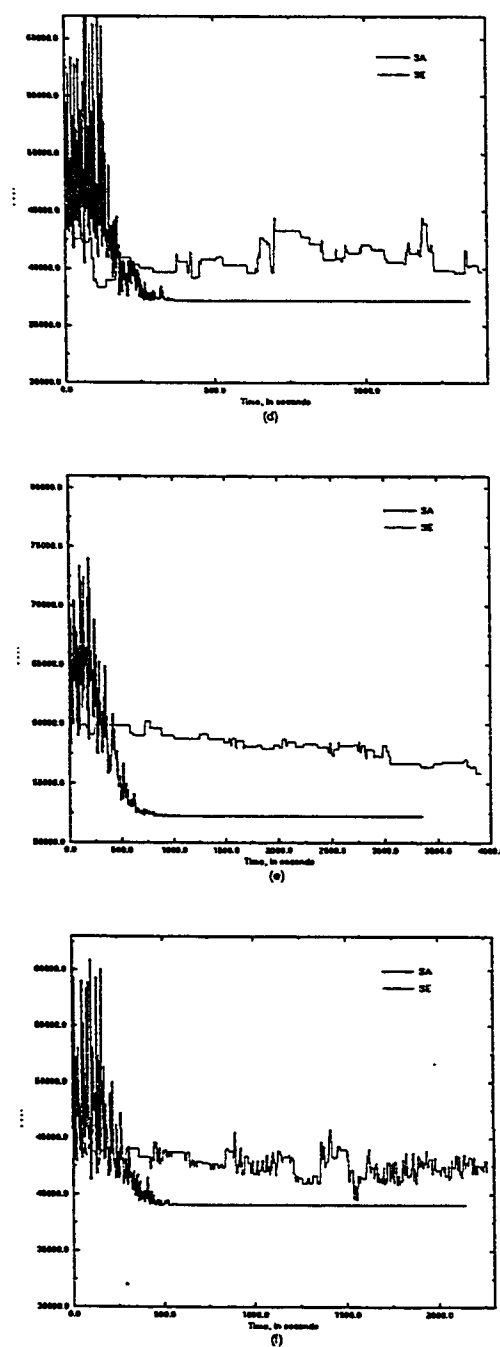


Figure 5.5: Plots for different DFG's using SA and SE techniques: (d) Graph 3, (e) Graph 4, and (f) Graph 5.

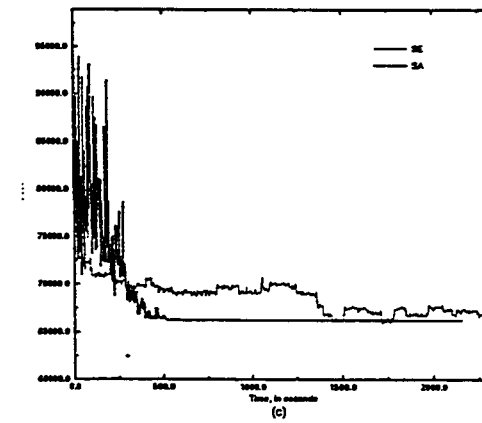
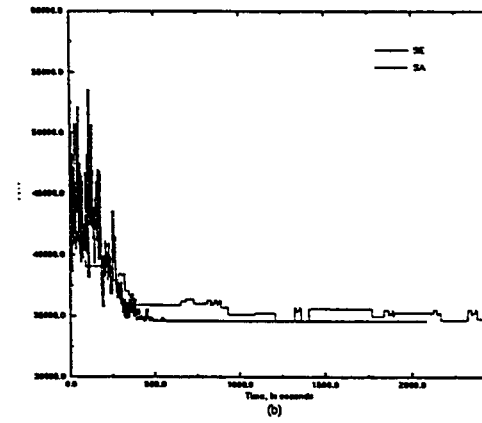
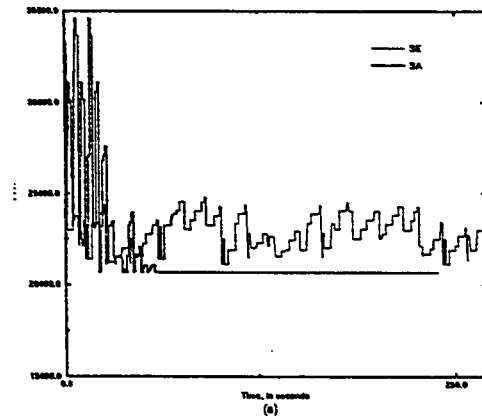


Figure 5.6: Plots for different DFG's using SA and MSE (post-processing) techniques: (a) EWF, (b) Graph 1, and (c) Graph 2 (contd).

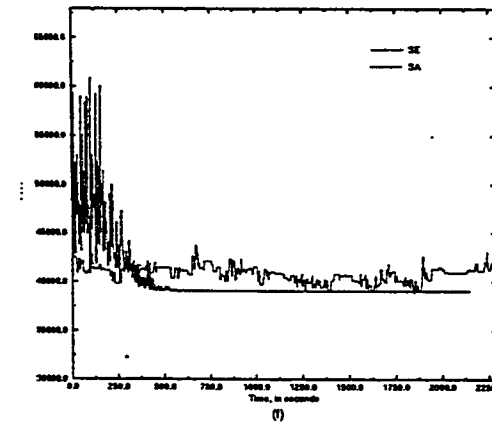
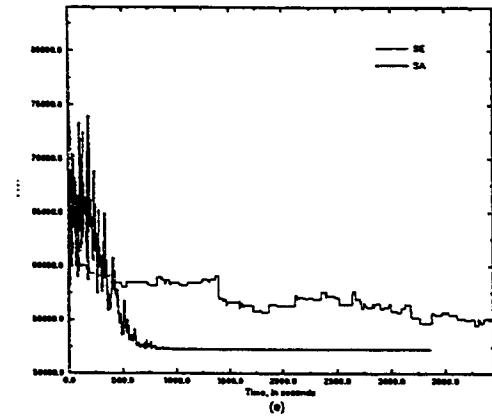
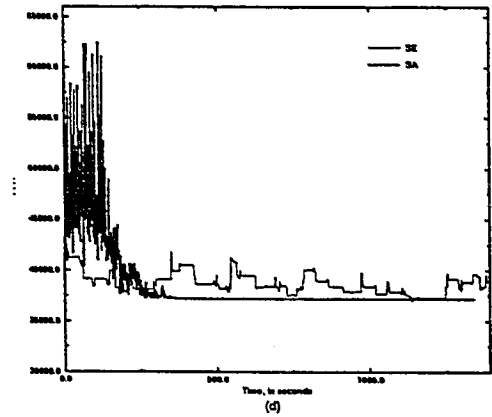


Figure 5.7: Plots for different DFG's using SA and MSE (post-processing) techniques: (d) Graph 3, (e) Graph 4, and (f) Graph 5.

Chapter 6

Conclusions and Future work

6.1 Conclusions

Stochastic evolution is a promising optimization technique. This thesis has presented its application to component selection and pipelining, a problem in high-level synthesis. The work involved finding an appropriate representation of the problem. This is necessary as it provides the flexibility to apply various optimization techniques. This representation helps in maintaining and modifying the data in an efficient and simple way. Various ordering strategies have been experimented with for the problem to determine a strategy that helps in obtaining cost-effective solutions. It was seen that the solutions were not affected irrespective of the ordering being used. This is because of the randomness in selecting components from the component library. A method for tuning the parameters of the update function has been developed to obtain cost-effective results. As the value of the parameter R , which determines the number of iterations SE is to be run increases the chances of getting a better

solution also increase.

A new method of improving results in SE called post-processing has been introduced. In post-processing, the DFG is scanned after obtaining a new state by disturbing the previous state. The new state is checked to see if the components can be replaced by slower components of the same type without violating the constraints. This has produced a substantial reduction in the costs. SE algorithm was tested on six different DFGs with varying complexities. It was found that SE with post-processing performed better than SE without post-processing in all the six DFGs.

Simulated annealing is another promising optimization technique. This thesis has presented its application to component selection and pipelining. The representation of the problem and the perturb strategies used were the same as those for SE. An important part of SA algorithm is developing an effective cooling schedule. A method for tuning the various control parameters of the cooling schedule has been defined and implemented. SA was also tested on the same six DFGs as that of SE.

SA performed better than SE without post-processing in all the six DFGs, while SE with post-processing performed better than SA in four of the six DFGs.

6.2 Future Work

The future work in case of SE could be done in the following areas.

- Finding better methods of representing the problem for efficient management.

Work could also be directed towards finding ways to reduce the time taken for processing.

- New methods could be designed to tune the value of R to obtain it's optimal value to obtain cost-effective results.
- The update function in the SE algorithm determines the range of negative gains to be accepted. Different methods could be formulated to determine the range of the negative gains to be accepted to improve the results.

In case of SA the work could be concentrated in developing better cooling schedules. Different optimization techniques could also be used to solve the problem of component selection and pipelining.

Bibliography

- [1] Sadiq M. Sait and Habib Youssef. "VLSI Design automation: Theory and practice". *Mc-GrawHill Book Co. Europe*.1995.
- [2] M. C. McFarland, A. C. Parker, and R. Camposano. "Tutorial on high-level synthesis". *In Proceedings of the 25th Design Automation Conference*, pages 330–336, 1988.
- [3] S. Bakshi and D. D. Gajski. "A component selection algorithm for high-performance pipeline". *EuroDac '94 + Euro VHDL '94*, pages 400–405, 1994.
- [4] L. Ramachandran and D. D. Gajski. "An algorithm for component selection in performance optimized scheduling". *In Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 92–95, 1991.
- [5] M. C. McFarland, A. C. Parker, and R. Camposano. "The high-level synthesis of digital systems". *Proceedings of the IEEE*, 78(2):301–318, 1990.
- [6] L. J. Hafer and A. C. Parker. "Register-transfer level digital design automation: the allocation process". *Proceedings of th 15th Design Automation Conference*, pages 213–219, 1978.

- [7] C. Y. Hitchcock and D. E. Thomas. "A method of automatic data path synthesis". In *Proceedings of the 20th Design Automation Conference*, pages 484–489, 1983.
- [8] H. Trickey. "Flamel: A high-level hardware compiler". *IEEE Transactions on Computer-Aided Design*, 5(3):259–269, 1986.
- [9] B. M. Pangrle. "Splicer: A heuristic approach to connectivity binding". In *Proceedings of the 25th Design Automation Conference*, pages 536–541, 1988.
- [10] Fur-Shing, Tsai, and Yu-Chin Hsu. "An automatic data path allocator". *IEEE Transactions on Computer-Aided Design*, 11(9):1053–1064, 1992.
- [11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". *Science*, 220(4958):671–680, 1983.
- [12] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallet. "Some experiments in local microcode compaction for horizontal machines". *IEEE Transactions on Computer Aided Design*, 30(7):460–477, 1981.
- [13] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn. *"Algorithmic and Register -Transfer level Synthesis: The Systems Architects Workbench"*. Kluwer Academic Publishers, Norwell, MA, 1990.
- [14] P. G. Paulin and J. P. Knight. "Force-directed scheduling for behavioral synthesis on ASIC's". *IEEE Transactions on Computer Aided Design*, 8(6):661–679, 1989.

- [15] K. S. Hwang, A. E. Casavant, C. T. Chang, and M. A. d'Abreu. "Scheduling and hardware sharing in pipelined data paths". *In Proceedings of the IEEE International Conference on Computer-Aided Design*, 1989.
- [16] Hong-Shin and Sun-Young Hwang. "Design of a pipelined datapath synthesis system for digital signal processing". *IEEE Transactions on Very large Scale Integration(VLSI) systems*, 2(3):292–303, 1994.
- [17] N. Park and A. C. Parker. "Sehwa: A software package for synthesis of pipelines from behavioral specifications". *IEEE Transactions on Computer-Aided Design*, 7:356–370, 1988.
- [18] Daniel Audet, Yvon Savaria, and Nicholas Arel. "Pipelining Communications in Large VLSI/ULSI systems". *IEEE Transactions on Very large Scale Integration(VLSI) systems*, 2(1):1–9, 1994.
- [19] Said Amellal and Bozena Kaminska. "Functional synthesis of digital systems with TASS". *IEEE Transactions on Computer Aided Design*, 13(5):537–552, 1994.
- [20] S. Ali, Sadiq M. Sait, and M. S. T. Benten. "Application of Tabu Search in High-level Synthesis of Digital Systems". *International Conference on Electronics, Circuits and Systems*, pages 423–428, 1994.
- [21] S. Ali, Sadiq M. Sait, and M. S. T. Benten. "GSA: Scheduling and Allocation using Genetic Algorithm". *European Design Automation Conference with Euro-VHDL*, pages 84–89, 1994.

- [22] C. T. Hwang, Y. C. Hsu, and Y. L. Lin. "PLS: A scheduler for pipeline synthesis". *IEEE Transactions on Computer-Aided Design*, pages 24–27, 1989.
- [23] R. Jain, A. Parker, and N. Park. "Module selection for pipelined synthesis". In *Proceedings of the 25th Design Automation Conference*, pages 542–547, 1988.
- [24] R. Jain, A. Parker, and N. Park. "Module selection for pipelined synthesis with multi-cycle operations". In *Proceedings of the IEEE Conference on Computer-Aided Design*, pages 212–215, 1990.
- [25] A. H. Timmer, L. Stok, M. J. M. Heijligers, and J. A. G. Jess. "Module selection and scheduling using unrestricted libraries". In *Proceedings of the European Design Automation Conference*, pages 547–551, 1993.
- [26] H. F. Al-Sukhni, H. Youssef, Sadiq M. Sait, and M. S. T. Benten. "A New Loop Based Scheduling algorithm". *IEEE Phoenix conference on Computers and Communications*, pages 76–81, 1995.
- [27] Y. G. Saab and V. B. Rao. "Combinatorial optimization by stochastic evolution". *IEEE Transactions on Computer-Aided Design*, 10(4):525–535, April 1991.
- [28] S. Nahar, S. Shani, and E. Shragowitz. "Simulated annealing and combinatorial optimization". In *Proceedings of the 23rd Design Automation Conference*, pages 293–299, 1986.
- [29] David Connolly. General purpose simulated annealing. *Journal of Operations Research Society*, 43(5):495–505, 1992.

- [30] C. Koulamas, SR. Antony, and R. Jaen. "A survey of simulated annealing : Applications to operations research problems". *International journal of Management Sciences*, 22(1):41–56, 1994.
- [31] David E. Jeffcoat and Robert L. Bulfin. "Simulated annealing for resource-constrained scheduling". *European journal of operations research*, 70(5):43–51, 1993.
- [32] S. A. Kravitz and R. B. Rutenbar. "Placement by simulated anealing on a Multiprocessor". *IEEE Transactions on Computer-Aided Design*, CAD(4):534–549, july 1987.
- [33] D. W. Jepsen and Jr. C. D. Gelatt. "Macro Placement by Monte Carlo Annealing". *Proc. International Conference on Computer Aided Design*, pages 495–498, Nov 1984.
- [34] H. W. Leong, D. F. Wong, and C. L. Liu. "A simulated annealing channel router". *Proceedings ICCAD*, pages 226–228, 1985.
- [35] R. B. Rutenbar. "Simulated Annealing Algorithms: An overview". *IEEE circuits and Devices Magazine*, pages 19–26, 1989.
- [36] M. Vecchi and S. kirkpatrick. "Global wiring by simulated annealing". *IEEE transactions on Computer-Aided Design*, CAD:215–222, 1984.
- [37] N. Metropolis et al. "Equation of state calculations by fast computing machines". *Journal of Chem, Physics*, 21:1087–1092, 1953.

- [38] Emile Aarts and Jan Korst. "Simulated Annealing and Boltzmann MACHines: A stochastic Approach to Combinatorial Optimization and Neural Computing". *John-Wiley and Sons Ltd*, 1989.
- [39] H. Szu and R. Hartley. "Fast simulated annealing". *Physics Letters*, 122(4):157–162, June 1987.
- [40] J. W. Greene and K. J. Supowit. "Simulated annealing without rejected moves". *IEEE Transactions on Computer Aided Design*, (5):221–228, 1986.
- [41] P. J. M. Laarhoven and E. H. L. Aarts. "Simulated Annealing : Theory and Applications". *Reidel, Dordrecht*, 1987.
- [42] F. Catthoor, H. DeMan, and J. Vandewalle. "A parallel simulated simulated-annealing schedule with fully adaptive annealing parameters". *Integration*, (6):147–178, 1988.
- [43] F. Darema, S. Kirkpatrick, and V. A. Norton. "Parallel techniques for chip placement by simulated annealing". *IBM journal of Research and Development*, (5):391–402, May 1987.
- [44] F. Darema, S. Kirkpatrick, and V.A. Norton. "Parallel algorithms for cell placement". *Proceedings of International Conference on Computer Designs: VLSI in Computers and Processors*, ICCD-87(5):87–90, 1987.

Vita

- Mohammad Farook
- Born in Bangalore, India
- Received Bachelor of Engineering Degree in Computer Science from University Visvesvaraya College of Engineering, Bangalore University, Bangalore, India in 1991.
- Joined the Department of Information and Computer Science at King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia as a Research Assistant in January 1994
- Received Master of Science (M.S.) degree in Information and Computer Science from KFUPM, Saudi Arabia in June 1996