Scheduling and Allocation in High-Level Synthesis using Genetic Algorithm

by

Shahid Ali

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, 1994

Order Number 1360424

Scheduling and allocation in high-level synthesis using genetic algorithm

Ali, Shahid, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

SCHEDULING AND ALLOCATION IN HIGH-LEVEL SYNTHESIS USING GENETIC ALGORITHM

BY

SHAHID ALI

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 1994

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

This thesis, written by

Shahid Ali

under the direction of his Thesis Advisor, and approved by his Thesis committee, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Thesis Committee:

Muhammad Shafiq

Chairman (Dr. M. Shafique)

Sadiq Sait · M

Co-chairman (Dr. Sadiq M. Sait)

Member (Dr. M. S. T. Benten)

Member (Dr. T. H. Maghrabi)

Dr. M. Al-Mulhem
Department Chairman

Dr. Ala H. Rabeh
Dean, College of Graduate Studies

Date: 29.5.96

كلية الدراسات العليا
COLLEGE OF GRADUATE STUDIES

Dedicated to

My Parents

and Brothers

Acknowledgments

All Praise is for Almighty Allah for having guided me at every stage of my life.

I would like to express my gratitude to my thesis committee chairman Dr. M. Shafique for his assistance and support. I am also highly thankful to my co-chairman Dr. Sadiq M. Sait for his time, guidance and assistance. I am also grateful to my committee members Dr. M. S. T. Benten and Dr. T. H. Maghrabi for their sincere help and suggestions.

I also acknowledge the generous help and support for this research provided by King Fahd University of Petroleum and Minerals. I am indebted to my department chairman Dr. M. Al-Mulhem for his assistance and support. My thanks also to all faculty members and my friends, both from within and outside the department, who made my stay at the university a pleasant and memorable one. Special thanks go to my fellow and friend late Emad Amin Khan for his moral support and encouragement.

Contents

List of Tables

List of Figures

Abstract

Name: Shahid Ali

Title: Scheduling and Allocation in High-Level Synthesis using Genetic Algorithm

Major Field: Computer Science

Date of Degree: June 1994

High-level synthesis (HLS) is the process of automatically translating abstract behavioral models of digital systems to implementable hardware. Operation scheduling and hardware allocation are the two most important phases in the synthesis of circuits from behavioral specification. Scheduling and allocation can be formulated as an optimization problem. In this work, a unique approach to scheduling and allocation problem using the genetic paradigm is described. The main contributions include: (1) a new chromosomal representation for scheduling and for two subproblems of allocation, and (2) two novel crossover operators to generate legal schedules. In addition the application of tabu search to scheduling and allocation is also implemented and studied. Both techniques have been tested on various benchmarks and results obtained for data-oriented control-data flow graphs (CDFGs) are compared with other implementations. A novel interconnect optimization technique using genetic algorithm is also realized.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

June 1994

ملخص

الإســـــم: شاهد علي

العنــوان: الجدولة والتوزيع في التصميم عالي-المستوى باستخدام خوارزمية جيناتية

التخصص الرئيسي: علوم الحاسب والمعلومات

تاريخ الدرجـــــــة: ذو الحجة ١٤١٤ - حزيران ١٩٩٤

يعرّف التصميم عالي المستوى بأنه عملية التحويل الآلي لنماذج النظـم الرقميـة الموصوفـة إلى مكونات قابلة للبناء. وتعتبر جدولة العمليات وتوزيع المكونات من أهم مراحل تصميم الدوائر عـالي المستوى. ويمكن صياغة مسألتي الجدولة والتوزيع كنوع من مسائل تحديد الفعّالية القصوى.

يقدم هذا البحث دراسة لطريقة فريدة للجدولـة والتوزيـع باسـتخدام خوارزميـات جيناتيـة. ويمكن تلخيص أهم عناصر البحث كما يلي:

١- تمثيل كروموسومي جديد للجدولة ولمسألتين من مسائل التوزيع.

٢- عمليتا عبور جديدتان لتوليد جداول مقبولة.

أضافة إلى أنه تم دراسة وبناء عملية البحث المعروفة باسم "البحث الممنوع" كمـا تم تطبيقـه واستخدامه في الجدولة والتوزيع. وقد تم فحص كلا الطريقتين باختبارها في مجال "رسومات انسياب بيانات التحكم المعتمدة على البيانات" وتم مقارنة النتائج المتحصلة بنتائج معروفة سابقا لطرق أخرى. كما تم التحقق من طريقة لحل مسألة تحديد الفعّالية القصوى باستخدام خوارزمية جيناتية.

ماجستير في العلوم

علوم الحاسب والمعلومات

جامعة الملك فهد للبترول والمعادن

الظهران - المملكة العربية السعودية

ذو الحجة ١٤١٤ - حزيران ١٩٩٤

Chapter 1

Introduction

High-level synthesis (HLS) is the process of automatically translating abstract behavioral models of digital systems to implementable hardware. The behavioral specification is a high-level abstraction generally at the algorithmic level. It is usually written using a high-level programming language. It consists of circuit outputs in terms of circuit inputs without reference to any structure. Thus, while the behavioral specification aims at describing only the functionality of a circuit (what a circuit must do), the structure gives strong hints about the implementation (how the circuit must be built).

The place of HLS in design hierarchy is illustrated in Figure 1.1 [MPC90]. Hardware can be described in three domains - behavioral, structural and physical. The level of detail increases from left to right in Figure 1.1. Also, five levels of abstraction are shown in which synthesis can take place, namely, system, algorith-

mic, register-transfer, logic, and circuit levels. Going from algorithmic behavior to register-transfer level structure is HLS as shown by an arrow in Figure 1.1.

Figure 1.2 shows the relationship of HLS with the logic synthesis and layout synthesis in design automation. Design automation refers to the automatic synthesis of a physical design from some higher-level behavioral specification [SY45]. Logic synthesis converts a structural design, in terms of an interconnected set of register-transfer level components, into optimized combinational logic, and maps that logic onto the library of available cells (in a particular technology). Layout synthesis converts an interconnected set of cells, which describes the structure (topology) of a design, into the exact physical geometry (layout) of the design. An integrated synthesis system that offers all three synthesis levels is often referred to as a *silicon compiler* [WC91]. Logic and layout synthesis tools have gained a stable foothold in industry. HLS is the next step in the ladder of the design automation hierarchy. HLS provides shorter design cycle, fewer errors, exploration of dif ferent trade-offs between cost, speed, etc., and availability of IC technology to more people [MPC88]. So far automated synthesis has produced chips which are bigger and slower as compared to the chips designed by experienced (human) chip designers. This is a big challenge for the ongoing research efforts in this area.

The steps involved in HLS are illustrated in Figure 1.3. In order to extract the structure, the algorithmic specifications are first converted to an intermediate form (IF) such as Control/Data Flow Graphs (CDFGs), in which *nodes* correspond to

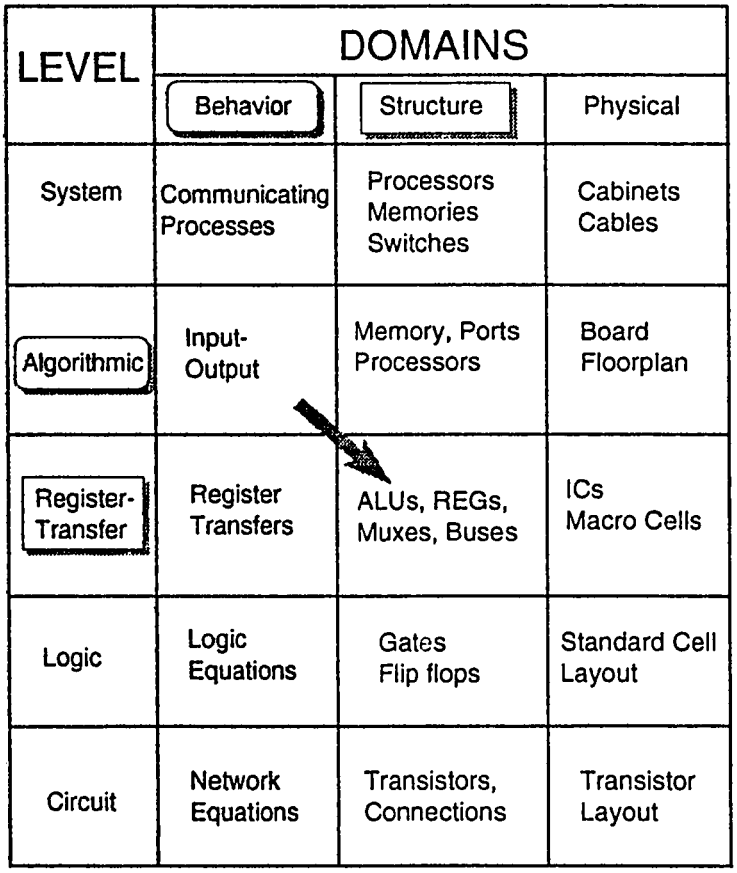


Figure 1.1: Place of HLS in design hierarchy.

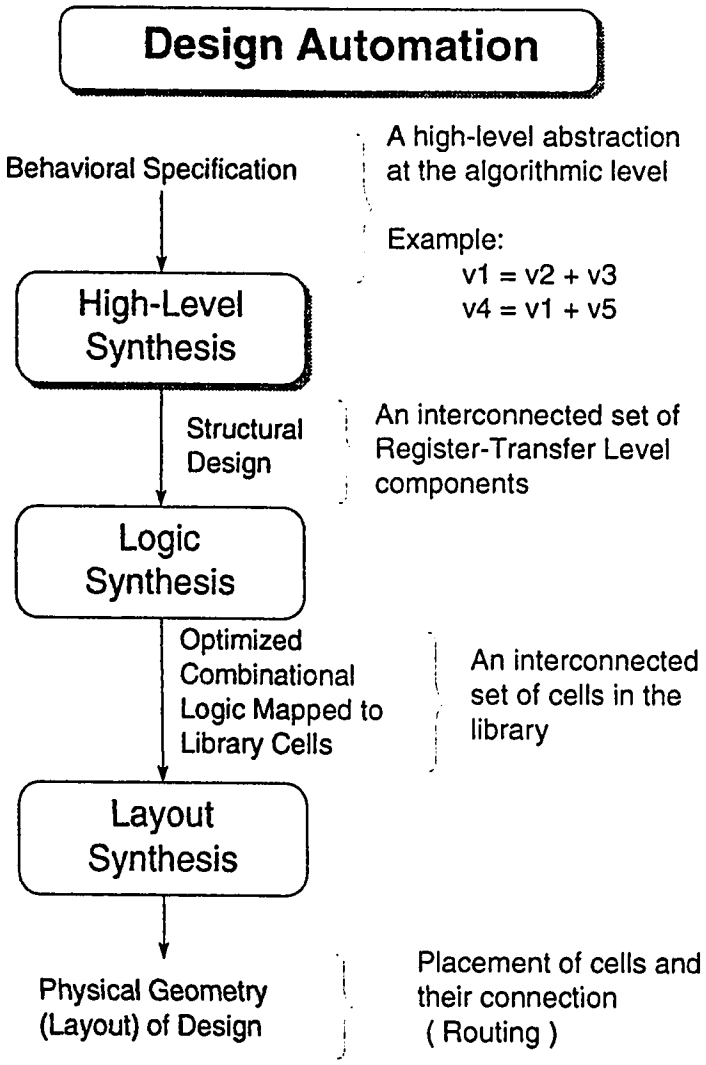


Figure 1.2: Relationship of HLS with logic and layout synthesis.

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 IF. This IF is then used as input to the scheduling and allocation phases.

Operation *scheduling* and hardware *allocation* are the two most important phases in the synthesis of circuits from behavioral descriptions. While *scheduling* distributes the execution of operations throughout time steps, *allocation* assigns hardware to operations and values. More specifically, the task of scheduling is to assign each CDFG node to a control step in such a way that all data/control flow dependencies are satisfied. On the other hand, the task of allocation is to assign hardware cells to CDFG nodes and edges in such a way that the resulting circuit can implement the specified behavior without resource conflicts. Allocation of hardware cells include functional unit allocation, register allocation and bus allocation.

Scheduling and allocation are closely interrelated, but are usually dealt with separately because of the complexity involved. There has been disagreement as to which should come first. Some techniques take scheduling first and depend on estimates of the required hardware. The others take allocation first, subject to given constraints, and then schedule, taking into account the already given hardware.

The *objective of scheduling* may be to minimize the total number of control steps

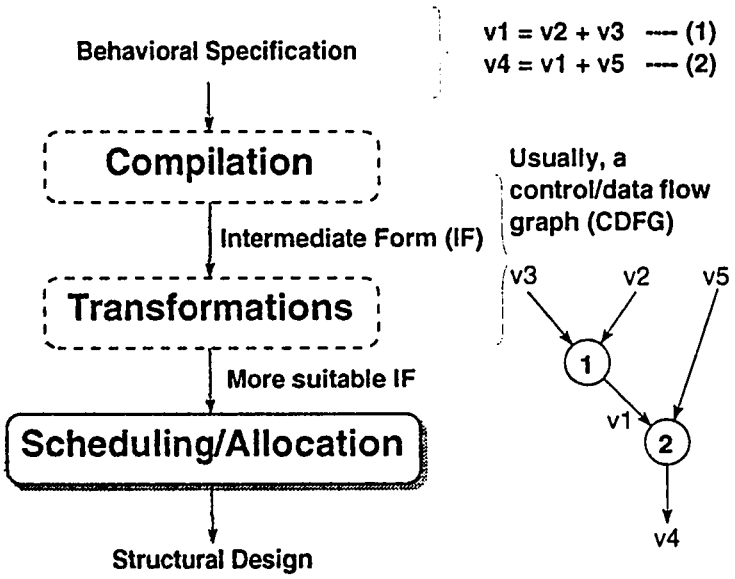


Figure 1.3: Steps involved in HLS.

or to minimize the area of hardware or a combination of both of these. The *objective of allocation* is to minimize the amount of hardware cost which includes the cost of functional units, registers, buses and interconnections.

Optimal scheduling and allocation are both NP-complete problems. Therefore, heuristic algorithms are generally used to solve them. As mentioned earlier, scheduling and allocation are highly inter-dependent. Optimizing them separately may give suboptimal results because the possibility that the best designs (in terms of overall cost) may require suboptimum schedules and/or allocations may not be considered. Thus, one can also combine scheduling and allocation in an attempt to optimize a cost function that includes both the number of control steps and the hardware. In this case, allocation may mean minimization of number of functional units, registers, buses and interconnect costs rather than actual allocation which is done after the optimization process. This is usually the case since computation for allocation is costly during optimization. The *objective function* involved in optimization process for scheduling and allocation is usually complex.

Several optimization techniques can be used for this purpose such as *simulated annealing* and *integer programming.* **Genetic algorithm** (GA) is another promising global optimization technique [Gol89]. It works by emulating the natural process of evolution as a mean of progressing toward the optimum. The algorithm starts with a *population* which consists of several solutions to the optimization problem. A member of population is called an *individual.* A *fitness* value is associated with

each individual. Each solution in the population or an individual is encoded as a string of symbols. These symbols are known as *genes* and the solution string is called a *chromosome.* The values taken by genes are called *alleles.* Several pair of individuals (*parents*) in the population *mate* to produce *offsprings* by applying the genetic operator *crossover.* Selection of parents is done by repeated use of a *choice* function. A number of indivi duals and offsprings are passed to a new *generation* such that the number of individuals in the new population is the same as old population. A *selection* function determines which strings form the population in the next generation. Each surviving string undergoes *mutation* with probability called *mutation rate* and *inversion* with probability known as *inversion rate.*

In evolution, the problem each individual faces is one of searching for beneficial adaptations to a complicated and changing environment. The *knowledge* that each individual has gained is embodied in the makeup of its chromosome. The operations that alter this chromosomal makeup are applied when parents reproduce. Random mutation provides background variation and occasionally introduces beneficial material into an individual's chromosome. Inversion alters the location of genes on a chromosome, allowing genes that are coadapted to cluster on a chromosome, increasing their probability of moving together during crossover. Crossover exchanges corresponding genetic material from two parent chromosomes, allowing beneficial genes on different parents to be combined in their offspring [Dav87].

The genetic algorithm differs from the other stochastic techniques by being able

to encode and exploit past information efficiently during a search. This learning capability provides the genetic algorithm with a guiding capability for searching efficiently through a complex multi-dimensional search space. The key steps in the application of genetic algorithm include an appropriate string encoding or chromosomal representation, a way to create an initial population of solutions, and an effective crossover operator.

1.1 Synthesis Process

A simplified example of synthesis process is illustrated in Figure 1.4 [PK89]. The behavioral description is compiled into an IF which is then transformed into optimized IF. The CDFG shown corresponds to this optimized IF. Note that common subexpression $A + B$ is computed only once. Scheduled CDFG shows one of the possible schedules for given CDFG. The small circles corresponds to registers. It is assumed that input values A, B, C, D, and E should be available even after the computation of output values F and G. Therefore, the corresponding registers can not be used. Note that a temporary register X is used to store $A + B$, whereas the register for G is temporarily used to store $C + D$. Lower part of the figure shows the data path for this schedule after the allocation of functional units, registers and buses is performed.

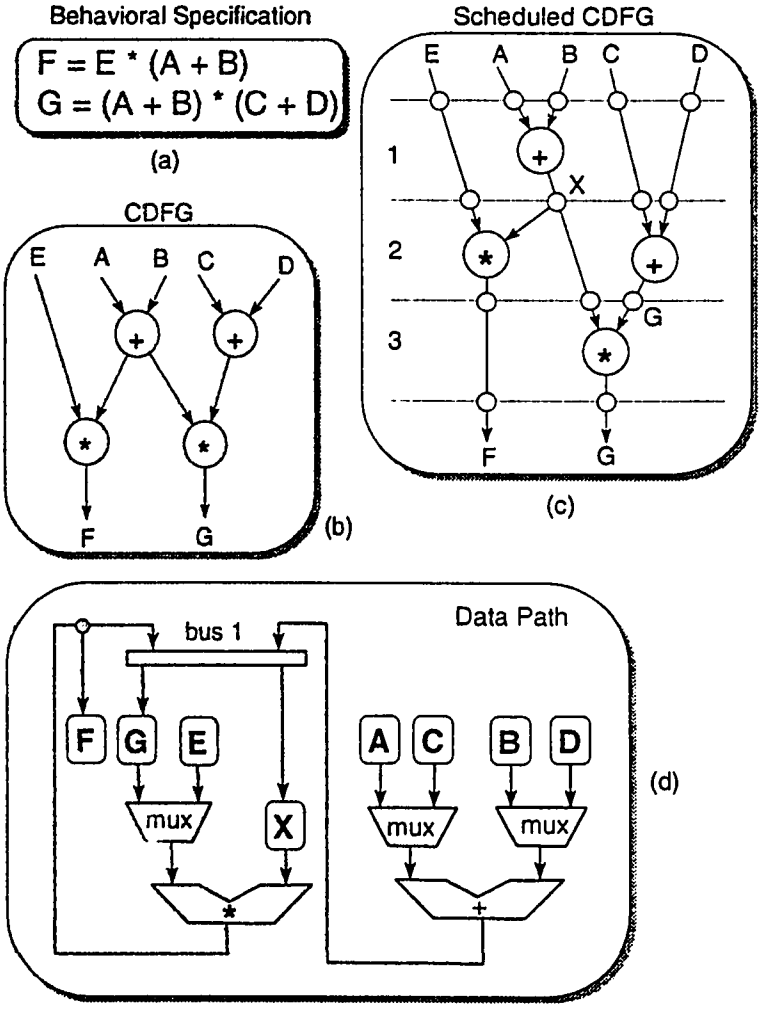
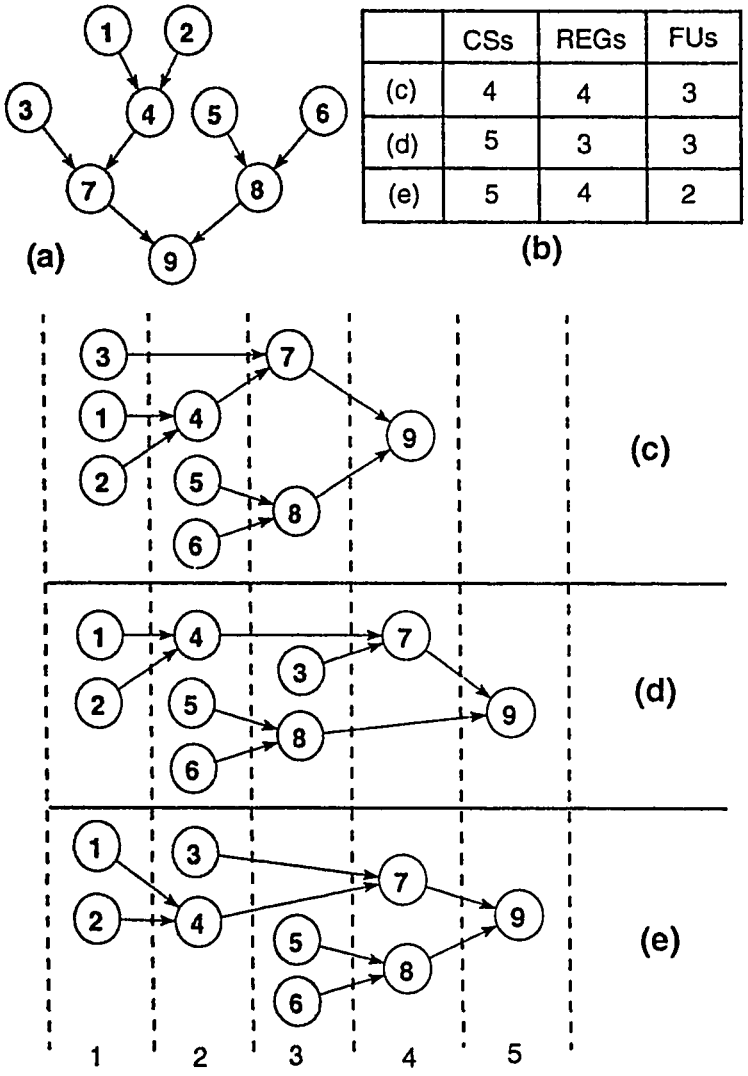


Figure 1.4: Example of synthesis process.

1.2 Problem Illustration

As the transformation of a behavioral specification into a structural design, with limited resources, is an NP-hard problem, one tries to simplify the search for efficient approximations by subdividing the general problem into subproblems of scheduling and allocation. As stated earlier, this approach gives suboptimal results. This is because the two subproblems are highly inter-dependent. The strong inter-dependence between the allocation of resources and the scheduling of operations into time steps can be illustrated with the help of a simple example of Figure 1.5 [CW91]. The CDFG is shown in Figure 1.5(a). Figure 1.5(b) gives the number of control steps, the minimum number of registers needed and the minimum number of functional units required for the different schedules of Figure 1.5(c), (d), and (e). The example makes it clear how the number of required control steps is mutually dependent on the number of available registers and functional units. It also shows how e very partial problem is inter-dependent within the allocation, namely, register allocation and functional unit allocation.

The effect of functional unit allocation and register allocation on interconnection cost is illustrated with the help of the example of Figure 1.6. Figure 1.6(a) shows a simple CDFG with three additions and one multiplication for the behavioral specification given in Figure 1.6(b). There are seven variables involved. The number of control steps for which the variable is active is called its *lifetime*. The lifetime anal-



	CSs	REGs	FUs
(c)	4	4	3
(d)	5	3	3
(e)	5	4	2

Figure 1.5: Inter-dependence between resource allocation and scheduling.

ysis of variables used is shown in Figure 1.6(c). Variables with disjoint lifetimes can be stored in the same register. Thus we can form groups of $(v1, v6)$ or $(v3, v6)$ and $(v4, v7)$ or $(v5. v7)$. The register assignments are shown in Figure 1.6(d). Variables $v1$ and $v6$ are grouped into register **R1** and $v5$ and $v7$ into **R5**. The data path using this assignment is shown in Figure 1.6(e). It can be seen that operation $a3$ can be assigned to adder 1 or 2. No more interconnections are needed if it is assigned to adder 2, as is don e in Figure 1.6(e). Had it been assigned to adder 1, an extra interconnection from register **R2** to adder 1 would have been necessary which will also necessitate a multiplexer at one of the inputs of adder 1. It can also be seen that if variable $v6$ were grouped with $v3$, an extra interconnection and a multiplexer can not be avoided whether we assign operation $a3$ to adder 1 or 2. Grouping $v7$ with $v5$ does not require any extra interconnection as both are produced by adder 2. This clearly illustrates the effect of functional unit allocation and register allocation on the interconnection cost.

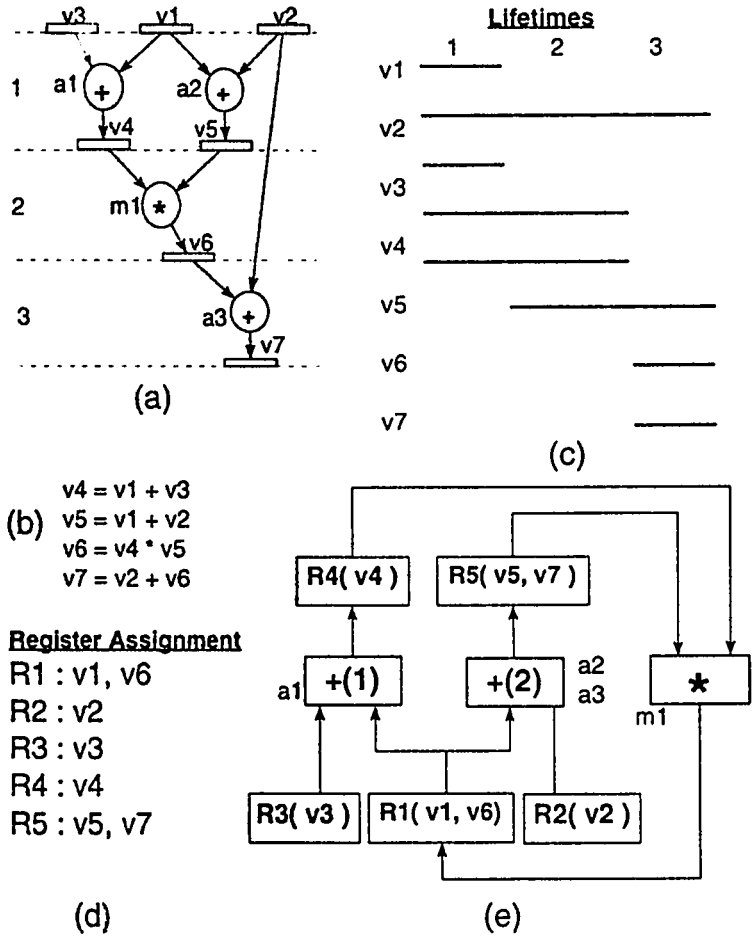


Figure 1.6: Effect of functional unit and register allocation on interconnection cost.

Chapter 2

Literature Survey

2.1 Scheduling Techniques

Existing **scheduling algorithms** can be classified into two groups: the *iterative/constructive* group and the *transformational* group [MPC90]. Iterative/constructive scheduling algorithms successively schedule operations until complete schedules are constructed. On the other hand, transformational scheduling algorithms start with default schedules which are typically maximally parallel or serial designs, and then repeatedly apply semantic preserving transformations to improve the initial schedules.

The first approach to scheduling in high-level synthesis was probably the exhaustive search. Since then, many scheduling algorithms for high-level synthesis have been proposed. Davidson et. al. [DLSM81] discuss exhaustive search using branch-

and-bound techniques, as-soon-as-possible (ASAP) scheduling, list 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 depth-first order. Examples of systems that employ ASAP are Flamel [Tri87] and [HP78]. An example of CDFG with its ASAP schedule under the constraint of one adder and one multiplier is shown in Figure 2.1.

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 that control step. This process is then repeated on the next control step, until the entire design is scheduled [WC91]. List scheduling is illustrated in Figure 2.2 [MPC90]. The priority function is the path length from the node to the end of block and is given in parenthesis in Figure 2.2(a), and the list schedule is shown in Figure 2.2(b). Variations of list scheduling are used in many high-level synthesis systems, for example, CMU's System Architect's Workbench [TLW+90] and SLICER [PG87].

A more complex scheduling method is *force-directed scheduling* [PK89], which

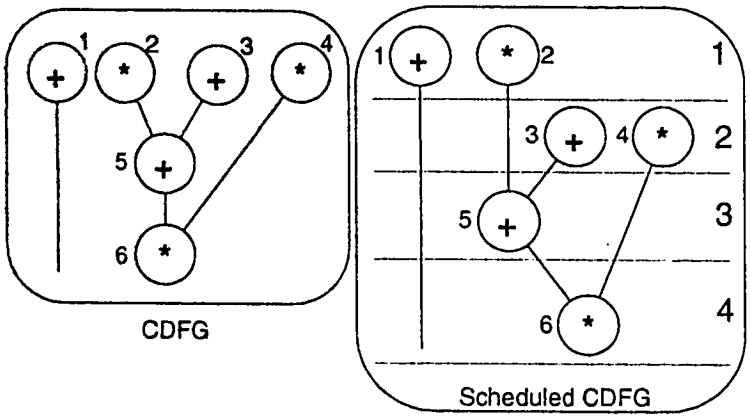


Figure 2.1: Example of ASAP scheduling.

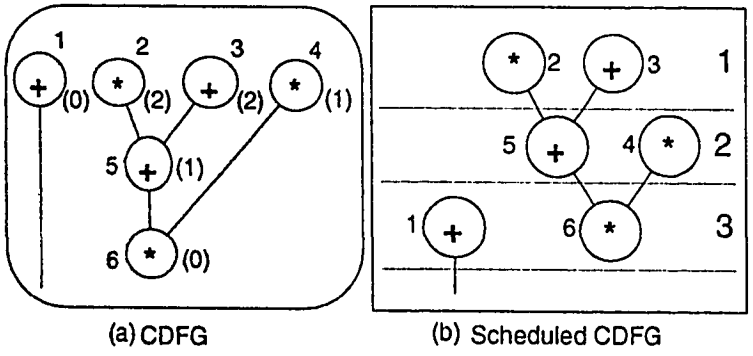


Figure 2.2: Example of list scheduling.

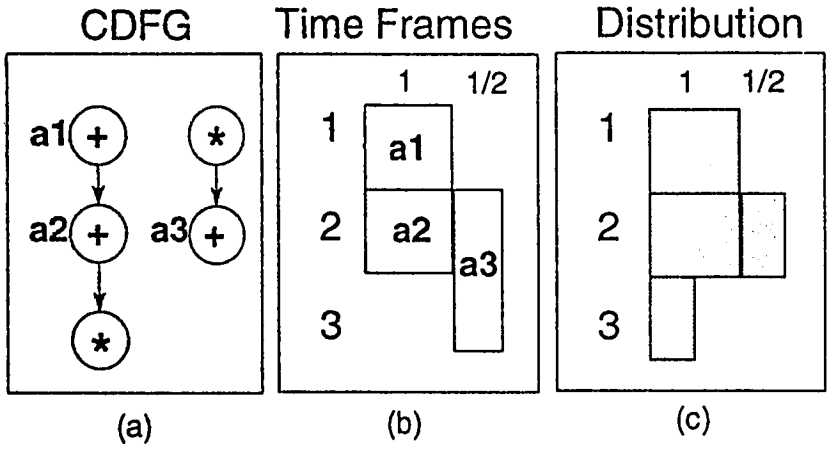
uses a global criterion that indicates how crowded a control step is compared with others to decide where to schedule an operation. The probabilities of operations being in a given control step can be calculated by using *mobility*, which is the difference between the ASAP and as-late-as-possible (ALAP) schedules. For example, if an operation can be scheduled in three possible steps, it has 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 many operations will probably be executed in a control step and hence, 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 force-directed scheduling (FDS). FDS is illustrated in Figure 2.3 [MPC90]. A CDFG with three add operations labeled as $a1$, $a2$, and $a3$ is shown in Figure 2.3(a). Figure 2.3(b) shows *time frames* for add operations, that is, the probability of each operation being in a given control step. Distribution is shown in Figure 2.3(c). Calculation of force involved in assigning $a3$ to control

step 2 is shown in Figure 2.3(d). As we can see in Figure 2.3(c) that 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 group. Another approach to scheduling by transformation is to use heuristics to guide the process. Starting with an initial schedule, transformations are chosen that promise to move the design closer to the given constraints or to optimize the objective. Examples are *control-step merging* and *control-step splitting* [Cam90]. 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 operation to a single control step and then divide this control step until we have no constraint violations.

2.2 Allocation Techniques

The **allocation techniques** can also be classified into two types: *iterative/constructive*, and *global* [MPC90]. Iterative/constructive techniques assign elements (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 an operation, value or interconnection to be assigned, make the assignment, and then iterate until all assignments are made.



Force involved in assigning a3 to step 2:

3/2 x 1/2 + 1/2 x (-1/2) = 1/2

DG value for step **2**

Change in a3's probability for step **2**

Similar values for step **3**

Positive Force indicates that a3 should *not* go into step 2

(d)

Figure 2.3: Example of FDS.

These techniques generally look at restricted window in the search space, and therefore are more time efficient, but are less likely to find optimal solutions. Examples of systems using iterative/constructive techniques are [HP78, HT83]. The STAR allocation system of [TH92] uses an iterative improvement technique. It uses a *rip-up and 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.

Global allocation techniques include graph theoretic formulation, branch-and-bound algorithms, and mathematical programming techniques. Facet [TS86] uses 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 of the elements in such a set can share the same hardware without conflict. Graph theoretic approach is illustrated in Figure 2.4. A schedule with four add operations is shown in Figure 2.4(a). Its compatibility graph is shown in Figure 2.4(b). The clique (completely connected subgraph) indicates that the three operations can share a single adder. An example of a system using branch-and-bound technique is SPLICER [Pan88]. For-

mulations of allocation 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 1 if the assignment is made and 0 if it is not. Constraints must be formulated that guarantee that each operation must be assigned to one and only one hardware element, and so on. The objective then is to find a valid solution that minimizes some cost function. This is done by Hafer and Parker on a small example [HP83].

Some approaches have formulated combined scheduling and allocation as an optimization problem to be solved by general optimization techniques. Among the optimization techniques used for this purpose are simulated annealing [DN89] and integer programming [BM89]. *Simulated annealing* (SA) approach is transformational. The scheduling and allocation is formulated as a two-dimensional placement problem of microinstructions in space and time. New states are generated during annealing process by interchanging/displacing code operations (Figure 2.5). This formulation allows simultaneous cost-constrained allocation of registers, functional units, and interconnect while trading off hardware cost against execution speed. The *integer programming* approach uses a constructive technique. A set of operations called candidate operations are prepared based on the data dependency and operators availability. These operations are bound to the available operators optimally, that is, to minimize extra interconnections. The values generated by the scheduled operations are bound to the available registers, again with a view to minimize in-

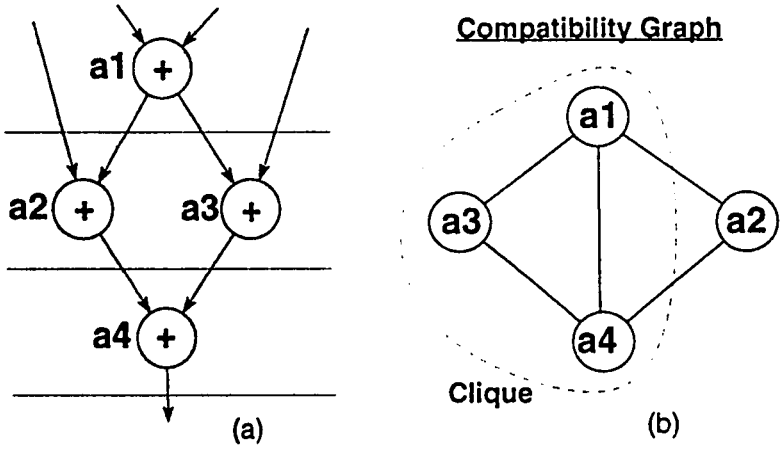


Figure 2.4: Allocation: Graph theoretic approach example.

terconnections. This is followed by updating the structure based on the current bindings. The updated structure is used in subsequent iterations.

Another optimization technique, *simulated evolution*, is used in [LM93]. Simulated evolution based synthesis explores the design space by repeatedly ripping up (like STAR system) parts of a design in a probabilistic manner, and then reconstructing these parts using application specific heuristics. But this approach solves scheduling and allocation tasks as separate problems. Cloutier and Thomas [CT90] have extended FDS in an attempt to combine scheduling and allocation. Their technique takes the same global view of the scheduling problem that is found in force-directed scheduling. Costs of register allocation and multiplexer inputs are not considered during scheduling and mapping is done only for functional units.

High-level synthesis is a formidable task. There are several issues involved to obtain high quality designs. Despite these difficulties, high-level synthesis systems are now emerging as important tools in digital design. Excellent surveys on high-level synthesis systems appear in [MPC88, Cam90, MPC90, WC91, CW91].

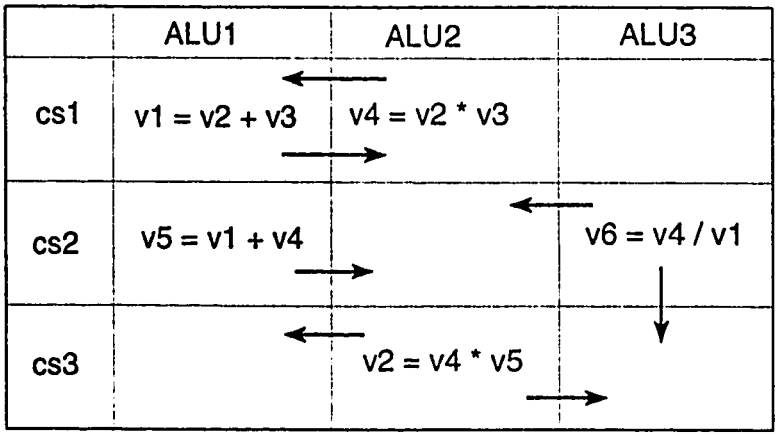


Figure 2.5: New states are generated in SA by interchanging/displacing code operations.

Chapter 3

Genetic Algorithm and Tabu Search

This chapter provides an introduction to the two optimization techniques employed in this research - namely, genetic algorithm and tabu search.

3.1 Genetic Algorithm

Genetic algorithm has its roots in the process of natural evolution. In the early 1970s, John Holland incorporated the features of natural evolution to yield a technique for solving difficult problems which he later named *genetic algorithm.* The algorithm manipulates bit strings which he called *chromosomes.* A simulated evolution is carried out on the population of such chromosomes to find better chromosomes.

Like in nature, the algorithm knows nothing about the type of problem it is solving. There are two mechanisms which link the genetic algorithm with the problem. One is the *encoding* of solutions to the problem with chromosomes and the other is the *evaluation function* that measures the merit of any chromosome in the context of the problem.

No single encoding works best for all problems. Devising a good encoding is an important step in attacking a problem using genetic algorithm. Evaluation function plays the same role in genetic algorithms that the environment plays in natural evolution. The interaction of an individual with its environment provides a measure of its fitness, and the interaction of a chromosome with an evaluation function provides a measure of fitness that the genetic algorithm uses when carrying out reproduction [Dav91]. In the following paragraphs we provide a step by step treatment of the functioning of the genetic algorithm.

3.1.1 Population

Unlike most of the optimization techniques, genetic algorithm works on a number of encoded solutions (chromosomes) rather than on a single solution. An important decision to be made in this respect is the size of the population. The most effective population size is dependent on the problem being solved, the representation used, and the operators manipulating the representation. The initial population consisting of feasible solutions is generated randomly. Constructive techniques are also reported

for generating initial population.

3.1.2 Crossover

Parents are selected from the population at random for mating. Each pair of parent undergoes crossover in which offsprings inherit parent's characteristics. Crossover operator distinguishes genetic algorithms from all other optimization algorithms. It acts as a critical accelerator of the search process and combines *building blocks* of good solutions from diverse chromosomes. In case of a binary chromosome, a building block consists of 1, 0, and # (don't care). We say that a chromosome has a building block if it matches the 1's and 0's on the building block exactly. These building blocks were named *schema* by Holland. He concluded that genetic algorithms manipulate *schemata* when they run. If the reproduction scheme makes reproduction chances proportional to chromosome fitness, then the relative increase or decrease of a schema in the next generation can be predicted. Holland's *schema theorem* says that a schema occurring in chromosomes with above-average fitness evaluations wil l tend to occur more frequently in the next generation, and one occurring in chromosomes with below-average fitness evaluations will tend to occur less frequently.

Holland described this feature of GAs as *intrinsic parallelism.* Each schema denotes a hyperplane. Intrinsic parallelism means that search effort is allocated simultaneously in many hyperplanes (regions) of the search space. The crossover

operator is effective as long as the population has diverse members and representative samples of different building blocks of good solutions. Thus crossover is a high performance search operator that spreads good schemata present in different members of the initial population.

The simplest type of crossover is *one-point crossover.* A random location is generated over the length of the two parent chromosomes as a crossing point. The left part of one parent is combined with right part of other parent to generate offsprings as shown in Figure 3.1. There are various other types of crossovers - namely, order crossover, cycle crossover, partially mapped crossover (PMX), etc. How often crossover is applied depends on *crossover rate* and *crossover probability.* Crossover rate determines number of times crossover operator should be attempted on the population and crossover probability is the probability with which crossover is applied.

3.1.3 Mutation

Mutation is a device for reintroducing diversity into the population. It plays a secondary role in GA. For binary representations, it is the random alteration of a single position. The *order* of schema is the number of non-# symbols it contains. Its *length* is the distance from the first to the last non-# position. Thus, the length of #1#0#1 is four, and its order is three. *Schema theorem* can now be stated as: short, low-order, above-average schemata receive exponentially increasing trials in

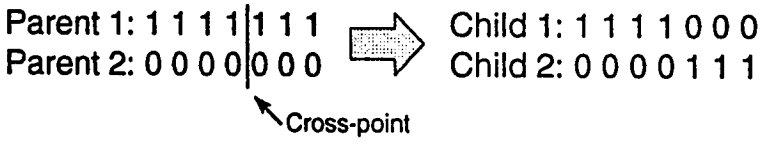


Figure 3.1: Illustration of one-point crossover.

subsequent generations.

3.1.4 Selection

After the application of crossover, we have the old population and a number of offsprings. For a fixed population size GA, we have to select individuals among these for the next generation. This selection is usually based on fitness values, but it may take different forms. The simplest is the biased roulette wheel in which each individual in the population has a roulette wheel slot sized in proportion to its fitness. A simple spin of the roulette wheel yields an offspring. This scheme is a variation of *stochastic sampling*.

Various other selection alternatives have appeared in the literature. *Deterministic sampling* is a scheme where the probabilities of selection $pselect_i$ are calculated as $pselect_i = f_i / \sum f$, where f is fitness. Then the expected number of times each string e_i should be selected is calculated as $e_i = pselect_i.n$, where n is population size. Each string is allocated samples according to the integer part of the e_i values. Strings are sorted according to their fractional part. The remainder of the strings needed to fill the population are drawn from the top of the sorted list. *Stochastic remainder sampling* methods are somewhat similar to deterministic sampling. Expected individual count values are calculated as before and each string is allocated samples according to integer part. In one variation of stochastic remainder sampling with replacement, the fractional parts of the expected number values are used to

calculate weights in a roulette wheel selection procedure tha t is then used to fill the remaining population slots. In another variation, the fractional parts of the expected number values are treated as probabilities. One by one, weighted coin tosses (Bernoulli trials) are performed using the fractional parts as success probabilities. *Stochastic tournament* is a procedure where selection probabilities are calculated normally and successive pairs of individuals are drawn using roulette wheel selection. After drawing a pair, the string with higher fitness is declared the winner, inserted in the new population, and another pair is drawn. This process continues until the population is full [Gol89].

The overall picture of a GA is depicted in Figure 3.2. Encoding is devised for a problem in hand. A population of encoded solutions is created. Fitness of each solution is found using evaluation function. Two parents are selected for crossover which results in two offsprings. Offsprings are then mutated with a very low probability. After the crossover is applied a specified number of times, we get a population of offsprings along with the old population of size n as shown in Figure 3.3. A selection function is used to select individuals from these two populations to get the new population of size n. The above steps are then repeated for specified number of generations. The best solution in the final population is the result of GA.

To further materialize the concepts, the pseudo code of a genetic algorithm is given in Figure 3.4. The algorithm continues for a fixed number of generations. Since the probability of crossover is 1, crossover rate specifies how many times the

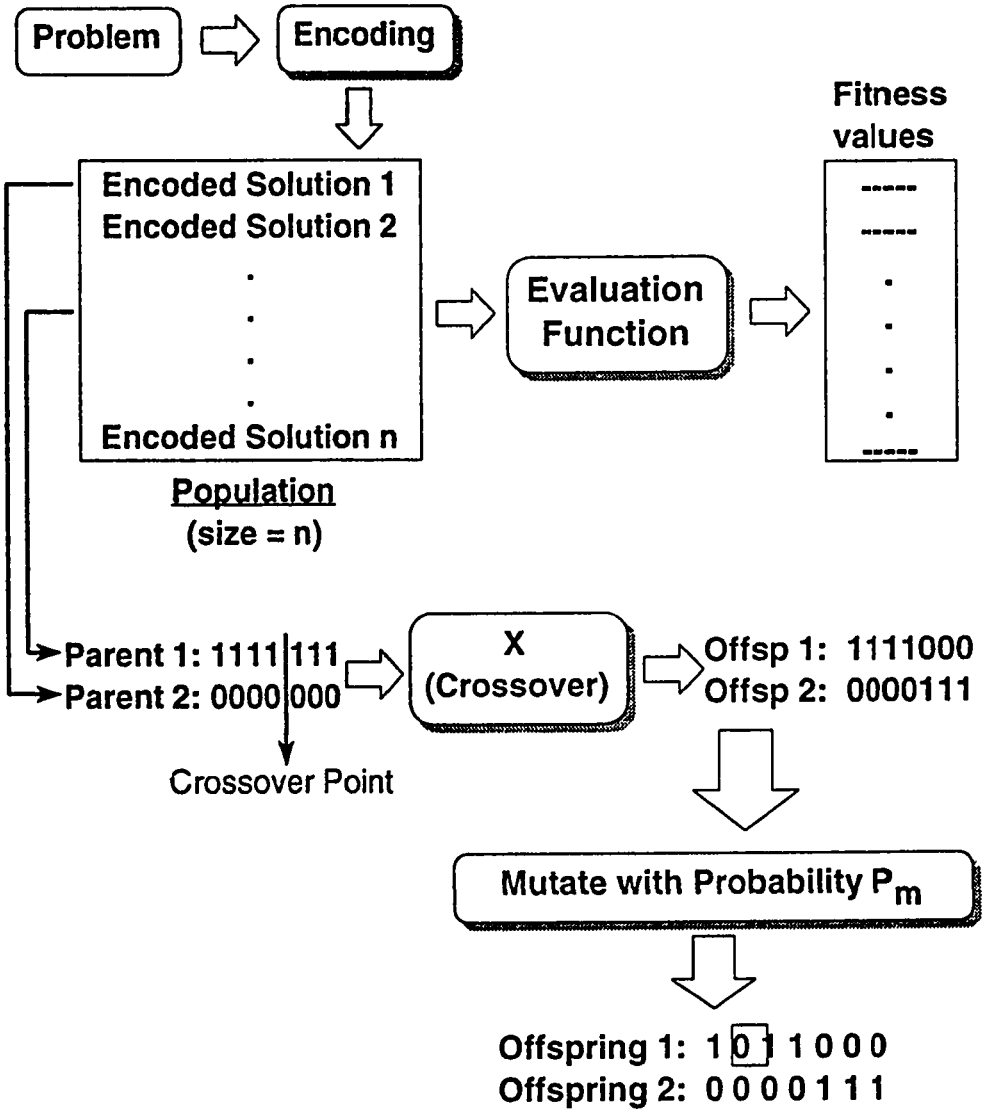


Figure 3.2: Genetic algorithm: Encoding, evaluation function, crossover and mutation.

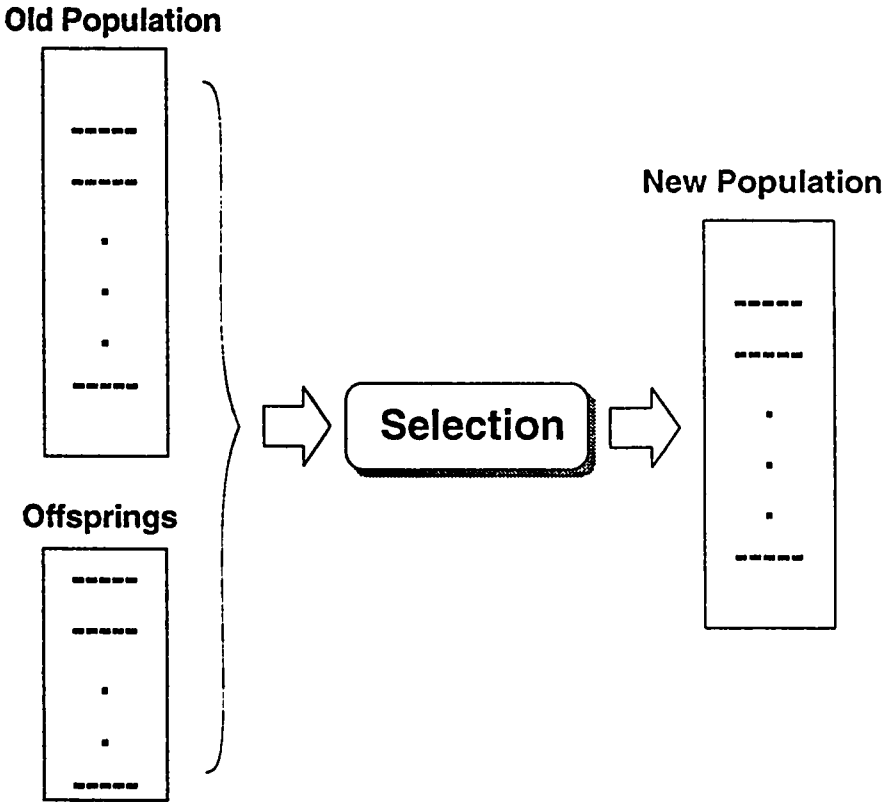


Figure 3.3: Genetic algorithm: Selection.

crossover has to be applied. Selection procedure specifies the new population for the next generation. Offsprings are mutated with probability P_m.

An excellent source on **genetic algorithms** is [Gol89]. Genetic algorithms have been successfully applied to a number of problems. The long list includes VLSI cell placement [CP87], circuit partitioning [JC92], floorplanning in VLSI [CHMR91], scheduling of task graphs [BS94b], multiprocessor scheduling [HHA90], job-shop scheduling [BD90], Steiner trees [HMS89], optimization of two-bit decoder PLAs [BS94a], bin-packing [FD92] and travelling salesmen problem [WSF89]. This partial list is a good indicator of the applicability of genetic algorithms to diverse problems. Although it is simple to apply genetic algorithms to problems with unconstrained objective functions, it can be seen from the above list that genetic algorithms have also been successfully applied to constrained optimization problems.

3.2 Tabu Search

A general iterative technique, called *tabu search* (TS), was proposed by Glover [Glo77, Glo89, Glo90b] for finding good solutions to combinatorial optimization problems. This technique is conceptually simple and elegant. It is a higher level heuristic which can be superimposed on any procedure which works by making moves to go from one trial solution to another.

Like simulated annealing, TS does not resort to pure randomization to conquer

1. Create Initial population
2. Evaluate population
3. For number of generations Do
4. For Pop. size times crossover rate Do
5. Choose Parents
6. Perform Crossover
7. With probability P_m mutate offspring
8. Evaluate offspring
9. End 4
10. Selection
11. End 3.

Figure 3.4: Pseudo code for GA.

intractability nor does it take the conservative approach that a proper rate of descent will lead us to a good local optimum which may be close to global one. TS uses a flexible attribute-based memory structures to exploit historical search information more thoroughly than by techniques using rigid memory structures (such as branch and bound and A* search) or by memoryless systems (such as simulated annealing). Using these memory structures, TS employs a mechanism of control which constrains and frees the search process. These corresponds to *tabu restrictions* and *aspiration criteria.* TS takes the aggressive exploration approach which seeks to make the best move possible subject to available choices, performance, and certain constraints.

3.2.1 Tabu Restrictions

TS goes from one trial solution to another by making moves. It makes several candidate moves and selects the move producing the best solution among all candidate moves for current iteration. This best candidate solution may not improve the current solution. With this strategy, it is possible to reach the local optimum, ascend, and then come back to local optimum in case of a minimization problem. Thus there is a possibility of cycling. Tabu restriction is a device to avoid such cycling by making selected attributes of these moves tabu (forbidden) to avoid move reversals. Tabu restrictions allow the search to go beyond the points of local optimality while still making best possible move in each iteration. Selecting the best move (which may or may not improve the current solution) is based on the supposition that good

moves are more likely to reach the optimal or near-optimal solutions. The set of admissible moves form a *candidate list.* TS selects the best move from the candidate list. Candidat e list size is a trade-off between quality and performance.

Tabu restrictions are enforced by a *tabu list* which stores the move attributes to avoid move reversals. Tabu list has an associated size. It can be visualized as a window on accepted moves. The moves which tend to undo moves within this window are forbidden (Figure 3.5). Good performance is achieved with tabu list sizes from 5 to 12. Magic number 7 is also used in many applications. Some experimentation is also reported which uncovered applications where preferred tabu list sizes lie in intervals related to problem dimension rather than linked to the magic number 7.

3.2.2 Aspiration Criteria

Aspiration level component of TS introduces diversification in the search. It temporarily overrides the tabu status if the move is sufficiently good. If a move is made tabu in iteration i and its reversal comes in iteration j, where $i < j + c$, then it is possible that the reverse move may take the search into a new region because of the effects of c intermediate moves. Aspiration criterion must make sure that reversal is leading to a solution which is better and is not the same as the previous one otherwise cycling can occur.

The simplest aspiration criterion is to override the tabu status if the reversal

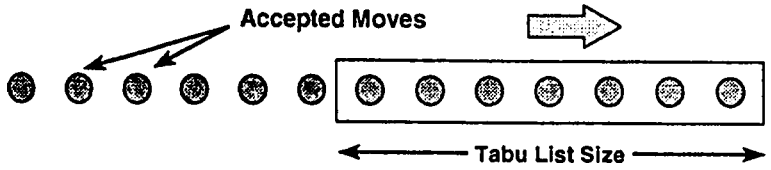


Figure 3.5: Tabu list can be visualized as a window over accepted moves.

produces the solution better than the best obtained thus far. Another approach is to use the same attribute of the move which is used to identify the tabu status and associate an aspiration level value with it. The reversal has to do better than this historical aspiration level. It is found useful in some applications to give aspiration level a tenure that parallels the tenure of the tabu list. This means that aspiration level of the selected attribute is updated whenever that move is made tabu and whenever aspiration level criterion is passed.

3.2.3 Algorithmic Description

A simplified description of TS is illustrated in Figure 3.6. Best solution is the best one found so far. Initially the current solution is the best solution. Copies of current solution are perturbed with moves to get a set of new solutions. The best among these is selected and if it is not tabu then it becomes the current solution, otherwise its aspiration criterion is checked. If it passes the aspiration criterion then it becomes the current solution, otherwise moves are regenerated to get another set of new solutions. If the current solution is better than the best so far then the best solution is updated.

An algorithmic description of a simple implementation of TS is shown in Figure 3.7. TS starts from an initial feasible solution s (current solution) in search space X. A neighborhood $N(s)$ is defined for each s. A sample of neighbor solutions $S(s) \in N(s)$ is generated. Then the best $s' \in S(s)$ generated is chosen and

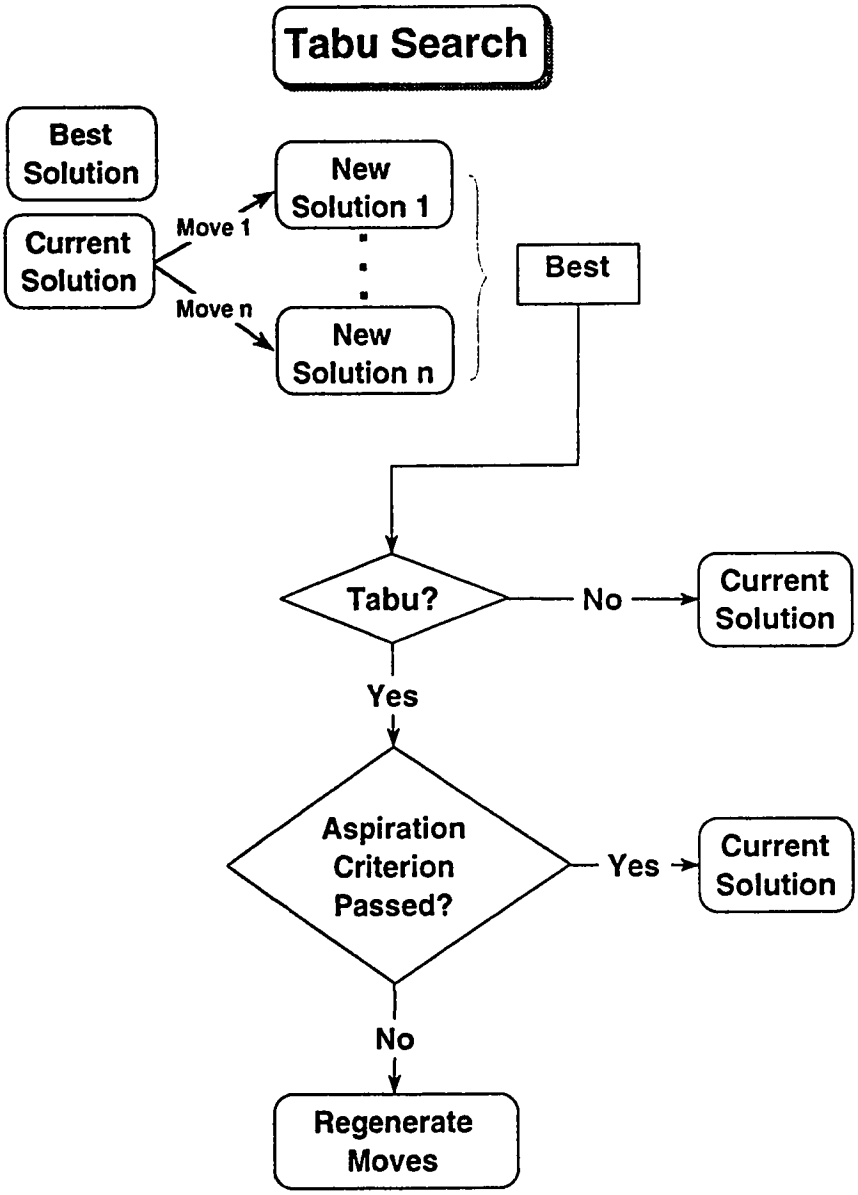


Figure 3.6: Tabu search illustration.

move from s to s' made. The move to s' is made even if s' is worse than s, that is $c(s') > c(s)$, where c is the cost. Tabu list T consists of the selected attributes of last k moves for a given k, where k is the tabu list size. A move to s' is then not allowed if s' is in T. Whenever a move is made, it is introduced into T and aspiration level (AL) is updated. If the best admissible move is tabu then the aspiration level check is performed. If $c(s') < AL$, then the move is accepted, otherwise a new set of neighbor solutions is generated and above steps are repeated. Whenever a move is accepted the iteration number is incremented. Since aspiration level is updated on each acceptance of a move, a tabu move has to do better than the aspiration level recorded when that move is made tabu. Another approach is to make a *candidate list* of admissible neighborhood solutions and then accept the best among these. Candidate solutions are either non-tabu or pass the aspiration criterion. Note that TS in Figure 3.7 continues for a fixed number of iterations. Another approach is to continue until no improvement on the best solution is obtained over a fixed number of iterations.

TS has also proven itself to be very useful in providing good solutions for many NP-hard problems in a reasonable amount of time. Examples include graph coloring [HdW87], graph partitioning [LC91], VLSI placement [SV92], circuit partitioning [AV93], maximum independent set problem [FHdW90], etc.

X :Set of feasible solutions.
s :Current solution.
s' :Best admissible solution.
c :Objective function.
$N(s)$:Neighborhood of $s \in X$.
$S(s)$:Sample of neighborhood solutions.
T :Tabu List.
AL :Aspiration Level.

```
1.  Start with an initial feasible solution s ∈ X.
2.  Initialize tabu lists and aspiration level.
3.  FOR fixed number of iterations DO
4.      Generate neighbor solutions S(s) ∈ N(s).
5.      Find best s' ∈ S(s).
6.      IF move s' to s is not in T THEN
7.          Accept move and update best solution.
8.          Update tabu list and aspiration level.
9.          Increment iteration number.
10.     ELSE
11.         IF c(s') < AL THEN
12.             Accept move and update best solution.
13.             Update tabu list and aspiration level.
14.             Increment iteration number.
15.         ENDIF
16.     ENDIF
17. ENDFOR
```

Figure 3.7: Algorithmic description of Tabu Search (TS).

Chapter 4

Scheduling and Allocation using Genetic Algorithm

Scheduling refers to the assignment of control steps to operations and allocation is the assignment of hardware to operations, variables and data transfers. The hardware consists of mainly functional units (adders, multipliers, etc.), registers, buses, and interconnection between them. In order to clarify the approach adopted for scheduling, we will first see the possible variations. Generally one can provide the following scheduling facilities:

- Scheduling supporting operation chaining.
- Scheduling supporting multicycle operations with or without pipelined functional units.

Scheduling two operations in the same control step is referred to as chaining. The clock cycle has to be large enough to allow chained multiple operations in the same control step. Given a fixed step size one can schedule multiple operations in the same control step. In the other facility, the clock cycle is equal to the fastest available functional unit. In this case there may be operations that will take multiple control steps. One can take advantage of pipelined functional units in this case. In a two stage pipelined functional unit, for example, the first stage of next operation can be started while the second stage of the previous one is being processed such that there is an execution overlap between different stages of two operations. The technique described in this chapter supports multicycle operations. It also provides facility to schedule with non-pipelined as well as pipelined functional units.

This chapter describes a unique approach to scheduling and allocation problem in high-level synthesis using genetic algorithm (GA). This approach is different from a previous attempt using GA [WGH90] in many respects. The contributions include: a new chromosomal representation for scheduling and two subproblems of allocation; and two novel crossover operators to generate legal schedules.

4.1 Cost Function

In order to formulate scheduling and allocation as an optimization problem, a suitable cost function is required. The optimization technique will then attempt to

optimize the value of this function. Since we want to optimize scheduling and allocation tasks jointly we need to incorporate both time related and hardware related terms in our cost function. The cost function C that will be optimized by the genetic algorithm is given and explained below:

$$C = W_{cs} * N_{cs} + W_{reg} * N_{reg} + W_{bus} * N_{bus} + W_{fu} * N_{fu} + W_{ic} * N_{ic} \qquad (4.1)$$

where,

W_{cs} weight assigned to each control step

W_{reg} weight assigned to each register

W_{bus} weight assigned to each bus

W_{fu} weight assigned to each functional unit

W_{ic} weight assigned to each interconnection

N_{cs} number of control steps

N_{reg} number of registers

N_{bus} number of buses

N_{fu} number of functional units

N_{ic} number of interconnections.

During the optimization process the operations are assigned to control steps and functional units. Each functional unit has two inputs labeled as 1 and 2. Besides assignment of operations to control steps and functional units, variables are assigned to functional unit inputs. Constants are always assigned to the same input as it helps

in optimizing the number of interconnections. The number of registers and buses are optimized. Allocation of variables to registers and data transfers to buses is not actually made. The number of registers and buses as given by the final solution are optimal for the given schedule and these numbers can be used during data path synthesis.

The algorithm starts with a specified upper bound on the number of control steps. The number of control steps N_{cs} for any schedule is the largest control step value taken by any operation. Similarly the number of functional units N_{fu} for a given schedule is determined by the maximum number of functional units used in any control step. If we think of variables' lifetimes as segments spanning control steps then the number of registers N_{reg} is given by maximum density of these segments crossing any control step boundary. The number of buses N_{bus} are calculated by finding the maximum number of parallel data transfers in any control step. This is done by finding the number of distinct sources and number of sinks in each control step. The maximum of these values corresponds to the maximum number of parallel data transfers. Finding exact number of interconnections is not possible unless variable to register and data transfer to bus mappings are known. Good mappings for both of these are computationally expensive to find during optimization process. On the other hand, in order to have good cost function, one needs to find computationally *efficient* and good *estimation* of interconnection cost. To satisfy both these properties is difficult. The estimation function for interconnection N_{ic}

used here is based on operation to functional unit mapping and variable (involved in operation) to functional unit's input mapping. This gives three sets of variables for each functional unit. Two sets for each of its inputs and one set for the output of the functional unit. Left-edge algorithm [HS71] is applied on variables' lifetimes in each set to determine the number of multiplexer inputs required. This provides a compromised estimation of the interconnection cost. The interconnection cost can further be optimized during data path synthesis.

Various parameters in the cost function affect each other considerably. For example, in order to reduce the number of buses, one may require to reduce the maximum number of parallel data transfers in any control step. In order to do so, number of control steps may need to be increased which may necessitate more registers to store variables until they are used. On the other hand, all this may reduce the number functional units. Thus, change in one of the factors affects other factors considerably. This simple example has overlooked many factors just to illustrate the point. For a fixed number of control steps there is a large search space to explore in order to optimize the number of registers, buses, functional units and interconnections. As the above discussion hints this search space is highly irregular.

4.2 Chromosomal Representation

Genetic algorithms work on the coding of the problem rather than on the actual problem. This coding is known as chromosomal representation. Devising a good coding is particularly necessary for better design space exploration by the genetic algorithm. A given high-level specification of the description of the circuit is compiled using *lex* and *yacc* unix utilities [MBL92]. A control data flow graph (CDFG) is then obtained from the compiled version. Any schedule should satisfy the precedence constraints implied by CDFG.

Since we want to combine scheduling and allocation into one optimization problem, the coding has to reflect this. This can be done only to a certain extent as finding an encoding for all the parameters is nearly impossible as there are too many constraints. More will be said about these constraints in the section on crossover operators. The coding that is adopted is shown in Figure 4.1. Each gene has three values - control step number, functional unit number, and the number of the functional unit input to which the left variable of the operation is assigned. The first row gives the operation number to which the above three values correspond. This coding will be manipulated by the genetic operators. It is necessary to see why this coding is good enough to optimize scheduling and allocation tasks. With this representation the three subproblems are solved completely, namely, control step assignment, functional unit assignment, and functional unit input assignment. Given

this information, the exa ct number of registers and buses can be found, whereas only a fair estimation of interconnection cost can be obtained. The chromosome in [WGH90] has operation number in a specified order and alleles corresponding to mobility values that are filled constructively. Special genes at the end of chromosome give the number of each type of functional unit.

4.3 Initial Population

Good initial population is necessary for proper functioning of genetic algorithm reported in this research. Genetic algorithms work by adopting good structures from the population to generate better individuals. In scheduling, if an operation in an optimal solution is supposed to be in a certain control step, and if it is not assigned to that control step in any member of the initial population, then the genetic search without mutation operator will not be able to produce the optimal solution. Thus initial solution should be as diverse as possible. In this implementation the members of the initial population are created by using following four scheduling schemes.

1. As Soon As Possible (ASAP) scheduling.

2. As Late As Possible (ALAP) scheduling.

3. Mobility-down variation of ASAP.

4. Mobility-up variation of ALAP.

Operation Number	1	2	3	...	...	8	9
Control Step	4	2	3	...	...	2	1
Function Unit	1	3	1	...	...	2	3
FU input	1	1	2	...	...	2	1

Figure 4.1: Chromosome.

ASAP scheduling assigns the operations in the earliest possible control steps, whereas ALAP scheduling assigns the operations in the latest possible control steps. For a given limit on control steps, mobility of each operation is calculated. The term mobility-down scheduling as used here means that operations are scheduled in ASAP manner within their mobility range taking care of the precedence constraints. For example, assume operation op_i precedes operation op_j in the CDFG. Further assume that operation op_i has mobility from control step 3 to 6 and operation op_j has mobility from control step 4 to 7. To schedule op_i a random number r is generated between 3 and 6 and op_i is assigned to control step r. To schedule operation op_j a random number is generated between $r+1$ and 7 and op_j is assigned to that control step. Note that in a CDFG, mobility is used from upper nodes to lower nodes and hence the name mobility-down. If we reverse this sequence one will get mobility-up s cheduling. In this case a random number s is generated first between 4 and 7 and op_j is assigned to that control step. Then we generate another random number between 3 and $s-1$ and op_i is assigned to that control step.

It is clear that mobility-down or up scheduling give a variety of schedules. But why do we need both of them to produce solutions to be used in initial population? The answer lies in the fact that as we perform mobility-down scheduling the operations that are to be scheduled earlier and are up on CDFG have more freedom in selecting the control step. As we go down this freedom becomes less and less. The net effect is the high possibility that later operations might never be able to utilize

their complete mobility range. To overcome this problem we incorporated mobility-up scheduling where later operations have more freedom than earlier operations on CDFG.

An improvement is made in the mobility-up scheduling to have a better initial population. As mentioned earlier we specify a control step limit to calculate the mobility. In mobility-up scheduling when the lowest operations are to be scheduled, a random number is generated between its start of mobility and control step limit. Thus we can perform mobility-up scheduling with different control step limits. This provides better solutions to be used in initial population.

Control step assignment is only one part of the chromosome. Functional units for each control step are assigned sequentially. For example the first add operation to go in control step cs_i is assigned to adder 1 and the second add operation to go in cs_i to adder 2 and so on. After the assignment is complete for all the operations, the functional unit assignments are randomly perturbed within the maximum range of that type of functional unit. For example if functional units used by (say) three operations in control step cs_i are {1,2.3} and maximum number of functional units in the given schedule are five then after perturbation the assignment may be {3,5,1}. More will be said on advantage gained by doing this when we will discuss the crossover operators.

The third part of the chromosome is the assignment of left variable to functional unit input. This is done randomly. At this point it should be mentioned that

changing the order of variables for a commutative operation is equivalent to changing the input of the functional unit to which one of the variable goes.

4.4 Choice Function

The first step to get new generation is to select parents on which genetic operators are to be applied. The selection of parents is an important step which affects the population in the new generation. Selection of fittest parents leads to premature convergence. Thus an appropriate choice function is required. This depends on how the fitness of a member of the population is calculated.

4.4.1 Fitness Calculation

Genetic algorithm works naturally on the maximization problem whereas our cost function has to minimized. Thus the cost minimization problem is converted to a fitness maximization problem as follows. The maximum cost C_{max} in the entire population is determined and each cost c_i is subtracted from this value to get the fitness f_i of individual i as follows:

$$f_i = C_{max} - c_i \tag{4.2}$$

If the choice of parents is based on this raw fitness value, a premature convergence will result. Fitness scaling is used to avoid this premature convergence. One method is *linear scaling* [Gol89]. Given fitness f_i of an individual as above the scaled fitness

value f_i' is calculated as follows:

$$f_i' = a \times f_i + b \tag{4.3}$$

where constants a and b are calculated such that averages of raw fitness and scaled fitness are equal.

Linear scaling runs into problems in later runs of the genetic algorithm when most of the fitness values are close to each other and some lethal members have very low fitness values. This leads to negative fitness values. To avoid this situation *sigma* (σ) *truncation* was proposed [Gol89]. All the fitness values are preprocessed to calculate modified fitness values f_i'' as follows:

$$f_i'' = f_i - (f_{avg} - C_{mult} \times \sigma) \tag{4.4}$$

where σ is the standard deviation of the population and C_{mult} is the multiplying constant between 1 and 3. The negative values ($f_i'' < 0$) are arbitrarily set to zero. After this truncation, linear scaling can proceed without the danger of negative results as follows:

$$f_i' = a \times f_i'' + b \tag{4.5}$$

Fitness scaling attempts to maintain the variation in the population which is necessary for further exploration of search space. Once the population consists of same type of individuals the genetic algorithm looses its ability to explore the search space until the population gains some variation by the slow process of mutation.

4.4.2 Sample Space

Based on the scaled fitness value a probability is calculated for each individual. This is multiplied by the size of the population n to get expected number of times an individual should be selected (e_i) as parent:

$$e_i = (f_i' / \sum_{i=1}^{n} f_i') \times n \tag{4.6}$$

A *sample space* is defined based on e_i values. It consists of an array of records with two fields - a member identification number field and a probability field. For example if $e_j = 2.6$, then individual j will receive three slots $(j, 1.0)$, $(j, 1.0)$, and $(j, 0.6)$ in the sample space (Figure 4.2). Note that the first field in a slot is the individual's identification and the second field is the probability with which this slot should be accepted.

Assume that there are total of m slots in the sample space. To select a parent a random number is generated between 1 and m and the individual corresponding to that slot is selected as parent with the probability of that slot. This process is repeated until a parent is selected. Since, the fitter individual will get more slots in the sample space, they have high chance of being selected. The diversity in the population is maintained because the selection is random over the sample space. The following types of choice functions were tested and the last one was found most effective:

1. Random selection of parent without any regard to their fitness values.

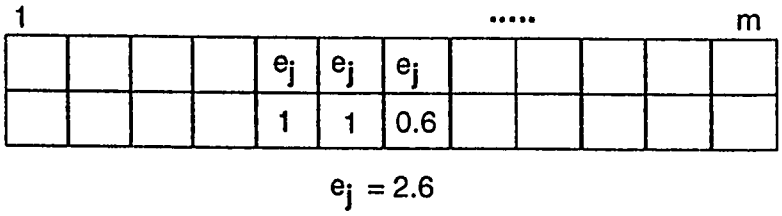


Figure 4.2: Sample space.

2. Random selection of parent corresponding to a certain slot in the sample space without any regard to the probability of that slot.

3. Random selection of parent corresponding to a certain slot in the sample space with regard to the probability of that slot.

4.5 Crossover

The nodes in CDFG have precedence constraints that should not be violated when the crossover operator is applied. In [WGH90] a simple two point crossover followed by a modified ASAP scheduling was proposed. This technique can produce schedules which are longer than the specified control step limit and is thus believed to take longer to find good schedules. Note that scheduling is to be performed each time the crossover is applied. We opted to have a crossover that will always give valid schedule rather than a crossover where scheduling has to be done separately. Given the coding as described in a previous section, it is a difficult proposition that is to be resolved. If we fix the order of nodes in the chromosome a simple one or two point crossover will result in an invalid offspring chromosome. The following schemes are developed to generate valid offspring.

4.5.1 Alternating Crossover

The term alternating crossover as used here means that given the same order of genes in both parents, we take genes from the two parents in the alternating sequence such that whenever there is a violation of precedence constraint we take the gene from the other parent but maintain the alternating sequence. If we traverse the graph in the depth-first (DF) order we get a specified order of nodes in the graph. Think of a chromosome where genes represent the nodes in this order. Consider a crossover operator in which we take alternating genes from two parents such that whenever there is a precedence constraint violation we take the gene from the other parent but maintain the alternating sequence. This will result in an invalid offspring chromosome.

The above example of an invalid crossover operator indicates that order of genes is important. Thus this became the main theme in the search of a valid crossover operator for the given chromosome. It is found that if we put the genes in the reverse DF order such that successors are always on the left hand side of their predecessors, we can use the alternating crossover to generate valid offspring. It works because whenever we take a node that is to be scheduled all of its successors are already scheduled and thus we can check for any violations.

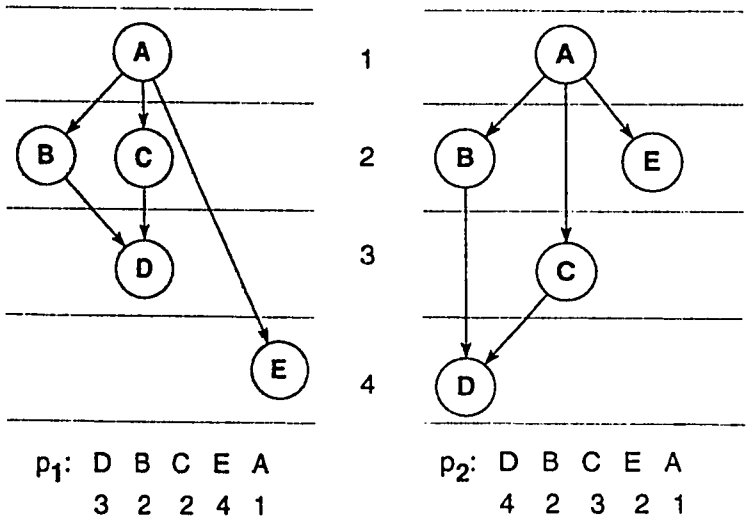
A working example of the alternating crossover is shown in Figure 4.3. Figure 4.3(a) shows the two selected parents (p_1 and p_2) for crossover and the Figure 4.3(b)

shows the resulting offspring (*os*) with genes labeled with the parent tag from which it is taken. It can be seen that there are no scheduling violations in this example. An example which results in such a violation is shown in Figure 4.4. Figure 4.4(a) shows the two parents. As indicated in Figure 4.4(b), during crossover we take alternating genes from each parent. At one point we can not take gene from parent 1 so this gene is taken from parent 2 but the alternating sequence is maintained and the next gene is also taken from parent 2.

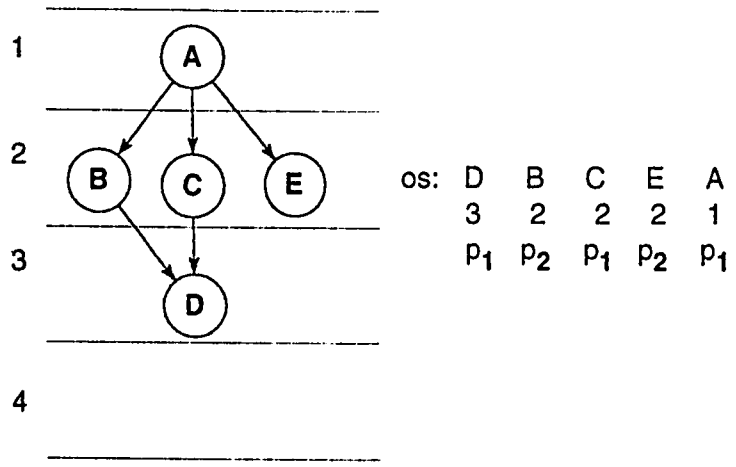
4.5.2 Order Crossover

It is found that alternating crossover is not able to adopt good structures from the parents. The main reason for this is that it works bottom up and things become constrained for upper operations. Thus the chances of mixing the genes becomes less. For this reason we started looking for a better crossover operator. Let us remove the restriction on the order of the genes in the chromosome. A simple *order crossover* works as follows: A cross point is randomly generated and genes on left side of one parent are copied to offspring in those positions. The other parent is scanned from left to right and leftover genes are stored in the remaining positions of the offspring in that order (Figure 4.5). This ensures that no genes are duplicated or missed.

Using this simple order crossover will of course give invalid schedules. The technique we adopted to avoid invalid schedules is as follows. The cross point is randomly

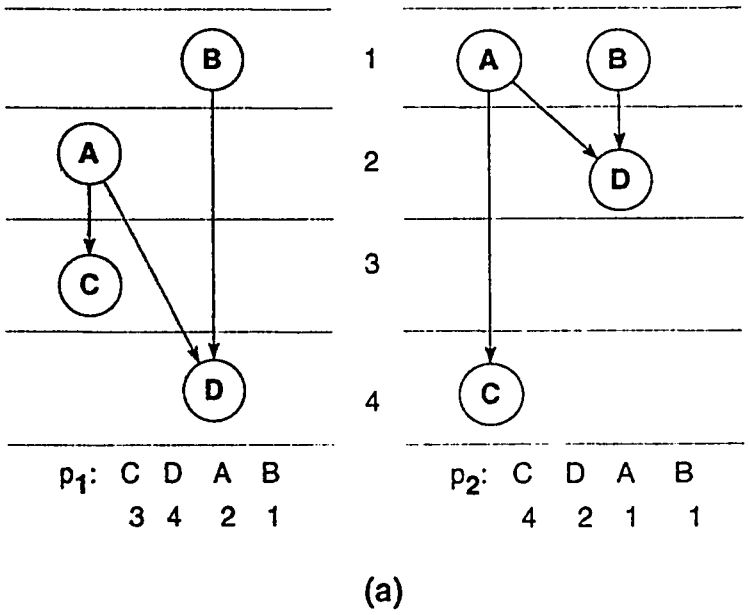


(a)



(b)

Figure 4.3: Alternating crossover example with no scheduling violations: (a) Parents; (b) Offspring.



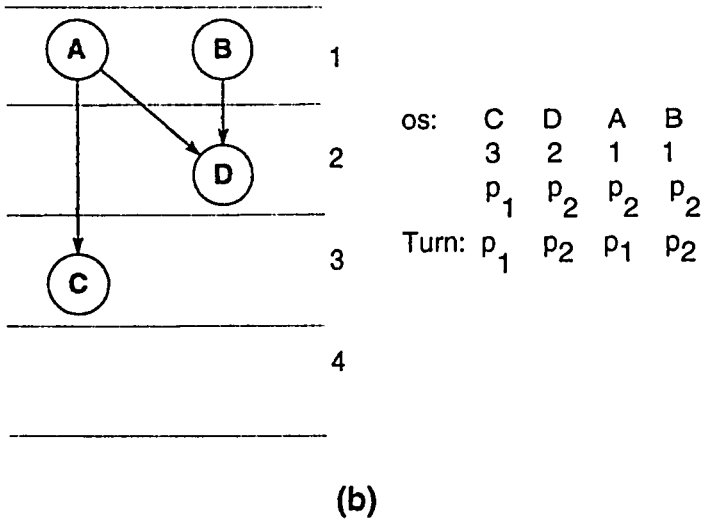


Figure 4.4: Alternating crossover example with scheduling violations: (a) Parents; (b) Offspring.

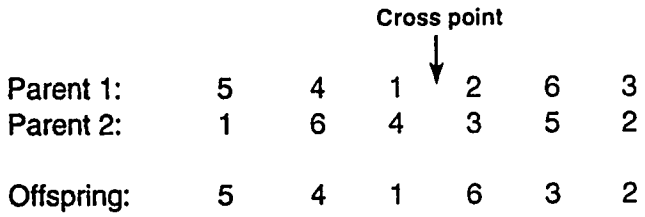


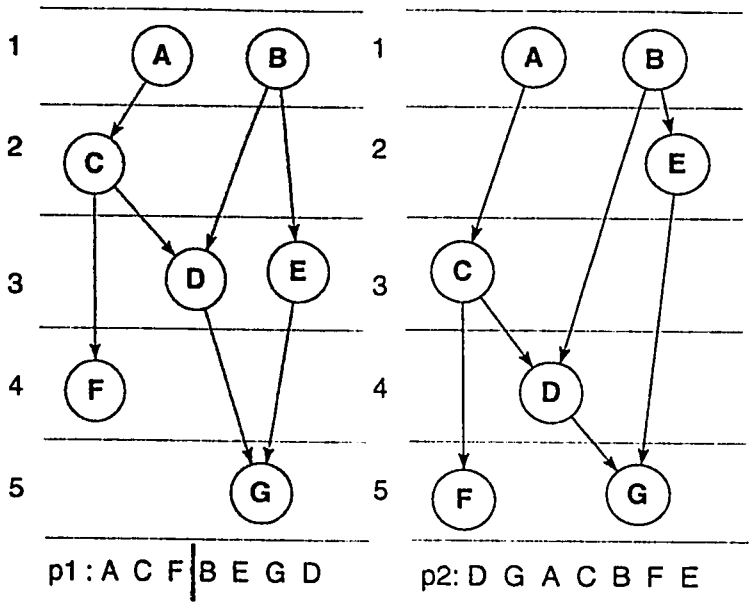
Figure 4.5: Simple order crossover.

generated and left genes of one parent are copied to the offspring. This determines the schedule for some operations. Given schedule for some operations in CDFG, the ASAP schedule for the remaining operations can be determined. Those genes from the other parent which do not violate the precedence constraints are copied to the offspring and those which do violate are taken from the first parent. The ASAP values are used to check any violations. An example of this is shown in Figure 4.6. The cross point is between the third and fourth gene of parent p_1. The left three genes (A, C, F) are copied from parent p_1 to the offspring and the ASAP schedule for the remaining genes as induced by genes (A, C, F) is determined. Since none of the remaining genes from the other parent violate the precedence constraints they are copie d without any trouble. This crossover is able to group together good structures in an offspring which is passed from generation to generation.

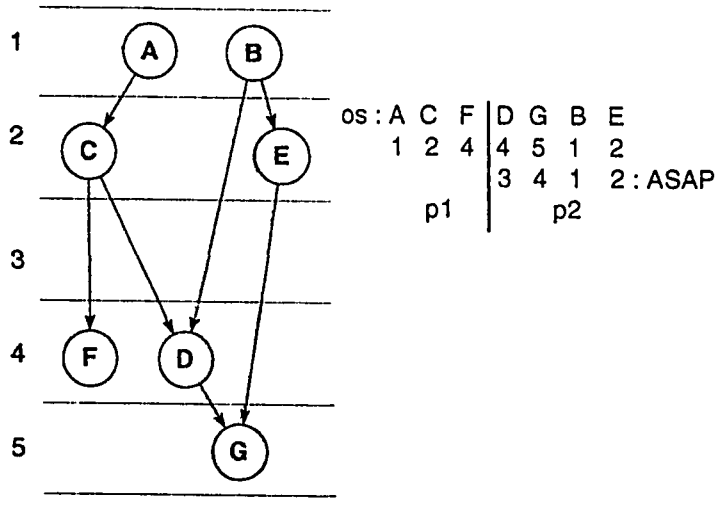
4.5.3 Functional Unit Violation

Functional unit and functional unit input assignments are also taken from the same parent. One can easily notice that sometimes this will result in concurrent assignment of the same functional unit to two or more operations in the same control step. One way to resolve this situation is to include a violation term in the cost function of Equation 4.1. Thus the cost function will then become

$$C = W_{cs} \times N_{cs} + W_{reg} \times N_{reg} + W_{bus} \times N_{bus} + W_{fu} \times N_{fu} + W_{ic} \times N_{ic} + W_{viol} \times N_{viol} \quad (4.7)$$



(a)



(b)

Figure 4.6: Example of order crossover tailored for scheduling: (a) Parents; (b) Offspring.

where,

W_{viol} weight assigned to each functional unit violation

N_{viol} number of violations.

The weight assigned to each violation should be high enough so that there are no violations in the final result and low enough so that the individual is not completely neglected. It is easy to see that one violation of a functional unit means that we may or may not need an extra functional unit. Thus the weight assigned to a violation is approximately the same as the weight assigned to a functional unit.

The other way around is to reassign the functional units for violating operations only. The advantage that one can think of for the first scheme is that one would expect the functional unit assignment to improve genetically. But if there are too many violations then it will undermine any genetic improvement. This is indeed the case as found by experiments. In such circumstances the second scheme looks practical. It is important to note that reassignment of functional units is like mutation of the functional unit assignment part of the chromosome with a somewhat high probability.

4.5.4 Normalization of Functional Unit Assignment

Besides functional unit violation there is one more problem that is to be handled when we apply the crossover operator. Suppose that the maximum number of

functional units in one or both parents are 3. The crossover can produce an offspring that uses only two functional units (Figure 4.7). Since we inherit the assignments from the parents unless there is a violation, it may happen that some operations are assigned to functional unit 3 whereas the maximum number of functional units used in offspring are only 2. This means that one or both of the functional units are free in the control steps corresponding to those operations. Thus the functional unit assignments for these operations are performed again within the maximum range.

4.6 Mutation

Mutation operator occasionally introduces beneficial material into some members of the population. It is applied on offsprings (after the crossover operation) with a very low probability. It may also destroy good properties of the offsprings, but since the fitness value reflects how good that member is, it is hoped that it will not be passed to the next generation if the effect of mutation is bad. Three types of mutation operators are used in the present implementation.

4.6.1 Control Step Mutation

This is the most important type of mutation. An operation is selected randomly. An attempt is made to either move it up or down. The direction is generated randomly. If it does not result in any violation, its control step value is changed. Otherwise,

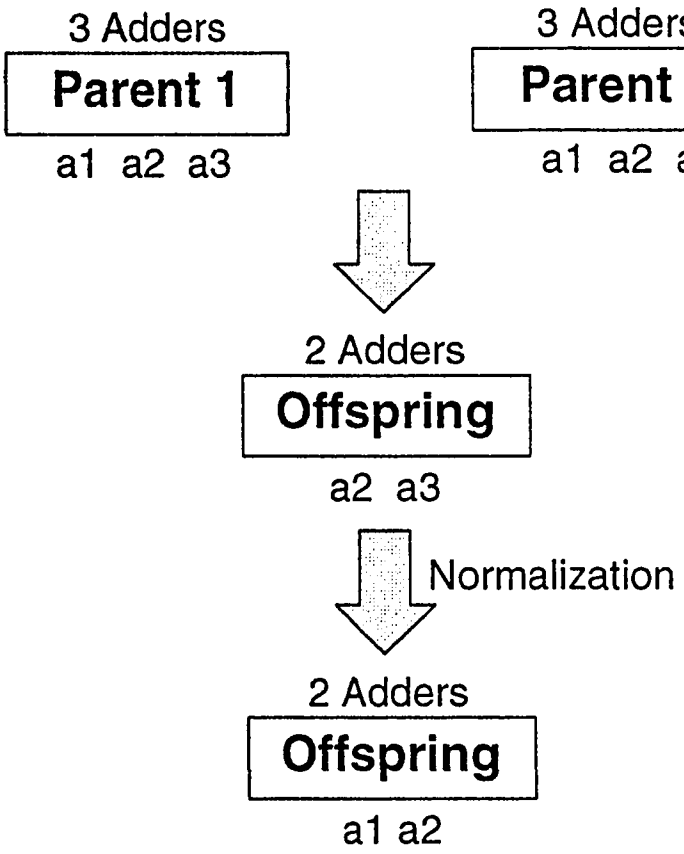


Figure 4.7: Functional unit normalization.

the mutation attempt is repeated on other operations a limited number of times. If no valid control step mutation is found the mutation is abandoned. Control step mutation has very far effects. It can produce better schedule and reduce the number of functional units, buses and registers.

4.6.2 Functional Unit Assignment Mutation

An operation is selected randomly and a new functional unit number is generated. If this one is not used by any other operation in that control step then the functional unit assignment of the operation is changed to this one. If it is used by some other operation another mutation attempt is made. This is repeated a limited number of times after which the mutation is abandoned. This type of mutation helps in reducing the number of interconnections.

4.6.3 Functional Unit Input Mutation

An operation is selected randomly and if it is a commutative operation then the assignment of its left variable to the functional unit input is changed. This type of mutation also helps in reducing the number of interconnections.

4.7 Selection

Crossover is applied on the population with a specified rate. After all the crossovers are complete we get an increased population consisting of parents and offsprings. We opted to have a fixed population size. Thus the next step is to transfer some of the individuals among parents and offsprings to the next generation. This is done by a selection function based on fitness value. We create another sample space in the same manner as discussed in Section 4.4 for the increased population. Thus the selection function is the same as the choice function. This is applied as many times as population size to get the new population. It is found that good results can be obtained if this scheme is combined with one or more of the following schemes:

1. Always selecting the best individual in the population.

2. Selecting a specified quantity of the best individuals.

3. Selecting some specified quantity randomly.

These schemes help in improving the search and maintaining the diversity in the population, which is necessary for search space exploration, and avoids premature convergence to the local optimum.

Replacement technique is also tested. Replacement means that some fraction of the population is replaced by the offsprings. The term *generation gap* is used in this context. If the generation gap is 1.0 then the whole population is replaced. When

replacement is used the crossover rate should always be greater than or equal to the generation gap. Thus if generation gap and crossover rate is 0.25 then 25% of the population is replaced. Replacement technique did not produce good results.

Chapter 5

Scheduling and Allocation using Tabu Search

This chapter will discuss the tabu search (TS) implementation for scheduling and allocation. An introduction to TS is given in Section 3.2. The main tasks in order to formulate scheduling and allocation for TS are as follows:

- Starting with a proper initial solution.
- Defining a neighborhood for a given solution.
- Generation of moves.
- Formulation and maintenance of tabu list.
- Defining a proper aspiration level criterion.

- Finding a good tabu list size.
- An efficient way to accept moves.

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

5.1 Initial, Current and Best Solution

Although in theory the initial solution can be any feasible solution, it is found that TS may take longer if given a poor initial feasible solution. We may start with ASAP or ALAP schedule with a specified limit on the number of control steps. In both these schedules only a few operations can be disturbed or rescheduled in the beginning. Thus, it is found better to use either mobility-up or mobility-down scheduling (see Section 4.3). In the final implementation mobility-up scheduling is used for the initial solution. As TS proceeds we keep two solutions - one is the current solution and the other is the best solution found so far. The best solution found in n iterations is the output of TS, where n is specified by the user.

5.2 Generation of Moves

Given a solution, the generation of moves in the neighborhood of this solution is an important step in TS. The following three kinds of moves are defined for this purpose:

1. Moves based on changing the control step of an operation.

2. Moves based on changing the functional unit assignment.

3. Moves based on changing the functional unit input assignment of variables.

The first move is intended toward optimizing the number of control steps, functional units, registers and buses, whereas the last two moves are intended for the optimization of interconnections. Probabilities are assigned for each of these moves in accordance with the importance of each move toward optimizing the cost. Thus the move type is selected probabilistically and N_{cm} moves of that type are generated, where N_{cm} is the number of candidate moves. The solutions obtained by each move is evaluated using Equation 4.1. The moves are generated such that the new solutions are always feasible, but some or all of the moves may be tabu or may not pass the aspiration criterion. In terms of first type of move, a feasible solution means that the precedence constraints are never violated. An operation is moved up or down where the direction is generated randomly. Functional unit changes are only performed if there are free functional units for the control step in which that operation is scheduled. Functional unit input changes are performed only for commutative operations.

Another way of generating neighborhood solutions is by making more than one of the moves. This approach has less chances of finding the global optimum as the solution may be disturbed too much and, in fact, it might not be in the neighborhood

of the present solution. Thus this approach is not used.

5.3 Tabu Lists

Formulation of the tabu list is one of the main steps in mapping a problem for TS. Since we have three types of moves, decision need to be made whether to use one tabu list or three tabu lists. It has been suggested by Glover [Glo90a] that when the solution depends on multiple parameters it is appropriate to use more than one tabu list. Maintaining multiple tabu lists helps in generating search paths with different characteristics.

In the present implementation we used three tabu lists - one for each type of move. TS continues for the maximum number of specified iterations, n. Since separate tabu lists are maintained, we need to count how many times the particular type of move is performed. This number corresponds to the iteration number for that type of move. When the sum of iterations for three types of moves becomes n, TS stops. Attributes selected for the control step move are discussed next. A two-dimensional array *csTabuList* is maintained for the tabu list. The first dimension corresponds to the total number of operations and the second dimension corresponds to the possible control steps to which an operation can be assigned. If an operation op_i is moved from (say) control step cs_i to a new control step cs_j, we store the current iteration number corresponding to the control step moves in $csTabuList[op_i][cs_i]$.

Note that the reverse move is stored as this makes it easier to check the tabu status of future moves.

Similarly two one-dimensional arrays $fuTabuList$ and $fuInpTabuList$ are maintained for other types of moves. The dimension corresponds to the number of operations. If the effected operation in such moves is (say) op_j then the iteration number for that type of move is stored in $fuTabuList[op_j]$ or $fuInpTabuList[op_j]$. All these recordings of tabu status are found effective and good results are obtained.

5.4 Aspiration Level Criteria

After all the candidate moves of a particular type are generated for iteration itr, the best of these is selected, which may not be better than the current solution. If it is not tabu, it is accepted. Accepted move becomes the current solution for the next iteration. The tabu list size (T_{size}) is an important parameter in TS. In the present implementation *magic number* 7 is used for T_{size} and is same for all three lists. Since we store the iteration number in order to check the tabu status, a move is tabu if the difference of itr and stored iteration number is less than or equal to T_{size}. If it is tabu, its aspiration level is checked as described below.

A common aspiration level (AL) criterion is used for all the three moves. Aspiration levels are associated with each of the operations and are initialized to infinity. If a move m_i affects an operation op_i and m_i is tabu (forbidden), AL(op_i)

value is checked against $c(m_i)$, the cost of the solution achieved by move m_i. If $c(m_i) < AL(op_i)$, the move is accepted and $AL(op_i)$ is set to $c(m_i) - 1$. Otherwise another set of same type of candidate moves are generated. A maximum limit on regenerations are specified after which a new type is selected for candidate moves. Note that aspiration level of an operation is updated only when it overrides the tabu status of that operation. Thus aspiration level is not the best historical value. It may allow to go to a previous solution reached by a tabu move, but this can happen only once. Thus cycling is avoided. This aspiration level criterion gives more freedom to explore the search space and is found effective for scheduling and allocation.

5.5 Alternate Implementation

In an alternate implementation we tested with separate tabu list sizes for each type of move. Two aspiration level criteria are used - one for the control step moves and the second for other types of moves. For control step moves there is a separate aspiration level for each operation for each of its possible control steps. Aspiration level corresponding to an operation for a particular control step is one less than the cost of the solution obtained when that operation was last assigned to that control step. It is updated each time a move is accepted and is thus not a historical value. It serves to override tabu status to explore new search paths.

Aspiration level for an operation corresponding to other two moves is one less than the interconnection cost obtained when one of these moves was last applied to that operation. Interconnection cost is used because these two moves are intended toward optimizing interconnection cost. In this implementation a candidate list is prepared and consists of solutions reached through non-tabu moves or, if tabu, then they passed the aspiration criterion. The best among these is selected. The candidate list size is kept between 5 and 10.

Although both implementations were able to find good solutions, it should be noted that aspiration criteria are more strict in the second implementation than the first one. The first implementation is not strict in selecting moves and aspiration criterion is easy to pass. Because of the use of candidate list strategy the second implementation is able to find good solutions quicker than the first one. An instance where implementation 2 has achieved better result than implementation 1 is shown in Figure 5.1. The graph shows the plot of move costs for both implementations for the case of 7 control step limit on discrete cosine transform CDFG with pipelined multipliers option. Note that although implementation 1 achieved its lowest cost solution earlier than implementation 2 it was not able to reach the lowest cost solution of implementation 2 for the same number of iterations.

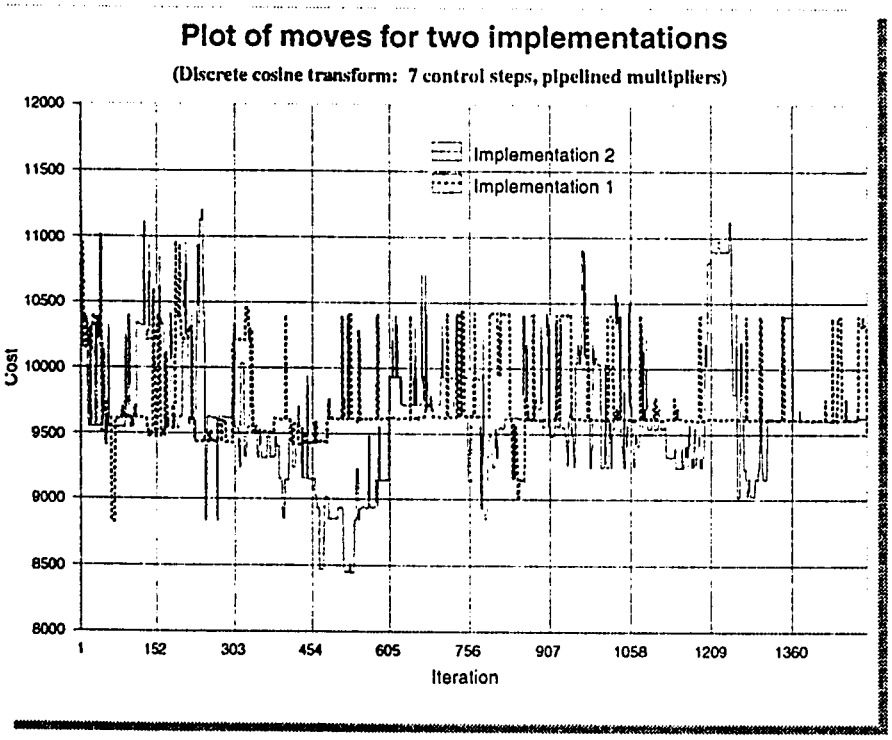


Figure 5.1: Comparison of two implementations.

Chapter 6

Data Path Synthesis Using Genetic Algorithm

Interconnection of registers, buses, multiplexers and ALUs is called *data path* and the process of forming such an interconnection is called *data path synthesis* (DPS). Allocation using genetic algorithm as described in the previous chapters has only done some part of actual allocation. Functional units are allocated to operations and variables involved in the operation are assigned to the functional unit inputs. Number of registers, buses, and interconnections are optimized in an attempt to solve scheduling and allocation as a combined problem. The minimum number of registers and buses for the given schedule is known. Interconnection cost is only optimized. We still do not know the exact data path. Mappings of variable to register and data transfer to bus still need to be done. Both these mappings have

a profound effect on the interconnection cost. Different mappings will give different interconnection costs and this can also be formulated as an optimization problem. This is the subject of this chapter.

After we map the variables to registers and data transfers to buses, we have the following information at our disposal about the high-level description from which we started.

- An operations is scheduled during which control step.
- An operation is performed on which functional unit.
- A variable goes to which input of the functional unit.
- A variable is stored in which register during any specific control step.
- A data transfer is performed on which bus.

Once we know all this information one can easily generate the data path for the given high-level description.

6.1 Architecture

The architecture used for optimizing the number of interconnections is shown in Figure 6.1. Outputs of functional units and registers are connected to buses. Multiplexers are provided at the inputs of functional units and registers if the input

comes from more than one bus. A direct interconnection is provided in case the input comes from only one bus. The interconnection cost is estimated by number of multiplexer inputs.

6.2 Genetic Algorithm for DPS: A brief overview

The main task here is how to formulate this problem for the genetic algorithm. As described earlier, two mappings are to be performed. A chromosome need to be devised that can incorporate both these mappings. The fitness of each individual is based on finding the exact number of interconnections which involves calculating the total number of multiplexer inputs. Suitable initial population has to be created. Choice and selection functions are somewhat similar as discussed in Chapter 4. A crossover operator which produces valid mappings is to be found. Suitable mutation function is required. We will address all these problems in the remaining sections of this chapter.

6.3 Initial Population

Since we have to perform two types of mappings, the chromosome has to incorporate both these mappings. Thus the chromosome has two parts - one for the variable to register mapping and the other for data transfer to bus mapping.

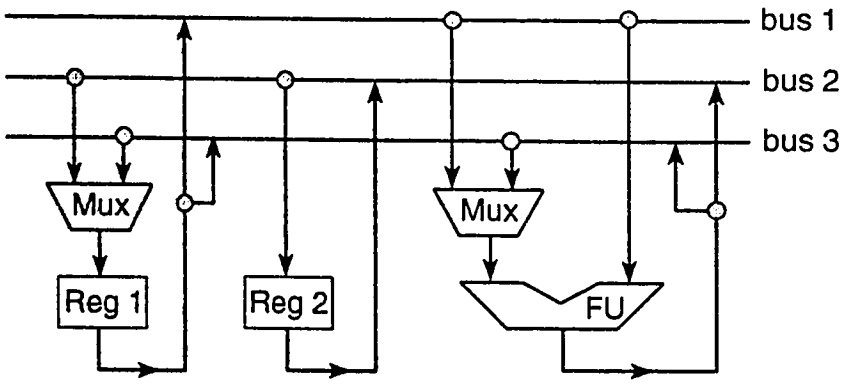


Figure 6.1: Architecture on which data path is mapped.

6.3.1 Chromosome Part for Variable to Register Mapping

Given the life time analysis of the variables, the left-edge algorithm [HS71] can be used to map all variables to registers optimally for the given schedule. (It should be mentioned here that if a variable is regenerated then multiple life times are kept for each regeneration as it may help in optimizing the number of interconnections). One problem with the left-edge algorithm is that it does not take the interconnection cost into consideration while grouping variables. This is because the left-edge algorithm considers left edges in the sorted order. This problem is illustrated in Figure 6.2. Lifetimes are shown in Figure 6.2(a). Assume that sorted order is (v_1, v_2, v_3, v_4). Left-edge algorithm will give the grouping of Figure 6.2(b). Another grouping is given in Figure 6.2(c). It may happen that grouping of Figure 6.2(c) may result in less interconnections than the grouping of Figure 6.2(b). Thus there is a possibility of interconnection cost optimization by considering various optimal (in terms of number of registers) groupings.

We can utilize this fact to create initial population for this part of the chromosome. Given the lifetime analysis we can perturb the order of segments that start from the same control step. This will not affect the sorted order of the left edges. Applying left-edge algorithm on this configuration one can get a different grouping. In this way, the initial population for this part of chromosome is created.

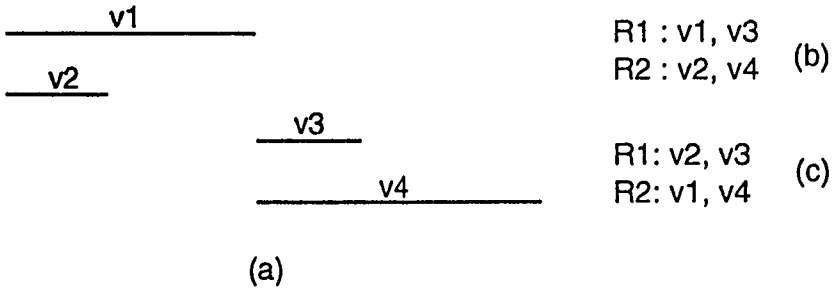


Figure 6.2: Grouping variables into registers.

6.3.2 Chromosome Part for Data Transfer to Bus Mapping

A list of data transfers for the scheduled CDFG can be made using depth first search. All data transfers during control step (say) i can take place on any available bus, but only one data transfer can take place on any bus at one time. As mentioned earlier different mappings will give different number of interconnections. An easy way to make a chromosome out of this situation is to think of any bus as consisting of segments for each of the control steps as shown in Figure 6.3. Data transfers can be assigned to segments or slots. A list of data transfers is made and bus chromosome is filled with the particular index to the data transfer list. Some slots will remain empty meaning that there is no data transfer on that bus during that particular control step.

To create the initial population a sample chromosome is prepared. This can be done by noticing that data transfers can be assigned to buses by using left-edge algorithm. Data transfers in each column of sample chromosome (Figure 6.4) can be interchanged randomly to generate initial population.

6.3.3 Complete Chromosome

Complete chromosome is shown in Figure 6.5. The left part is the bus chromosome and the right part is the register chromosome. One can think of register chromosome as hooked to the bus chromosome. Crossover is only applied to the bus chromosome

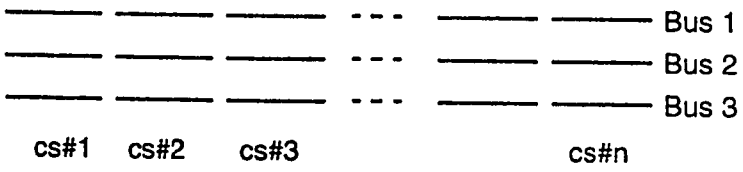


Figure 6.3: Structure of the bus chromosome.

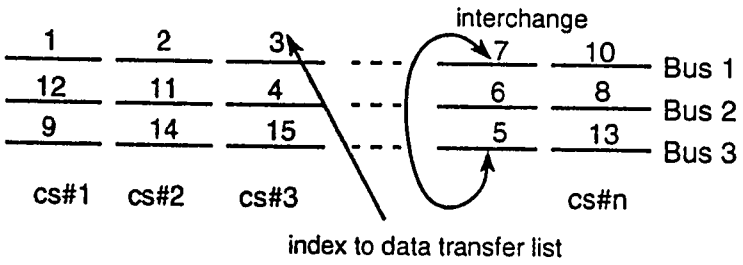


Figure 6.4: Sample bus chromosome.

whereas mutation is applied on both bus and register chromosomes. One should think of bus chromosome as if all buses are put next to each other one after another.

6.4 Fitness Calculation

The first step in calculating the fitness value for each individual is to calculate the number of multiplexer/bus inputs. We keep the following information for each data transfer:

1. Functional unit involved.
2. The functional unit input to which it goes.
3. The type of transfer (input or output).
4. The control step in which it takes place.
5. Register in which the variables involved are stored or to be stored.
6. Bus on which it will take place.

Given this information one can calculate the number of multiplexer/bus inputs required. Since this is also a minimization problem, the number of multiplexer inputs is subtracted by a specified maximum number of interconnections. The resulting number for different individuals will not be much apart, so it is multiplied by a large

Figure 6.5: Complete chromosome.

number to create some difference between them. Following this, sigma truncation and linear scaling is applied as usual to get the final fitness value. The sample space is created and choice and selection functions take the same form as discussed in chapter 4.

6.5 Crossover

As mentioned earlier crossover is only applied to the bus part of the chromosome. A necessary property for the crossover operation is that the data transfers should not change their control step during crossover operation. With reference to Figure 6.4, it means that they should remain in the same column. Another required property for the crossover is that the data transfers should not be duplicated or missed.

Different crossovers were considered that failed to satisfy one or both of these properties. Simple one or two point crossover will change the data transfer control steps as well as duplicate them. Order crossover or PMX will not duplicate data transfers but they will change the control steps.

It is noticed that cycle crossover has an interesting property that can be utilized here. In cycle crossover offspring inherits genes from one parent or the other in the same position as the corresponding parent. An example of a cycle crossover is illustrated in Figure 6.6. There are two cycles: 3-1-6-3 in parent 1 and 4-2-5-4 in parent 2. We randomly start with parent 1. During first cycle, offspring 1 get genes

3, 1, and 6, and offspring 2 gets genes 1, 6, and 3. For second cycle we start with parent 2. During this cycle, offspring 1 gets genes 4, 2, and 5, and offspring 2 gets genes 2, 5, and 4. Note that the net effect of the second cycle is to swap the genes in the two parents as they are passed to offsprings.

Now, consider the two parent bus chromosomes shown in Figure 6.7. Assume that there are four control steps and thus there are three buses. The non-negative numbers are indices to the data transfer list and -1s indicate that there are no data transfers in those control steps. One would notice that we can not apply cycle crossover directly on this bus chromosome. The reason is that the gene's value (*alleles*) are not distinct. In order to have a bus chromosome on which cycle crossover can be applied, we fill the segments having no data transfer (in the sample chromosome) with the numbers greater than the total number of data transfers. In Figure 6.7, there are eight data transfers. Thus -1s positions are filled with numbers 9, 10, 11, and 12 as shown in Figure 6.8(a). After cycle crossover is applied the resulting offsprings are shown in Figure 6.8(b).

6.6 Final Data Path

As mentioned in section 6.4, we store enough information for each data transfer that can be used to find the number of multiplexer inputs. Using the same information one can easily generate the final data path for the high-level description. This

Parent 1:	3	2	1	5	4	6
Parent 2:	1	4	6	2	5	3
Offspring 1:	3	4	1	2	5	6
Offspring 2:	1	2	6	5	4	3

Cycle 1: 3 - 1 - 6 - 3 (in parent 1)
Cycle 2: 4 - 2 - 5 - 4 (in parent 2)

Figure 6.6: Cycle crossover example.

3	-1	8	6	-1	4	1	5	7	-1	2	-1
6	7	-1	1	8	-1	2	-1	5	3	-1	4

Figure 6.7: An example of a bus chromosome.

information is as follows:

- Functional unit number for each operation.
- Functional unit input number for each input variable.
- Variable to register mapping for each control step.
- Data transfer to bus mapping for each data transfer.

Multiplexers are provided only if there are multiple inputs coming to the two inputs of a functional unit or the input of a register. A direct interconnection is implied if there is only one input to a bus.

Parents

3	9	8	6	12	4	1	5	7	11	2	10
6	7	11	1	8	9	2	10	5	3	12	4

(a)

Offsprings

3	7	8	6	12	9	1	10	5	11	2	4
6	9	11	1	8	4	2	5	7	3	12	10

(b)

Figure 6.8: (a) Bus chromosome suitable for cycle crossover, (b) Offsprings resulting from cycle crossover.

Chapter 7

Experimental Results

Genetic scheduling and allocation (GSA) and tabu scheduling and allocation (TSA) are tested on various benchmarks. Table 7.1 shows the results for differential equation benchmark. Table 7.2 shows the results for more complicated fifth order elliptic wave filter (EWF) benchmark. STAR system uses parallel data transfers, so that the bus comparison with this system is of little significance. The results shown for 17_{LU} control steps are for *L*oop *U*nfolding. Table 7.3 shows the results obtained for discrete cosine transform (DCT) benchmark. The results are compared with simulated evolution (SE) [LM93], HAL system [PK89], SALSA II [RN93], STAR system [TH92], EMUCS system [HT83], SAW [TLW+90] and CATREE system [GE87]. Comparisons are given for number of control steps (CS), adders (+), multipliers (*), registers (Reg), and multiplexers or buses (Mx). The cost column indicates the cost achieved by the respective technique. The costs of control steps, adders,

multipliers, registers, multiplexers, and interconnection are derived from [DN89]. The p in (*) column stands for pipelined multiplier. The (+) column in DCT table actually corresponds to number of functional units capable of performing addition and subtraction. It is assumed that addition takes one whereas multiplication takes two control steps.

The performance of GSA for differential equation benchmark is shown in Figure 7.1. The graph shows the average cost versus generations. The moves taken by TSA for differential equation benchmark is shown in Figure 7.2. The graph shows the move costs for each iteration. The convergence trend is also shown.

Data path synthesis result using genetic algorithm for differential equation benchmark is shown in table 7.4. The result is comparable with the best known systems. The performance of data path synthesis using genetic algorithm for differential equation benchmark is shown in Figure 7.3. The graph shows the interconnection cost versus generations.

System	CS	ALU	*	Reg	Mx	Cost
GSA	8	1	1p	5	4	1710
TSA	8	1	1p	5	4	1710
HAL	8	1	1p	5	4	-
SE	8	1	1p	5	5	-

Table 7.1: Differential Equation Results.

System	CS	+	*	Reg	Mx	Cost
GSA	17	3	3	11	10	4786
TSA	17	3	3	11	10	4804
SE	17	3	3	11	11	-
HAL	17	3	3	-	-	-
SALSA II	17	3	3	-	-	-
GSA	17	3	2p	11	10	3986
TSA	17	3	2p	11	10	4006
SE	17	3	2p	11	12	-
HAL	17	3	2p	-	-	-
CATREE	17	3	2p	12	-	-
SALSA II	17	3	2p	-	-	-
GSA	17_{LU}	2	1p	10	8	2638
TSA	17_{LU}	2	1p	10	8	2638
STAR	17_{LU}	2	1p	11	5*	-
GSA	18	2	2	11	8	3484
TSA	18	2	2	10	8	3424
SE	18	2	2	10	9	-
SALSA II	18	3	2	-	-	-
GSA	19	2	2	10	7	3370
TSA	19	2	2	10	7	3340
SE	19	2	2	10	11	-
HAL	19	2	2	12	-	-
EMUCS	19	2	2	12	12	-
SAW	19	2	2	12	-	-
GSA	19	2	1p	11	8	2644
TSA	19	2	1p	10	7	2558
SE	19	2	1p	11	9	-
HAL	19	2	1p	12	6	-
STAR	19	2	1p	11	4*	-
SALSA II	19	2	1p	-	-	-

Table 7.2: EWF Results.

System	CS	+	*	Reg	Mx	Cost
GSA	7	6	8	19	18	10894
TSA	7	6	8	19	18	10904
SALSA II	7	6	8	-	-	-
GSA	7	8	4p	21	21	8788
TSA	7	6	5p	19	17	8442
SALSA II	7	8	4p	-	-	-
GSA	8	5	6	15	17	8712
TSA	8	5	6	15	16	8660
SALSA II	8	5	6	-	-	-
GSA	8	5	4p	17	15	7030
TSA	8	5	4p	16	16	7110
SALSA II	8	5	4p	-	-	-
GSA	9	4	6	15	14	8076
TSA	9	4	6	14	15	8158
SALSA II	9	4	6	-	-	-
GSA	9	4	3p	13	14	5626
TSA	9	4	3p	13	14	5660
SALSA II	9	4	3p	-	-	-

Table 7.3: DCT Results.

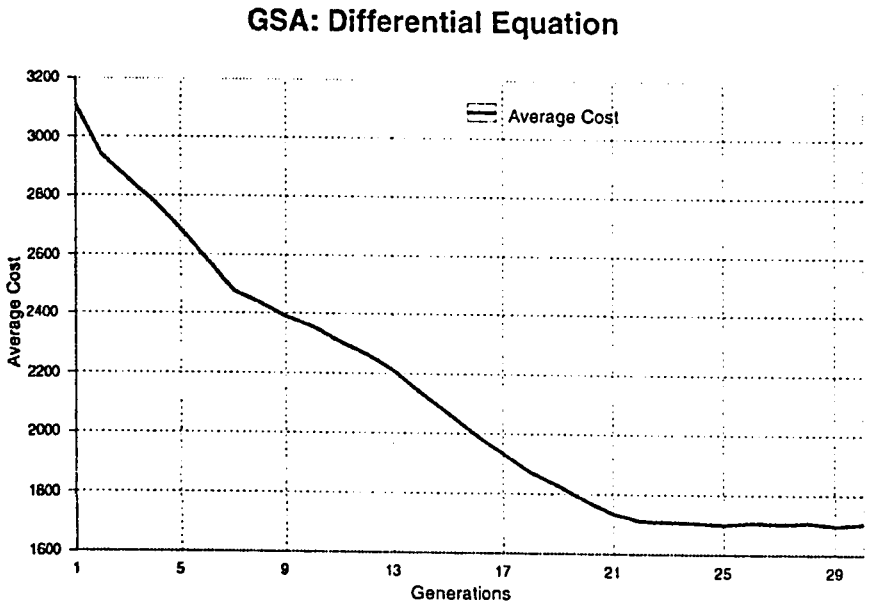


Figure 7.1: GSA: Graph of average cost versus generations for differential equation.

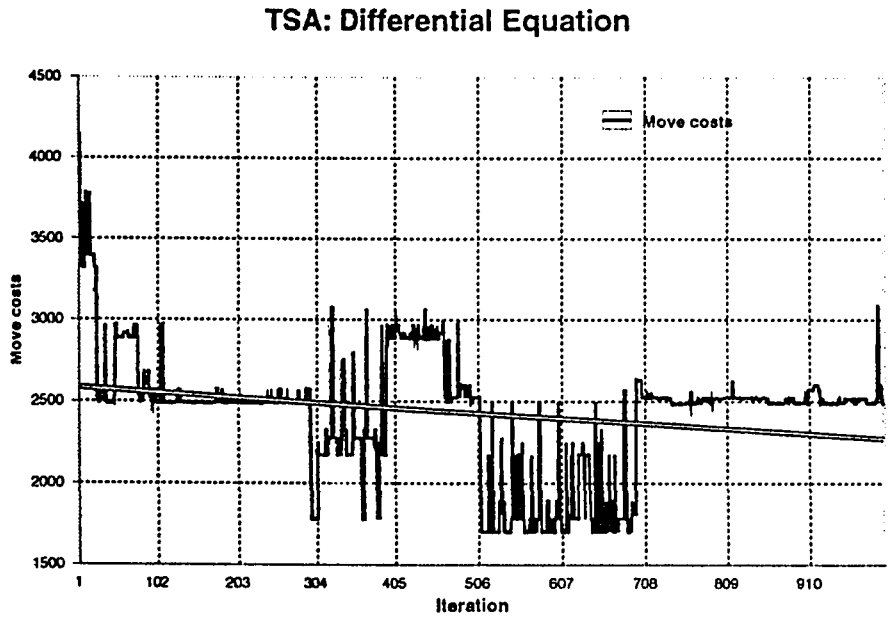


Figure 7.2: TSA: Graph of move costs for each iteration for differential equation.

System	Mux Inputs
DPS	14
Splicer	16
HAL	13
SE	12

Table 7.4: Data path synthesis result.

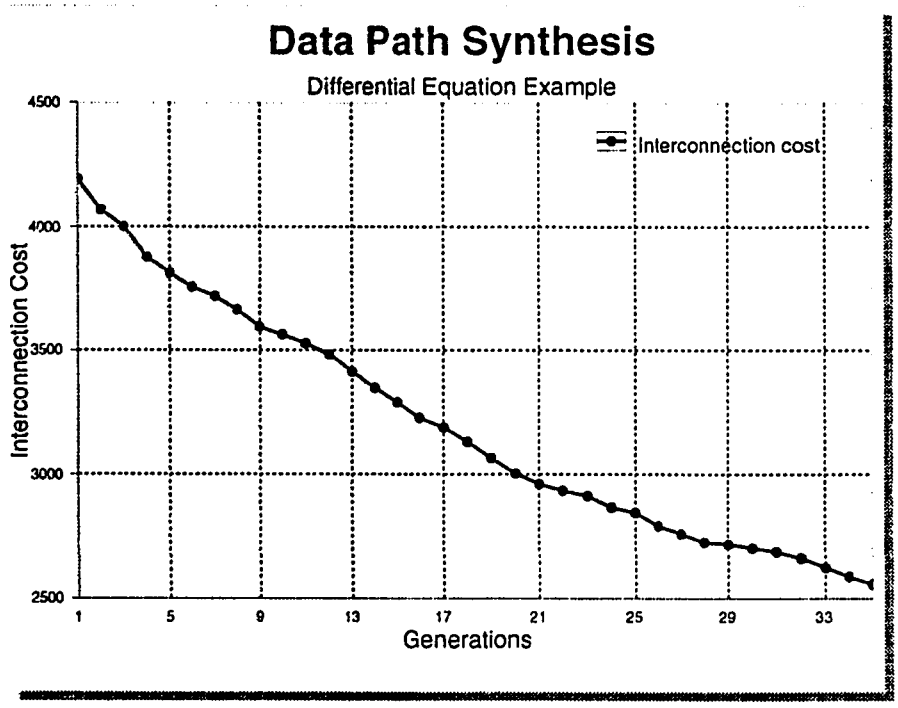


Figure 7.3: Data path synthesis using GA: Interconnection cost versus generations.

Chapter 8

Conclusions and Future Work

This short chapter concludes this thesis and highlights some future work.

8.1 Conclusions

Genetic algorithm is a promising optimization technique. This thesis has presented its application to scheduling and allocation in high-level synthesis. The work involves finding an appropriate string encoding or chromosomal representation. The initial population of solutions is constructed to get better results. Two scheduling techniques, mobility-up and mobility-down, are used for this purpose. Two new crossover operators (alternate crossover and order crossover for scheduling) are presented which can find application in many other areas. Genetic scheduling and allocation (GSA) is tested on three benchmark circuits namely differential equation,

fifth-order elliptic wave filter, and discrete cosine transform. Results obtained are comparable with those obtained by other systems. GSA approach is different from a previous attempt using GA [WGH90] in many respects. The contributions include: a new chromosomal representation for scheduling and two subproblems of allocation; and two novel crossover operators to generate legal schedules. In [WGH90] a simple two point crossover followed by a modified ASAP scheduling was proposed. This technique can produce schedules which are longer than the specified control step limit and is thus believed to take longer to find good schedules. In their technique scheduling has to be performed each time the crossover is applied. GSA uses crossovers that will always give valid schedule rather than a crossover where scheduling has to be done separately.

Tabu Search is another promising optimization technique. This thesis has presented its application to scheduling and allocation in high-level synthesis. Investigation is done in finding a good initial solution to start with, defining a neighborhood for a given solution, generation of moves, formulation and maintenance of tabu list(s), defining a proper aspiration level criteria, finding a good tabu list size and an efficient way to accept moves. Two implementations are reported and compared. Tabu scheduling and allocation (TSA) is also tested on above mentioned benchmarks. Results obtained are comparable to those obtained by other systems. The results of both GSA and TSA are compared with simulated evolution (SE) [LM93], HAL system [PK89], SALSA II [RN93], STAR system [TH92], EMUCS system

[HT83], SAW [TLW+90] and CATREE system [GE87].

A novel interconnect optimization approach using genetic algorithm is also reported in this research. It can be used to optimize number of interconnections for a given schedule and functional unit allocation. It tries to find genetically good mappings for variables to registers and data transfers to buses with the aim of optimizing interconnection.

8.2 Future Work

Future work will focus on designing a complete data path synthesis system using GA. Efforts will be directed to include facilities such as chaining and loop winding. The data path synthesis system should be able to take high-level description and produce register-transfer level description of the circuit. Research can also be directed to find more effective implementation for TS and designing a complete data path synthesis system using TS with above mentioned facilities.

Bibliography

[AV93] Shawki Areibi and Anthony Vannelli. Circuit Partitioning Using A Tabu Search Approach. In *1993 IEEE International Symposium on Circuits and Systems*, pages 1643–1646, 1993.

[BD90] John E. Biegel and James J. Davern. Genetic algorithms and job shop scheduling. *Computers and Industrial Engineering*, 19(1–4):81–91, 1990.

[BM89] M. Balakrishnan and P. Marwedel. Integrated scheduling and binding: A synthesis approach for design space exploration. In *Proceedings of the 26th Design Automation Conference*, pages 68–74, June 1989.

[BS94a] M.S.T. Benten and Sadiq M. Sait. GAP: A genetic algorithm approach to optimize two-bit decoder PLAs. *International Journal of Electronics*, 76(1):99–106, 1994.

[BS94b] M.S.T. Benten and Sadiq M. Sait. Genetic scheduling of task graphs. *International Journal of Electronics*, 1994.

[Cam90] Raul Camposano. From behavior to structure: High-level synthesis. *IEEE Design and Test of Computers*, 7(5):8–19, October 1990.

[CHMR91] J. P. Cohoon, S. U. Hedge, W. N. Martin, and D. Richards. Distributed genetic algorithms for the floorplan design problem. *IEEE Transactions on Computer-Aided Design*, 10(4):483–492, 1991.

[CP87] James P. Cohoon and William D. Paris. Genetic placement. *IEEE Transactions on Computer-Aided Design*, 6(6):956-964. November 1987.

[CT90] Richard J. Cloutier and Donald E. Thomas. The combination of scheduling, allocation, and mapping in a single algorithm. In *Proceedings of the 27th Design Automation Conference*, pages 71–76. June 1990.

[CW91] Raul Camposano and Wayne Wolf. *High-Level VLSI Synthesis.* Kluwer Academic Publishers. 1991.

[Dav87] Lawerence Davis, editor. *Genetic Algorithms and Simulated Annealing*, chapter 1, pages 1–11. Pitman, London, 1987.

[Dav91] Lawerence Davis, editor. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[DLSM81] 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.

[DN89] Srinivas Devadas and A. Richard Newton. Algorithms for hardware allocation in data path synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7):768–781, 1989.

[FD92] E. Falkenauer and A. Delchambre. A genetic algorithm for bin packing and line balancing. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 1186–1192, May 1992.

[FHdW90] C. Friden, A. Hertz, and D. de Werra. TABARIS: An Exact Algorithm based on Tabu Search for Finding a Maximum Independent Set in a Graph. *Computers and Operations Research*, 19(1–4):81–91, 1990.

[GE87] C. H. Gebotys and M. I. Elmasry. A VLSI methodology with testability constraints. In *Proceedings of the 1987 Canadian Conference on VLSI(Winnipeg)*. October 1987.

[Glo77] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, 1977.

[Glo89] Fred Glover. Tabu Search - Part I. *ORSA Journal of Computing*, 1:190–206, 1989.

[Glo90a] Fred Glover. Artificial intelligence, heuristic frameworks and tabu search. *Managerial and Decision Economics*, 11:365–375, 1990.

[Glo90b] Fred Glover. Tabu Search - Part II. *ORSA Journal of Computing*, 2:4–32, 1990.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

[HdW87] A. Hertz and D. de Werra. Using Tabu Search Techniques for Graph Coloring. *Computing*, 29:345–351, 1987.

[HHA90] E.S.H. Hou, R. Hong, and N. Ansari. Efficient multiprocessor scheduling based on genetic algorithms. In *16th Annual Conference of IEEE Industrial Electronics Society - IECON'90.* pages 1239–1243, November 1990.

[HMS89] J. Hesser, R. Manner, and O. Stucky. Optimization of Steiner trees using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 231–236, 1989.

[HP78] L. J. Hafer and A. C. Parker. Register-transfer level digital design automation: The allocation process. In *Proceedings of the 15th Design Automation Conference*, pages 213–219, June 1978.

[HP83] L. J. Hafer and A. C. Parker. A formal method for the specification, analysis and design of register-transfer level digital logic. *IEEE Transactions on Computer-Aided Design*, 2(1):4–18, January 1983.

[HS71] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th D. A. Workshop (Las Vegas)*. pages 155–169, 1971.

[HT83] 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, June 1983.

[JC92] Lin-Ming Jin and Shu-Park Chan. Analogue placement by formulation of macro-components and genetic partitioning. *International Journal of Electronics*. 73(1):157–173, 1992.

[LC91] Andrew LIM and Yeow-Meng CHEE. Graph Partitioning Using Tabu Search. In *1991 IEEE International Symposium on Circuits and Systems*, pages 1164–1167. 1991.

[LM93] Tai A. Ly and Jack T. Mowchenko. Applying simulated evolution to high level synthesis. *IEEE Transactions on Computer-Aided Design*, 12(3):389–409, March 1993.

[MBL92] Tony Mason, Doug Brown, and John Levine. *lex & yacc*. O'Reilly and Associates, 2 edition, October 1992.

[MPC88] Michael C. McFarland, A. C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *Proceedings of the 25th Design Automation Conference*, pages 330–336, June 1988.

[MPC90] Michael C. McFarland, A. C. Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[Pan88] B. M. Pangrle. Splicer: A heuristic approach to connectivity binding. In *Proceedings of the 25th Design Automation Conference*, pages 536–541, June 1988.

[PG87] B. M. Pangrle and D. D. Gajski. Design tools for intelligent silicon compilation. *IEEE Transactions on Computer-Aided Design*, 6(6):1098–1112, November 1987.

[PK89] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 8(6):661–679, June 1989.

[RN93] Michael R. Rhinehart and John Nestor. SALSA II: A fast transformational scheduler for high-level synthesis. In *1993 IEEE International Symposium on Circuits and Systems*, pages 1678–1681, 1993.

[SV92] L. Song and A. Vannelli. VLSI Placement using Tabu Search. *Microelectronics Journal*, 17(5):437–445, 1992.

[SY45] Sadiq M. Sait and Habib Youssef. *VLSI Design Automation: Theory and Practice (in press)*. Mc-Graw Hill Book Co., Europe, 1994/5.

[TH92] Fur-Shing Tsai and Yu-Chin Hsu. STAR: An automatic data path allocator. *IEEE Transactions on Computer-Aided Design*, 11(9):1053–1064. September 1992.

[TLW+90] 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 System Architects's Workbench.* Norwell, MA: Kluwer Academic Publishers. 1990.

[Tri87] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Transactions on Computer-Aided Design*, 6(2):259–269, June 1987.

[TS86] C. Tseng and D.P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design*, 5(3):379–395, July 1986.

[WC91] Robert A. Walker and Raul Camposano, editors. *A Survey of High-Level Synthesis Systems*, pages 3–34. Kluwer Academic Publishers, 1991.

[WGH90] N. Wehn, M. Glesner, and M. Held. A novel scheduling/allocation approach for datapath synthesis based on genetic paradigms. In *IFIP Working Conference on Logic and Architecture Synthesis, Paris*, pages 47--56, 1990.

[WSF89] Darrell Whitely, Timothy Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 133–140, 1989.

Vita

- Shahid Ali.
- Born in Karachi, Pakistan.
- Received Bachelor's degree in Computer Systems Engineering from N.E.D. University of Engineering & Technology, Karachi, Pakistan, in 1991.
- Worked as Software Development Engineer in Digital Communications Pvt.
- Joined Information & Computer Science Department at KFUPM in December 1991.
- Completed Master's degree requirements at King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia in Spring 1994.