Genetic Algorithm Based Design and Optimization of

VLSI ASICs and Reconfigurable Hardware

by

Pradeep Ruben Fernando

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor:  Srinivas Katkoori, Ph.D.
Nagarajan Ranganathan, Ph.D.
Hao Zheng, Ph.D.
Sanjukta Bhanja, Ph.D.
Stephen Suen, Ph.D.

Date of Approval:
October 17, 2008

Keywords:  Evolutionary Algorithms, Multi-objective VLSI Floorplanning, Hardware IP Core Implementation, Evolvable Hardware Design, Extreme Environment Electronics

© Copyright 2009, Pradeep Ruben Fernando
DEDICATION

To my family
ACKNOWLEDGEMENTS

I would like to thank Dr. Katkoori immensely for all his support and encouragement throughout the entire course of my research. I would like to thank all my committee members, Dr. Ranganathan, Dr. Zheng, Dr. Bhanja, and Dr. Suen, for agreeing to be on my committee and for their insightful comments. I would like to thank Dr. Keymeulen, Dr. Zebulum and Dr. Stoica of Jet Propulsion Labs for all their help while working on the Self Reconfigurable Electronics for Extreme Environments project. I would also like to thank the entire CSE technical support team consisting of Daniel, Peter, and Brian, for their quick responses to all my computer and software needs. I would like to specially thank all my friends and all my VLSI lab colleagues for their friendship, support, and encouragement which helped me through many hard times. Last but not least, I would like to thank my parents and my entire family for their unconditional emotional and financial support.
TABLE OF CONTENTS

LIST OF TABLES iii

LIST OF FIGURES iv

ABSTRACT vii

CHAPTER 1 INTRODUCTION 1
  1.1 Hardware Implementation Methodologies 2
  1.2 Motivation 4
  1.3 Contributions of the Dissertation 5
  1.4 Organization of the Dissertation 6

CHAPTER 2 GENETIC ALGORITHMS 7
  2.1 Components of a GA Based Optimization Engine 9
    2.1.1 Individual Encoding 10
    2.1.2 Fitness of an Individual 10
    2.1.3 Selection Mechanism 11
    2.1.4 Genetic Operators 11
    2.1.5 Crossover Operators 12
      2.1.5.1 One-Point Crossover 12
      2.1.5.2 Two-Point Crossover 13
      2.1.5.3 Uniform Crossover 14
  2.2 Elitism in Genetic Algorithms 15
  2.3 Multi-Objective Genetic Algorithms 16
  2.4 Summary 16

CHAPTER 3 MULTI-OBJECTIVE GENETIC FLOORPLANNING FOR VLSI ASICS 17
  3.1 Multi-objective Optimization 19
  3.2 Floorplanning Using Sequence Pair Representation 22
    3.2.1 Conversion from a Floorplan to a Sequence Pair 22
    3.2.2 Conversion from a Sequence Pair to a Floorplan 23
  3.3 Related Work 24
  3.4 Proposed Multi-Objective Genetic Floorplanner 30
  3.5 Proposed Crossover Operators 38
    3.5.1 Modified Two-Point Order-Based Crossover (MTOX) Operator 38
    3.5.2 Heuristic One-Point Order-Based Crossover (HOOX) Operator 44
  3.6 Empirical Determination of GA Parameter Settings 50
  3.7 Experimental Results 54
    3.7.1 Performance of the Proposed Genetic Operators 56
    3.7.2 Comparisons Against State-of-the-Art Floorplanners 57
  3.8 Summary 61
CHAPTER 4  DESIGN OF AN FPGA BASED GENERAL PURPOSE GENETIC ALGORITHM IP CORE  62
  4.1  Background and Related Work  64  
  4.1.1  Prior Work  64  
  4.1.2  Pseudo-Random Number Generation and GA Performance  69  
  4.1.3  Basics of Evolvable Hardware  71  
  4.2  Proposed FPGA Based Genetic Algorithm IP Core  72  
  4.2.1  Implementation and Interfacing Details  73  
  4.2.2  Design Considerations for ASIC Implementation and Space Applications  79  
  4.3  Experimental Results  83  
  4.3.1  RT-Level Simulations  83  
  4.3.2  FPGA Implementation Results  90  
  4.3.3  Runtime Comparison With Software Implementation  97  
  4.4  Summary  98  

CHAPTER 5  A DIGITAL FRAMEWORK FOR EVOLUTIONARY DESIGN, AUTONOMOUS MONITORING AND PERFORMANCE COMPENSATION OF EXTREME ENVIRONMENT ELECTRONICS  99  
  5.1  Related Work  100  
  5.2  Architecture and Implementation of Proposed Digital Framework  103  
  5.2.1  Overall System Level Architecture and Operation  103  
  5.2.2  Detailed Architecture of the Proposed Framework  106  
  5.3  Digital ASIC Implementation  117  
  5.3.1  Design Methodology  118  
  5.4  Simulations and Results  119  
  5.4.1  Layout-Level Simulations  120  
  5.5  Summary  122  

CHAPTER 6  CONCLUSIONS  123  

REFERENCES  125  

ABOUT THE AUTHOR  End Page
LIST OF TABLES

Table 3.1. Brief summary of relevant floorplanning works that simultaneously optimize area and wirelength

Table 3.2. Best area and total wirelength results obtained by the proposed genetic floorplanner for the n300 benchmark for various crossover and mutation rates (A – Area in sq.units, W – Total Wirelength in units)

Table 3.3. Area and wirelength comparisons with the AdaptGA, QP-LFF, and Parquet floorplanners (A = Area in mm², W = Total Wirelength in mm, %S = Percentage Savings, “.” = not reported)

Table 3.4. Area, wirelength and runtime results for the proposed genetic floorplanner on the GSRC benchmarks

Table 3.5. Percentage savings in Area and Wirelength of the Proposed Genetic Floorplanner compared against the PARQUET Floorplanner for the GSRC benchmarks

Table 4.1. Review of existing literature on FPGA implementation of a general-purpose genetic algorithm (“N/A” : Not Applicable, “.” : UNKNOWN)

Table 4.2. Port interface of the proposed GA core

Table 4.3. Index values of GA core’s programmable parameters

Table 4.4. Preset modes available in the proposed GA core

Table 4.5. RT-level simulation results obtained for the three test functions (BF6, F1, and F2) under various GA parameter settings

Table 4.6. Post place-and-route statistics for the proposed GA core on Virtex II Pro device (xc2vp30-7ff896)

Table 4.7. Best fitness values obtained by the GA for the mBF6_2 function for different parameter settings (XR = Crossover Rate)

Table 4.8. Best fitness values obtained by the GA for the mBF7 function for different parameter settings (XR = Crossover Rate)

Table 4.9. Best fitness values obtained by the GA for the Shubert function for different parameter settings (XR = Crossover Rate)
LIST OF FIGURES

Figure 1.1. High level view of a typical design cycle for VLSI hardware 2
Figure 2.1. High level view of a typical GA optimization cycle 9
Figure 2.2. Example illustrating single-point crossover 13
Figure 2.3. Two-point crossover on binary encoded chromosomes 14
Figure 2.4. Example illustrating uniform crossover 15
Figure 3.1. Pareto-optimal solutions and non-domination levels in multi-objective optimization 22
Figure 3.2. Gridding process to obtain the Sequence Pair corresponding to a floorplan 23
Figure 3.3. Pseudo-code of the proposed hybrid elitist non-dominated sorting GA-based floorplanner 31
Figure 3.4. Computing crowding distance of a solution in the objective space 33
Figure 3.5. Procedure to compute the crowding distances of solutions belonging to a non-domination front 34
Figure 3.6. Procedure for Slack_Move1 used for local optimization by the proposed floorplanner 37
Figure 3.7. Legality violations when using two point crossover on permutation-based chromosomes 39
Figure 3.8. Order crossover proposed for permutation-based chromosomes 40
Figure 3.9. Generation of the first offspring using the MTOX operator 42
Figure 3.10. Generation of sub-floorplans using single cut-point in the HOOX operator (steps 1-3 of Procedure HOOX) 46
Figure 3.11. Generation of offspring configurations in the HOOX operator (steps 4-5 of Procedure HOOX) 47
Figure 3.12. Convergence plot of wirelength for n300 benchmark with cxRate=0.8, mutRate=0.1 51
Figure 3.1. Convergence plot of area for n300 benchmark with cxRate=0.8, mutRate=0.1
Figure 3.2. Convergence plot of wirelength for n300 benchmark with cxRate=0.5, mutRate=0.1
Figure 3.3. Convergence plot of area for n300 benchmark with cxRate=0.5, mutRate=0.1
Figure 3.4. Convergence plot of wirelength for n300 benchmark with cxRate=0.5, mutRate=0.01
Figure 3.5. Convergence plot of area for n300 benchmark with cxRate=0.5, mutRate=0.01
Figure 3.6. Floorplan of ami33 benchmark with area=1.21mm² and wirelength=35.43mm
Figure 3.7. Floorplan of n100 GSRC benchmark with area = 205,758 sq.units and wirelength = 133,497 units
Figure 4.1. High level view of the implemented GA optimization cycle
Figure 4.2. Typical system showing the communication between the different modules and the GA core (signal numbers are in reference to Table 4.2)
Figure 4.3. Implementation of a hybrid intrinsic EHW system (with internal and external fitness modules) using the proposed GA core
Figure 4.4. (Zoomed in) Plot of the modified binary F6 [5] test function
Figure 4.5. Convergence plot for the BF6 test function using number of generations=32 – Run #3 of Table 4.5
Figure 4.6. Convergence plot for the BF6 test function with initial seed for RNG=1567 – Run #4 of Table 4.5
Figure 4.7. Convergence plot for the BF6 test function with crossover rate=0.75 – Run #5 of Table 4.5
Figure 4.8. Convergence plot for the test function F2 with population size=32 – Run #6 of Table 4.5
Figure 4.9. Convergence plot for the test function F2 with population size=64 – Run #7 of Table 4.5
Figure 4.10. Convergence plot for the test function F3 with initial seed for RNG=10593 – Run #9 of Table 4.5
Figure 4.11. Convergence plot for the test function F3 with initial seed for RNG=1567 – Run #10 of Table 4.5
Figure 4.12. Convergence plot for the test function \( mBF6_2(x) \) with initial RNG seed=\((061F)_{16}\), crossover threshold=10, and popSize=64 (data collected from hardware execution) 95

Figure 4.13. Convergence plot for the test function \( mBF6_2(x) \) with initial RNG seed=\((A0A0)_{16}\), crossover threshold=10, and popSize=64 (data collected from hardware execution) 95

Figure 4.14. Convergence plot for the test function \( mBF7_2(x,y) \) with initial RNG seed=\((AAAA)_{16}\), crossover threshold=12, and popSize=64 (data collected from hardware execution) 96

Figure 4.15. Convergence plot for the test function \( mShubert2D(x1,x2) \) with initial RNG seed=\((AAAA)_{16}\), crossover threshold=10, and popSize=64 (data collected from hardware execution) 96

Figure 5.1. Block diagram of overall system (including analog and digital systems) 104

Figure 5.2. High level functional view of the proposed digital framework 106

Figure 5.3. Architecture of the proposed digital framework 107

Figure 5.4. FSM for the digital framework controller module 108

Figure 5.5. FSM for the system monitor module 111

Figure 5.6. Chromosome encoding used by the GA based compensation algorithm 112

Figure 5.7. Slew rate measurement by the internal fitness evaluation module of the proposed digital framework 115

Figure 5.8. FSM of the FEM_Controller module in the internal fitness evaluation module 116

Figure 5.9. Design flow for digital ASIC implementation 118

Figure 5.10. Layout level simulation of the digital framework controller module 120

Figure 5.11. Layout level simulation of the system monitor module (in a monitoring cycle) 121

Figure 5.12. Layout level simulation of the system monitor module (in a compensation cycle) 121

Figure 5.13. Layout level simulation of the internal fitness evaluation module (containing FEM_Controller, excitation, and slew rate modules) 122
GENETIC ALGORITHM BASED DESIGN AND OPTIMIZATION OF
VLSI ASICS AND RECONFIGURABLE HARDWARE

PRADEEP RUBEN FERNANDO

ABSTRACT

Rapid advances in integration technology have tremendously increased the design complexity of very large scale integrated (VLSI) circuits, necessitating robust optimization techniques in many stages of VLSI design. A genetic algorithm (GA) is a stochastic optimization technique that uses principles derived from the evolutionary process in nature. In this work, genetic algorithms are used to alleviate the hardware design process of VLSI application specific integrated circuits (ASICs) and reconfigurable hardware.

VLSI ASIC design suffers from high design complexity and a large number of optimization objectives requiring hierarchical design approaches and multi-objective optimization techniques. The floorplanning stage of the design cycle becomes highly important in hierarchical design methods. In this work, a multi-objective genetic algorithm based floorplanner has been developed with novel crossover operators to address the multi-objective floorplanning problem for VLSI ASICs. The genetic floorplanner achieves significant wirelength savings (>19% on average) with little or no increase in area (<3% penalty) over previous floorplanners that perform simultaneous area and wirelength minimization.

Hardware implementation of genetic algorithms is gaining importance because of their proven effectiveness as optimization engines for real-time applications. Earlier hardware implementations suffer from major drawbacks such as absence of GA parameter programmability,
rigid pre-defined system architecture, and lack of support for multiple fitness functions. A compact IP core that implements a general purpose GA engine has been designed to realize evolvable hardware in field programmable gate array devices. The designed GA core achieved a speedup of around 5.16x over an analogous software implementation.

Novel reconfigurable analog architectures have been proposed to realize extreme environment analog electronics. In this work, a digital framework has been developed to realize self reconfigurable analog arrays (SRAA) where genetic algorithms are used to evolve the required analog functionality and compensate performance degradation in extreme environments. The framework supports two methods of compensation, namely, model based lookup and genetic algorithm based compensation and is scalable in terms of the number of fitness evaluation modules. The entire framework has been implemented as a digital ASIC in a leading industry-strength silicon-on-insulator (SOI) technology to obtain high performance and a small form factor.
CHAPTER 1
INTRODUCTION

Hardware solutions are required for many applications in a wide variety of problem domains. Optimal design and implementation of hardware solutions is a complex and difficult process that is usually broken down into a sequence of smaller design stages. Each of these design stages has well-defined objectives to help the designer concentrate on particular issues in each stage. Figure 1.1 shows a high level abstract view of a typical design cycle for developing VLSI (Very Large Scale Integrated) hardware solutions. A brief outline of the design cycle is below:

- **System Specification:** The input to the design cycle is a set of user specifications that details the functionality, performance, reliability, and other requirements of the system.
- **Behavioral Modeling:** In this stage, the behavior/functionality of the entire system is modeled using a high level specification language such as C, VHDL, etc.
- **Circuit Synthesis:** In this stage, the minimal logic required to implement the system functionality is obtained. A minimal circuit description is synthesized to implement the optimized logic of the entire system. Optimization techniques are required for obtaining minimized logic and circuit solutions of the system.
- **Physical Design:** In this stage, a geometrical description of the circuit is developed that satisfies all the user-defined requirements including functionality, performance, and reliability. Efficient optimization techniques are needed for obtaining the geometrical circuit description that has the optimal values for multiple objectives such as area, performance, etc.
• **Fabrication:** The final hardware solution is obtained from the geometrical description of the entire system using the chosen hardware implementation methodology.

![Design Cycle Diagram](image)

Figure 1.1. High level view of a typical design cycle for VLSI hardware

1.1 **Hardware Implementation Methodologies**

Hardware solutions can be implemented using different design methodologies such as ASICs (Application Specific Integrated Circuits) and FPGAs (Field Programmable Gate Arrays). The hardware design cycle will differ from the generic cycle shown in Figure 1.1 based on the choice of implementation methodology depending on the nature of the application. VLSI ASICs
provide the highest design flexibility of all the implementation methodologies as they allow the
user to decide the placement of all the logic gates and the interconnections. This allows the
designer to achieve the best performance possible at the expense of long design turn-around times
and high costs. Applications such as critical real-time applications that require high performance,
irrespective of the cost and design time involved, will opt for the VLSI ASIC implementation
methodology.

FPGAs consist of arrays of programmable logic modules with pre-wired interconnections
between them. FPGAs are programmed to perform a logic function by first mapping the logic
function on to a subset of the programmable logic modules and then programming switchboxes,
which control the connectivity of the pre-wired interconnections, to achieve the desired
connectivity. FPGAs provide the lowest design flexibility as the placement of both the logic gates
and the interconnections is fixed. Hardware design using FPGAs entails realizing the entire logic
of the target application using the logic gates and interconnections that are available in the FPGA
device. But FPGAs provide the flexibility of changing the physical realization of the system just
by changing the configuration of the logic gates and the switchboxes controlling the
interconnections. Although FPGAs suffer low performance due to the switchboxes in the
interconnection paths, they can be (re)programmed immediately with a new design resulting in the
lowest design turn-around times and a low cost. Thus, applications with reconfiguration needs and
lower priority on system performance will opt for FPGA based implementation.

The complexity of the VLSI hardware design process has also increased significantly with
technological advances [1]. With the advent of the nanometer era, more than a billion transistors
are being fabricated on a single integrated circuit. Hence the complexity of the designs being
implemented has increased significantly leading to entire systems being implemented on a single
chip. To ease the difficulty of the entire design process, designers have resorted to a number of
techniques throughout the hardware design cycle. For easing the complexity of the behavioral
modeling stage, designers partition the systems being designed into a number of smaller design units with well defined functionality so that they can be implemented using existing IP modules. Hence development of efficient IP modules is becoming an important task, especially in FPGA based designs.

1.2 Motivation

Irrespective of the hardware implementation choice, optimization is required at all the design levels to obtain the best implementation of the target application. Many of the optimization problems in the design stages are NP-complete problems. Greedy heuristics can be used to quickly obtain a solution for the NP-complete problems. However greedy heuristics do not possess any mechanism to escape out of locally optimal solutions. Hence, they may result in solutions of very poor quality as they can get stuck in local optima. Stochastic optimization methods such as simulated annealing and genetic algorithms make use of random moves to escape out of local optima. Given enough time, the stochastic optimization methods generally have the ability to reach the globally optimal solution.

Although the need for a good global optimization technique in the hardware design process is apparent, the criticality of the optimization techniques will vary with the design levels and the type of hardware implementation. In VLSI ASIC design, the circuit synthesis and physical design stages are the most crucial stages as they play a significant role in obtaining the best system implementation.

In FPGA based hardware design, design flexibility and optimization opportunities are limited compared to full custom ASICs as the placement of logic cells and interconnections is fixed. Hence, the FPGA design cycle is very sensitive to the high level model of the system. Thus, producing an accurate and succinct behavioral model of the system is very critical in FPGA based hardware design. This re-iterates the necessity for well designed IP modules that can be re-used by
the designer during high level modeling. The availability of an IP module library reduces the complexity of high level modeling. With entire systems being implemented on the same FPGA, many FPGA designs require stochastic optimization cores as part of the application itself. Since the requirements of the applications are widespread, a robust stochastic optimization technique that is easy to implement in hardware is needed.

A genetic algorithm (GA) [2-5] is a stochastic optimization technique modeled on the theory of evolution in nature. It has been successfully employed for a wide variety of problems including NP-complete problems such as the Traveling Salesman problem [6], real-time problems such as reconfiguration of evolvable hardware [7] and other optimization problems that have complex constraints. Genetic algorithms are easy to implement in software and hardware. They can also be easily adapted to a wide variety of problems. They can make use of existing knowledge about a problem by incorporating successful existing operators into their optimization framework. They are a multi-agent optimization technique and hence can be easily modified for multi-objective optimization.

1.3 Contributions of the Dissertation

In this dissertation, genetic algorithm based optimization is applied to the VLSI hardware design domain at various design levels:

- **VLSI ASIC Design**: A multi-objective genetic floorplanner has been developed for simultaneous optimization of area and wirelength in VLSI ASICs (layouts).

- **FPGA based Hardware Applications**: A customizable general purpose GA IP core has been developed for FPGA based applications that need a stochastic optimization engine.

- **Extreme Environment Electronics**: A genetic algorithm based VLSI ASIC has been developed to evolve analog circuits, monitor their performance in extreme environments, and compensate any performance degradation.
1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows: Chapter 2 introduces genetic algorithms in detail and discusses both the weighted sum and pareto-front based multi-objective genetic algorithms. Chapter 3 defines the problem of multi-objective (area and wirelength) VLSI floorplanning and presents the genetic floorplanner developed to solve the problem. Chapter 4 discusses the development of a hardware IP core for a general purpose genetic algorithm that can be used as a stochastic optimization engine for a wide variety of applications. Chapter 5 describes the design and development of a digital framework for evolution, autonomous monitoring and automatic compensation of extreme environment electronics. Chapter 6 draws conclusions from the research work of this dissertation.
CHAPTER 2
GENETIC ALGORITHMS

A genetic algorithm is a stochastic optimization technique inspired by the principles of evolution. John Holland proposed the first simulated evolution algorithm [2] that mimicked the evolutionary process in nature. Since then, genetic algorithms have been developed to successfully address many different problems including combinatorial optimization problems [6], real time problems [7], and problems with unknown characteristics [3].

The phenomenon of evolution in nature refers to the process by which living organisms change their physical characteristics, referred to as their phenotype, in response to the changing environment. According to the theory of genetics, the physical characteristics of all organisms are based on the makeup of cellular structures called chromosomes. The chromosome structure is a collection of smaller organic structures called genes. The structure of each gene is directly responsible for a particular physical characteristic of the organism.

All living organisms share the finite amount of resources present in this world. The survival of an individual is dependent on its ability to compete with other individuals for these limited resources. The physical characteristics of the individual that help it to compete with other individuals are directly dependent on its genetic makeup. When individuals reproduce in nature, the chromosomes of the parents combine in a random manner to create a new chromosome for the offspring. Thus, the offspring’s chromosome will inherit genes (and hence physical characteristics) only from the gene pool belonging to its parents and their ancestors. If the offspring chromosome obtained a good mixture of its parents’ genes, then its ability to compete for resources will be
good, leading to an increased period of survival. Moreover, fitter individuals have a higher probability of producing fitter offspring and preserving the genetic makeup of that species. Offspring that gets a poor mixture of its parents’ chromosomes will not be able to compete with fitter individuals and will die soon. In the long run, species with such a poor chromosomal makeup will become extinct. This is the principle of survival of the fittest in nature.

A genetic algorithm mimics this natural evolutionary process in its optimization cycle. Figure 2.1 shows a high level abstract view of a typical GA optimization cycle. It models the problem at hand as an ecosystem with limited resources. It models the solutions to the problem as chromosomes of individuals that are competing for the limited available resources. The solutions that have better values for the objective functions relevant to the problem are deemed to be fitter individuals. Thus, each individual is assigned a fitness measure that denotes the quality of the solution that it encodes. A genetic algorithm begins its optimization cycle with an initial population of randomly generated individuals. Genetic algorithms evolve the population during each generation using genetic operators such as crossover and mutation. In each generation, highly fit individuals from the current generation are selected as parents to produce new individuals for the next generation. Genetic information encoded in the parents’ representation is mixed to form one or more offspring using an operator called crossover. This operator mimics the mating process in nature. The offspring produced by the crossover operator replaces individuals with the worst fitness in the current population. This ensures that the average fitness of the population increases over generations as the fitter individuals survive through the generations. To prevent premature convergence to the best individual in the initial population, an operator called mutation introduces random changes into the population. After a pre-specified number of generations, the best individual found in the entire genetic optimization cycle is output as the solution to the problem.
2.1 **Components of a GA Based Optimization Engine**

A simple genetic algorithm based optimizer is characterized by the following components:
- Individual encoding,
- Individual fitness,
- Selection mechanism, and
- Genetic operators.

2.1.1 Individual Encoding

Genetic algorithms encode solutions to the given problem as chromosomal strings and operate on these encodings during the optimization process. This helps minimize the amount of problem specific information needed during the optimization process of a genetic algorithm. An encoding scheme that maps each chromosome string to a unique solution is preferred as the genetic algorithm will not waste time evaluating multiple encodings of the same solution. The chosen encoding scheme should also be easy to decode using minimal time and memory resources. For example, for a traveling salesman problem (TSP), a permutation of all the cities in the problem instance can be used a solution encoding scheme.

2.1.2 Fitness of an Individual

A fitness measure is needed to evaluate each chromosome encountered by the GA optimizer. The fitness measure of the chromosome should reflect the quality of the corresponding solution to the problem. For example, in a TSP instance, the length of the overall tour represented by the permutation encoding could be assigned as the fitness measure. The genetic algorithm then works on finding the permutation encoding that results in a minimal tour length for the given problem instance. Since genetic algorithms follow natural selection, the fitness value of an individual will decide whether or not it will survive through the generations. The fitness of an individual may also decide whether or not the individual will be chosen to participate in genetic operations.
2.1.3 Selection Mechanism

The selection mechanism is the scheme used by a genetic algorithm to select two individuals for crossover (mating). The purpose of these operations is to allow substrings in the fit individuals in a population to survive for many generations. Hence, the parent individuals for these operations are generally selected based on their fitness values. This will promote survival of fitter genes in the offspring and should lead to fitter individuals in the future generations. Many selection mechanisms exist including proportionate selection, roulette wheel selection, and rank-based selection. The selection mechanism is chosen based on the nature of the problem being optimized and other factors including computation time and memory requirements. Two most effective parent selection schemes [3] are described below:

- **Roulette Wheel Selection** – In this scheme, the probability that an individual will be chosen as a parent for the current crossover operation is equal to the proportion of the individual’s fitness as compared to the total fitness value of the current population. The proportion of offspring produced in this scheme by a fit individual as compared to the offspring by a less fit individual will be proportional to the ratio of their corresponding fitnesses.

- **Tournament Selection** – In this scheme, $n$ individuals are randomly chosen from the current population to play in a tournament against each other. The winner of the tournament is selected as the parent. A tournament involving $n$ individuals is called an $n$-way tournament.

2.1.4 Genetic Operators

Genetic algorithms use two kinds of genetic operators called crossover and mutation. The crossover operator performs a probabilistic exchange of chromosomal information between two individuals to produce a new individual. The crossover operator selects two parent individuals
from the population based on a selection scheme. It then produces an offspring individual by using
certain information from the first parent and the rest of the information from the second parent.
Thus, the offspring individual inherits a subset of properties from both of its parents. The mutation
operator typically picks a random individual from the population and performs an inversion or
some other random operation on the individual chromosome. After a certain number of
generations, the crossover operator tends to produce offspring that are very similar to the parent
individuals. Then the mutation operator plays a critical role in restoring lost genetic material or
providing diversity in the current population. Thus, the mutation operator helps prevent
convergence to local optima.

2.1.5 Crossover Operators

Traditional genetic algorithms worked on binary encodings of the problem instances.
These genetic algorithms use simple crossover operators such as the uniform crossover and the
one-point crossover to produce the offspring individual.

2.1.5.1 One-Point Crossover

The one-point crossover operator randomly generates a single cut-point on both the parent
individuals’ chromosomes. The first offspring is produced by concatenating the first part of the
first parent’s chromosome and the second part of the second parent’s chromosome. A second
offspring may be optionally produced by concatenating the first part of the second parent’s
chromosome and the second part of the first parent’s chromosome. Figure 2.2 illustrates the one-
point crossover.
2.1.5.2 Two-Point Crossover

The one-point crossover operator cannot produce offspring that contains discontinuous fragments of its parents. To provide more flexibility to producing all combinations of offspring, the two-point crossover was proposed [3]. The two point crossover operator randomly chooses two cut-points that split both the parents into three parts each. Two offspring are produced from the two parents as shown in Figure 2.3. The first offspring is formed by concatenating the first part of the first parent, the second part of the second parent, and the third part of the first parent. The second offspring is formed by concatenating the first part of the second parent, the second part of the first parent, and the third part of the second parent.
2.1.5.3 Uniform Crossover

The uniform crossover operator offers the most flexibility in obtaining all combinations of the parent individuals in the offspring by assigning equal probability to both the parents to pass each of their genes to the offspring. In a simple implementation of the uniform crossover operator, a random binary string is generated with a length equal to the length of the parents’ chromosomes. If the bit value at a position $i$ of the string is 0, then the gene at position $i$ of the offspring chromosome is filled with the $i^{th}$ gene from the first parent. If the value was 1, then the $i^{th}$ gene from the second parent will be used. Uniform crossover is a very disruptive operator [4] and tends to lose important genetic information after a certain number of generations. An example of the uniform crossover operator is shown in Figure 2.4.
Figure 2.4. Example illustrating uniform crossover

2.2 Elitism in Genetic Algorithms

An elitist genetic algorithm preserves some of the best (elite) solutions of the current generation into the next generation. The number of elite solutions preserved in the following generation is called the degree of elitism. Rudolph [8] proved that an elitist GA will converge to the global optimal solution in a single objective optimization problem where the objective is a real-valued function. Rudolph [9] also proved that an elitist evolutionary algorithm will converge to the pareto-optimal front of a multi-criterion optimization problem where all the objectives are real-valued functions. Intuitively, preserving the elite individuals increases the probability of generating better offspring in each generation thus leading to faster convergence towards the optimal front.
2.3 Multi-Objective Genetic Algorithms

Many different templates, both elitist and non-elitist, have been proposed for multi-objective optimization using genetic algorithms [10]. Vector evaluated genetic algorithms (VEGA) [11] was one of the first multi-objective genetic algorithms. VEGA, a non-elitist multi-objective GA, used a vector of objective function values instead of a scalar weighted sum fitness value. Later, other non-elitist GA templates such as the Non-Dominated Sorting GA [12] and the Niched-Pareto GA [13] were proposed. Elitist multi-objective GAs such as the Strength Pareto Evolutionary Algorithm (SPEA) [14] and the Non-Dominated Sorting based GA-II (NSGA-II) [15] outperformed the non-elitist multi-objective GAs.

2.4 Summary

Genetic algorithms have been extensively researched and applied to a wide variety of single objective and multi-objective optimization applications. In this dissertation, the application of genetic algorithms to three VLSI design problems is investigated.
CHAPTER 3

MULTI-OBJECTIVE GENETIC FLOORPLANNING FOR VLSI ASICS

The continuous technology scaling over the years has led to the nanometer era in VLSI design. The 2007 edition of the ITRS (International Technology Roadmap for Semiconductors) [1] predicts that transistor lengths will scale down to 10 nanometers by 2015. This implies that the number of transistors that can be packed into the same area of an integrated circuit will increase tremendously. This results in a significant increase in the complexity of the systems that can be fabricated on a single chip. As of 2008, entire systems are being fabricated on a single chip under the System-on-a-Chip (SOC) paradigm. This high design complexity affects the circuit synthesis phase of VLSI ASICs significantly since technology scaling introduces new issues regarding performance and reliability.

To handle the high complexity of designing VLSI ASICs, designers utilize hierarchical design methods so that only portions of the entire design have to be considered at any one time. Also, designers are increasingly identifying components in the target application that can be mapped to pre-defined IP (Intellectual Property) modules, which have been previously optimized. With this trend in VLSI design, the floorplanning phase of the VLSI physical design cycle has become a critical step as it has a major influence on interconnect issues including wiring congestion, crosstalk, and performance. These issues are the bottleneck to realizing the full potential of the technology improvements [1]. Traditional floorplanners work on optimizing either the bounding box area or the total wirelength of the floorplan. Current VLSI floorplanners must include multiple metrics in their objective function including interconnect metrics such as total
wirelength and longest path delay in the objective function in addition to the area metrics. Hence an optimization engine is necessary for the floorplanning problem that can handle multiple metrics.

Genetic algorithms are a natural choice for multi-objective optimization as they maintain a population of individuals at all times and can be easily modified to perform multi-objective optimization [10]. In this work, a multi-objective genetic algorithm is proposed for the outline-free, macro-cell based VLSI floorplanning problem that simultaneously optimizes both area and total wirelength.

The main contributions of this work are:

- An NSGA-II (Elitist Non-dominated Sorting based Genetic Algorithm) based multi-objective genetic floorplanner that performs simultaneous minimization of area and wirelength,
- A novel heuristic crossover operator (HOOX) that promotes the multiplication of good sub-floorplans, and
- A new crossover operator (MTOX) that is a novel combination of the classical two point crossover operator and order crossover operator.

The proposed multi-objective GA-based floorplanner is one of very few floorplanners [38] [39] to use non-domination concepts to rank floorplan solutions. The efficiency of the proposed method is demonstrated by the wirelength savings obtained with marginal or no area penalty. The proposed floorplanner obtained an average wirelength savings of 25.3% over an existing genetic floorplanner [29] that uses the same Sequence Pair encoding for its floorplan solutions, illustrating the efficiency of the proposed crossover operators. The proposed floorplanner obtained 26.9% average wirelength savings over a Quadratic Programming based floorplanner [24] for the MCNC benchmarks for a 2.63% average increase in area. The proposed floorplanner obtained an average savings of 19.17% in wirelength and 1.375% in area over a simulated annealing based floorplanner [21], averaged over both the MCNC and GSRC benchmarks. To the best of the
author’s knowledge, the results reported here are the best results compared to those reported in the literature on simultaneous area and wirelength optimization during outline-free floorplanning for the MCNC and GSRC benchmark suites.

### 3.1 Multi-objective Optimization

Multi-objective optimization is the process of finding a set of solutions that are optimal in terms of more than one conflicting objectives. If the multiple objectives are non-conflicting in nature, then the problem is the same as a single objective optimization problem. This is due to the fact that there exists exactly one optimal solution for such a problem that can be found by optimizing any one of the objectives. For problems that require multiple conflicting objectives to be optimized, there exists a set of globally optimal solutions called the pareto-optimal set.

- **Domination** – A solution \( p \in S \) is said to dominate another solution \( q \in S \), denoted as \( p \prec q \), if and only if, \( \forall i \in \{1, \ldots, n\}, f_i(p) \leq f_i(q) \), and \( \exists j \in \{1, \ldots, n\}: f_j(p) < f_j(q) \), where \( f_i \) is the \( i \)-th objective function under consideration and \( S \) is the solution space for the problem under consideration. Without any loss of generality, all the \( n \) objective functions are assumed to be minimization functions in the above definition.

- **Non-dominated Set** – A set of solutions \( P \subseteq Q \), is called a non-dominated set if \( \forall p \in P, q \in Q, q \) does not dominate \( p \).

All the solutions \( p \in P \), in the above definition, are said to lie in the same non-domination level or front. When \( Q = S \), the solutions in set \( P \) are not dominated by any other solution in \( S \) and are said to lie in the non-domination level zero. In this case, the solutions in \( P \) dominate all the other solutions in \( S \) and the set \( P \) is called the global pareto-optimal set.

Traditional optimization techniques transform the multi-objective optimization problem into a single objective optimization problem due to a dearth of solution methodologies that are
directly applicable to the multi-objective optimization problem. Consider the multi-objective optimization problem in Equation 3.1.1.

\[
\text{Minimize } f = \langle f_1, f_2, \ldots, f_n \rangle
\]  

where \( f_i, \ i \in \{1, \ldots, n\} \) form the set of objective functions to be minimized. In the traditional single objective optimization method, the problem would be transformed into

\[
\text{Minimize } C = \sum_i w_i \frac{f_i}{m_i}
\]  

where \( w = \langle w_1, w_2, \ldots, w_n \rangle \) is a weight vector provided by the user to specify the preferences among the objectives, and

\( m = \langle m_1, m_2, \ldots, m_n \rangle \) is a normalization vector used to transform the range of values that each objective function can take. In general, \( m_i \) is set to the maximum value of the corresponding objective function \( f_i \).

One of the major disadvantages of the single normalized weighted sum (SNWS) approach is that finding the correct weight vector to be used and predicting the maximum value or even a tight upper bound for some objective functions might be non-trivial. For instance, normalizing the wirelengths of floorplans is difficult as a tight upper bound for the total wirelength of circuits does not exist. In such a case, the preference given to the objective functions by the user assigned weight vector \( w \) may result in an undesirable bias towards a particular objective. The weighted single objective transformation will bias search engines such as Simulated Annealing towards a particular solution of the pareto-optimal set. This might lead to rejection of nearby solutions that are closer to the global pareto-optimal set causing increased difficulties in finding the desired solution. In fact, a simulated annealing based optimization engine will only consider solutions in a small subset of the entire solution space based on the user assigned preference vector. Hence even a minute discrepancy in the user-assigned preference vector might lead to large sub-optimalities.
Another major disadvantage of the single normalized weighted sum approach to multi-objective optimization is that it can yield only one of the multiple pareto-optimal solutions. A more direct approach to multi-objective optimization is to find the pareto-optimal set and then allow the user to choose one solution from this set.

Consider an optimization problem with two conflicting objectives. Figure 3.1 plots the solution space (with six solutions numbered 1-6) of such an optimization problem with two conflicting objectives, namely area and wirelength, both of which have to be minimized. It is clear from Figure 3.1 that the solutions numbered 1 and 2 dominate all the other solutions. But solution 1 cannot be deemed better than solution 2 as solution 1 is better than solution 2 only in terms of area but not in terms of wirelength. Hence both solutions belong to the global pareto-optimal front and are assigned a non-domination rank of zero.

If solutions 1 and 2 are removed from consideration, then the solutions numbered 3 and 4 are better than solutions 5 and 6 in terms of both objectives. But solution 3 is better than solution 4 only in terms of wirelength. Hence solutions 3 and 4 form the next local pareto-optimal front and are assigned a non-domination rank of 1. Similarly, solutions 5 and 6 are assigned a non-domination rank of 2.
Figure 3.1. Pareto-optimal solutions and non-domination levels in multi-objective optimization. Note: All values in generic units.

3.2 Floorplanning using Sequence Pair Representation

A Sequence Pair [16] is a representation of a floorplan using two sequences, \( \Gamma^+ \) and \( \Gamma^- \), of the \( n \) module indices. Without loss of generality, it can be assumed that all the indices of the \( n \) modules belong to the set, \( P = \{1, \ldots, n\} \).

3.2.1 Conversion from a Floorplan to a Sequence Pair

The Sequence Pair representation is obtained from its floorplan by a process called gridding. In gridding, each module’s diagonal is extended to the chip boundary using upward (downward) and left (right) extensions without any intersections with the loci of the other modules’ diagonals and the module boundaries. The first sequence (\( \Gamma^+ \)) is obtained by extending the SW-NE diagonals of all the modules and ordering the loci of these diagonals from left to right.
The second sequence (Γ-) is obtained by extending the NW-SE diagonals of all the modules and ordering these loci from left to right. Figure 3.2 illustrates this gridding process with an example.

Figure 3.2. Gridding process to obtain the Sequence Pair corresponding to a floorplan

### 3.2.2 Conversion from a Sequence Pair to a Floorplan

The geometric information of the modules can be obtained from the relative order of the modules in the two sequences. Consider two modules $i$ and $j$. Suppose $Γ^+ = <...i...j...>$ and $Γ^- = <...i...j...>$ then module $i$ is to the left of module $j$ in the floorplan corresponding to this Sequence Pair. Suppose $Γ^+ = <...j...i...>$ and $Γ^- = <...i...j...>$ then module $i$ is below module $j$. The geometric relation between any two modules in the floorplan can be obtained using these two rules. The actual co-ordinates of the modules can be obtained in $O(n^3)$ time by constructing horizontal and vertical constraint graphs based on the horizontal and vertical relations between the modules, where $n$ is the number of modules in the floorplan. Tang et al [17] proposed a faster $O(n \log \log n)$ algorithm to obtain module co-ordinates using longest common subsequences.

A Sequence Pair is characterized by the following two properties:
• Property 1 (Existence): All the given modules with module numbers from \( P = \{1, \ldots, n \} \) must be present in both the sequences.

If \( Q_1 = \{ x : x \in \Gamma^+ \} \) and \( Q_2 = \{ y : y \in \Gamma^- \} \) then

\[
Q_1 \cap P = P \text{ and } Q_2 \cap P = P.
\]

• Property 2 (Uniqueness): Each module must be present exactly once in both the sequences, i.e.

\[
\Gamma^+ (i) \neq \Gamma^+ (j) \quad \text{and} \quad \Gamma^- (i) \neq \Gamma^- (j), \forall i < j \leq n.
\]

A legal Sequence Pair \((\Gamma^+, \Gamma^-)\) is one that satisfies both existence and uniqueness properties.

### 3.3 Related Work

VLSI floorplanning is a well-studied problem for which a variety of optimization techniques have been applied including simulated annealing [17-22], mathematical programming [23, 24], and genetic algorithms [25-31].

Early floorplanners dealt with area optimization alone. But with the advent of the deep sub-micron regime, floorplanners shifted their focus to optimizing wirelength. But if wirelength is the only objective to be optimized, the resulting floorplan will have a lot of unused space. Hence, some floorplanners attempted to optimize both area and wirelength. Table 3.1 gives a brief summary of some of the recent floorplanning works in the literature that simultaneously optimize floorplan area and total wirelength. In the single normalized weighted sum (SNWS) approach to multi-objective optimization, simultaneous optimization of two objectives implies that the optimizer uses equal weights to multiply the normalized objectives before adding them together to obtain the single normalized weighted sum. It is to be noted that numerous floorplanning works exist in the literature that work either on area optimization alone or wirelength optimization. These
works will not be discussed here as the focus of this dissertation is on simultaneous optimization of area and wirelength.

Classical (outline-free) floorplanners based on Simulated Annealing [17-22] use the single normalized weighted sum approach to optimize the two objectives, namely area and total wirelength. These SA-based floorplanners differ only in the data structure, Sequence Pair [17,18], O-tree [19], or Transitive Closure Graph [20], used to represent the floorplans. Such floorplanners form a single scalar objective function using the two normalized objectives and the user defined weights for each objective as shown in Equation 3.3.1 below.

\[
f = W_1 \frac{\text{Area}}{\text{Area}_{\text{max}}} + W_2 \frac{\text{Wirelength}}{\text{Wirelength}_{\text{max}}}
\]

(3.3.1)

Mathematical programming based floorplanners use a wirelength estimate as the objective function to be minimized with a constraint on the floorplan area. Kim and Kim [23] proposed a linear programming approach to optimize area and wirelength simultaneously. Their approach uses a linear program with area constraints to optimize wirelength followed by a low temperature annealing process using the single normalized weighted sum approach to improve the solution quality. Sheqin et al [24] proposed a quadratic programming based floorplanner to optimize wirelength followed by a deterministic algorithm based on Less Flexibility First (LFF) principles to produce the final floorplan.

Genetic algorithms were first proposed for circuit placement by Cohoon and Paris [25]. The first genetic algorithm for floorplanning was proposed by Cohoon et al [26] and used Normalized Polish expressions to represent floorplans. Later, many other genetic floorplanners [27-31] were proposed that developed novel crossover techniques for different floorplan representation schemes. Esbensen [27] proposed a genetic macrocell placer for a binary tree-based representation. Valenzuela and Wang [31] also proposed a GA for floorplan area optimization that uses normalized polish expression representation. Hatta et al [28] proposed the first genetic
floorplanner based on the Sequence Pair representation that optimized floorplan area. Their genetic floorplanner used two new crossover operators, namely One-point Partially Matched Crossover (OPX) and Uniform Partially Matched Crossover (UPX). Hatta et al combined the well known Partially Matched Crossover [3] (meant for permutation-based chromosomes) with two binary crossover operators to formulate two new crossover operators (OPX and UPX) that could handle the Sequence Pair floorplan representation. OPX is a combination of the one point crossover and the Partially Matched Crossover. On the other hand, UPX is a combination of the uniform crossover and the Partially Matched Crossover operators.

Nakaya et al [29] later proposed a genetic floorplanner that also used the Sequence Pair floorplan representation. Their work used two novel crossover operators, Common Topology Preserving Crossover (CTPX) and Placement-based Partially Exchanging Crossover (PPEX) that were also specifically designed to work on Sequence Pairs. CTPX computes the longest common subsequences between the Sequence Pairs of the two parents and preserves it in the offspring. PPEX randomly selects a sub-floorplan from one parent and orders the modules making up the sub-floorplan according to their relative positions in the other parent. Nakaya et al report both area and wirelength results obtained by their genetic floorplanner on the MCNC benchmarks but do not report the results for simultaneous area and wirelength optimization.
Table 3.1. Brief summary of relevant floorplanning works that simultaneously optimize area and wirelength

<table>
<thead>
<tr>
<th>Optimization Technique</th>
<th>Work</th>
<th>Optimization Objectives</th>
<th>Floorplan Representation</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Area</td>
<td>Wire</td>
<td>Both</td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>Parquet [21]</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Guo et al [18]</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Lin and Chang [19]</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Lin and Chang [20]</td>
<td>x</td>
<td>x</td>
<td>possible</td>
</tr>
<tr>
<td>Linear Programming</td>
<td>Kim and Kim [23]</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Quadratic Programming</td>
<td>Sheqin et al [24]</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Genetic Algorithms</td>
<td>Chatterjee and Manikas [38]</td>
<td>x (+ temp)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Chatterjee, Manikas, and Markov [39]</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>This work</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
</tbody>
</table>
Modern VLSI floorplanning, as defined by Kahng [32], focuses on wirelength optimization within a fixed chip outline. With the chip complexity increasing with the improving integration technology, hierarchical design methods have become imperative. In a hierarchical design flow, floorplanning at the topmost-level might have a flexible chip outline. But the floorplans for the modules of the higher levels will fix the floorplan outline for the lower level sub-modules. This has led to an increased importance for the modern fixed outline floorplanning problem. It is to be noted that in modern floorplanning, wirelength is the primary objective while area is no longer an objective but rather a constraint. There have been many fixed outline floorplanners proposed in the literature [21, 33-35]. We will not review the fixed outline floorplanning works here as this work focuses on outline-free floorplanning. However, we will review a publicly available tool called Parquet [21, 22] as it is capable of both fixed outline and outline-free floorplanning.

Adya and Markov [21] proposed a novel simulated annealing based hybrid floorplanning tool called Parquet that is capable of both fixed and flexible outline floorplanning. Parquet uses the single normalized weighted sum approach during simulated annealing based optimization but also has some heuristic operators that drive the optimization engine towards solutions that obey the fixed outline constraint. They proposed the notion of slack of a module in Sequence Pair based floorplanning [21]. Generally, all the modules in a floorplan are compacted to the bottom-left corner of the space that they can occupy. To compute slack, all the modules are additionally compacted to the top-right corners in the floorplan. The slacks of a module $i$ are computed as:

$$X - slack(i) = xc_{topRight}(i) - xc_{botLeft}(i)$$

$$Y - slack(i) = yc_{topRight}(i) - yc_{botLeft}(i)$$

where $xc_{topRight}(i), yc_{topRight}(i)$ are the x- and y-coordinates of the lower left corner of module $i$ when the floorplan (i.e., all its modules) is compacted to the top-right corner, and
Adya and Markov [21] used the slack values of modules to estimate the amount of empty space around them. For instance, an X-slack value of zero for a module implies that it cannot be moved in the horizontal direction without altering the floorplan dimensions. A large Y-slack value for a module implies that there exists a lot of space for the module to move in the vertical direction and this empty space can be reduced by moving a small block that could fit in the empty space. Based on similar observations, Adya and Markov proposed novel slack-based heuristic operators to reduce the area and wirelength of the floorplan. Additionally, they used some of these operators to bias the simulated annealing engine to search for floorplans that obeyed the fixed outline constraint.

Multi-objective genetic algorithms and non-domination based solution ranking concepts have been successfully used for various problems belonging to different domains. In the VLSI domain, Dick and Jha [37] proposed a multi-objective hardware-software co-synthesis tool. Esbensen and Kuh [36] proposed a non-domination based solution ranking scheme for IC and MCM placement. The only works, to the best of the authors’ knowledge, which utilize a multi-objective genetic algorithm for the floorplanning problem, were proposed by Chatterjee and Manikas [38], and Chatterjee, Manikas, and Markov [39]. These works use the SPEA [14] multi-objective GA template. Chatterjee and Manikas [38] work on simultaneous optimization of chip area and maximum on-chip temperature and do not consider wirelength as an objective or constraint. Chatterjee, Manikas, and Markov [39] use the SPEA multi-objective GA template to optimize wirelength and temperature for both the fixed outline and classical floorplanning problems. Both these works do not propose any new crossover or mutation operators. They apply the SPEA multi-objective GA to the floorplanning problem and use traditional genetic operators.
Of the numerous floorplanning works in the literature, very few floorplanners work explicitly on simultaneous area and wirelength optimization. Among these floorplanners, all SA-based and most GA-based floorplanners use the single normalized weighted sum methodology to perform multi-objective optimization by assigning equal weights to the area and wirelength objectives. The proposed floorplanner works explicitly on simultaneous optimization of area and wirelength using the Elitist Non-dominated Sorting based Genetic Algorithm. It is to be noted that the proposed floorplanner can be easily extended to perform fixed outline floorplanning by incorporating a penalty function or by using a modified fitness assignment.

3.4 Proposed Multi-objective Genetic Floorplanner

The proposed floorplanner is an elitist non-dominated sorting based multi-objective genetic algorithm employing two novel crossover operators, a set of mutation operators, and a local optimization operator. The pseudo-code of the proposed genetic floorplanner is shown in Figure 3.3.

The following sections describe the various features of the proposed genetic floorplanner in detail:

- **Individual Representation** – Each individual in the population corresponds to a valid floorplan represented by a legal Sequence Pair. Sequence Pairs represent floorplans using two permutations ($\Gamma^+$, $\Gamma^-$) of the module indices. In this work, all modules are considered to be fixed modules in terms of their width and height but rotation of modules (by 0 or 90 degrees) are allowed. This rotation of modules is represented using a Boolean orientation vector denoted as $\theta$. The circuits in the GSRC macro-cell floorplanning benchmark suite do not contain pin information for the modules. In this case, other orientations and mirroring of modules will not affect the wirelength of the nets to which the module is connected. Thus the chromosomal encoding of an individual consists of two sequences namely, X sequence ($\Gamma^+$) and Y sequence ($\Gamma^-$), and the orientation vector ($\theta$).
Proposed Multi-Objective Genetic Floorplanner ($cx_rate, mut_rate, N\text{-}generations, popSize, tourneySize$)

{ 
1. $population \leftarrow$ Generate random Initial Population ($popSize$);
2. for $gen$ in $N\text{-}generations$ do 
   3. $EliteSet \leftarrow$ Non-Dominated Sort ($population$, $Area$, $Wire$);
   4. $MatePool \leftarrow$ Select Mating Pool ($population$);
   5. for $i$ in 1 to $cx\_rate$ do 
      6. ($P1$, $P2$) $\leftarrow$ Select Parents ($MatePool$, $tourneySize$);
      7. ($Off1$, $Off2$) $\leftarrow$ Crossover ($P1$, $P2$);
      8. }
   9. for $i$ in 1 to $mut\_rate$ do 
      10. $mutIndex \leftarrow$ Select Non-Elite Individual ($population$);
      11. Mutate ($mutIndex$);
      12. Update Population ($population$, $mutIndex$);
      13. }
   14. Perform Local Optimization ($offspring$);
   15. Update Population ($population$, $offspring$);
   16. }
17. return (Elite Set ($population$));
}

Figure 3.3. Pseudo-code of the proposed hybrid elitist non-dominated sorting GA-based floorplanner

- **Initial Population** – The proposed genetic floorplanner starts with an initial population of randomly generated Sequence Pairs and orientation vectors (line1 of procedure in Figure 3.3). The size of the population used by the proposed genetic floorplanner is not fixed. Empirical studies (to be discussed in Section 3.6) with a training set of benchmark circuits from both the MCNC and GSRC benchmark suites were conducted to find the values for the GA parameters that give the best results. Based on the results of the empirical studies, Equation 3.3.4 has been derived to determine the GA population size based on the problem size, i.e., number of modules ($n$). This equation was then applied to size the
population by the proposed floorplanner for all the benchmarks. To the best of the author’s knowledge, this is the first genetic floorplanner that varies the population size according to the number of modules \( (n) \) present in the benchmark circuit.

\[
\text{population size} = 10^n
\]

(3.3.4)

- **Non-dominated Sorting of the Population** – The proposed floorplanner sorts the entire population into various non-domination levels (described in the Section 3.1) in terms of area and total wirelength at the beginning of every generation (line 3 of procedure in Figure 3.3). All the individuals in the current population are assigned a non-domination rank starting from zero. The fittest individuals have the least non-domination ranks.

- **Elite Individuals** – The number of elite individuals in the current population of the proposed GA varies with each generation. All the individuals with the lowest non-domination rank (zero) form the current set of elite individuals. These individuals are not subject to mutation. Thus, individuals containing genetic information contributing to reduced area or wirelength will be preserved for the future generations.

- **Mating Pool Selection** – In every generation the proposed floorplanner selects a pool of individuals to use as parents for the crossover operations in that generation (line 4 of procedure in Figure 3.3. The size of this pool is set to half the population size as recommended in [5]. If the number of elite individuals is greater than this size, then the elite individuals with the largest crowding distance [10] [15] are picked to form the mating pool to ensure that a diverse set of individuals are maintained in the population.

- **Crowding Distance** – Crowding Distance \( (d_i) \) of a solution \( s_i \) is defined as the distance between the solutions \( s_{i+1} \) and \( s_{i-1} \) belonging to the same non-domination front as the solution \( s_i \) and are immediate neighbors to the solution \( s_i \). Crowding Distance can be measured in either the solution encoding space or the objective function space. The objective space in this work is two-dimensional as floorplan area and total wirelength are
the objectives considered. Each floorplan solution can be considered as a point in this two-dimensional objective space as shown in Figure 3.4. In the proposed GA, the crowding distance of a solution is measured in the objective space. Since the objective space is two-dimensional, the procedure in Figure 3.5 can be used to assign crowding distances to solutions. In the proposed GA, all the individuals in the population are assigned a crowding distance in addition to the non-domination ranks. The crowding distance of an individual is a measure of the proximity of neighboring solutions in the current population. In multi-objective genetic algorithms, premature convergence of a non-dominated front to a small section of the actual front must be prevented. This can be achieved by preserving solutions that do not have close neighboring solutions in the current population.

Figure 3.4. Computing crowding distance of a solution in the objective space. Note: All values are in generic units.
Figure 3.5. Procedure to compute the crowding distances of solutions belonging to a non-domination front

- **Parent Selection** – The proposed GA uses crowded tournament selection \[10\] \[15\] to select the two parents for crossover from the mating pool of individuals. In tournament selection, a group of parent candidates are selected randomly from the mating pool. A tournament is played between these candidate individuals to determine the fittest two individuals of the group.

In crowded tournament selection, individuals are first compared according to their non-domination ranks. The individual with the smaller non-domination rank is considered the winner. If the individuals being compared have the same non-domination rank, then the individual with the largest crowding distance is declared the winner. The proposed floorplanner uses 1% of the population’s individuals to form the parent candidate pool for crowded tournament selection. Since the population size is set to \(10^n\), the size of the parent candidate pool is \(n/10\).

- **Crossover** – The crossover operator (lines 5-8 of the procedure in Figure 3.3) is used by genetic algorithms to combine good traits from the parents to form highly fit offspring. A good crossover operator must also ensure that the offspring does not closely resemble either parent. These properties ensure that the crossover operator explores different promising areas in the solution space. The crossover rate (\(cx\_rate\)) limits the amount of

```
Procedure Compute_CrowingDistance (front, frontsize)
1. for i in 0 to frontsize-1 do
   a. d[i] = 0;
2. front = Sort (front, frontsize, area);
3. d[0] = d[frontsize-1] = INF;
4. for j in 1 to frontsize-2 do
   a. d[j] = (area[j+1] – area[j-1]) + (wire[j-1] – wire[j+1]);
5. return;
```
crossover performed in any generation. This value was set to 1.0 after empirical studies that will be described in Section 3.6. Two new crossover operators are proposed to work on Sequence Pairs, which are described in detail in Section 3.5.

- **Mutation Operators** – The mutation operator (lines 9-13 of the procedure in Figure 3.3) is used by genetic algorithms to produce diversity in the population. It also helps in avoiding a quick convergence to local optima. In elitist genetic algorithms, mutation is not applied to the elite individuals. Three mutation operators are used in the proposed genetic floorplanner. A non-elite individual is randomly chosen from the population and one of the three mutation operators is applied to the individual. Each of the mutation operators has an equal selection probability. The mutation rate (mut_rate) limits the number of mutation operations performed in any generation. A mutation rate of 0.1 gave the best results in empirical studies and was maintained for all the experiments. The three mutation operators used in the proposed floorplanner are described below.

- **Mutation Operator 1** (Random module position exchange in both Γ+ and Γ- sequences) – This mutation operator picks 2 random modules and exchanges their positions in both Γ+ and Γ-sequences, of the chosen individual. If the individual chosen for mutation is a legal Sequence Pair, then this mutation operator does not introduce any new modules in either sequence. Also, this mutation operator neither duplicates nor erases any module in either sequence. Hence, the Sequence Pair of the mutated individual will remain legal.

- **Mutation Operator 2** (Random module position exchange in Γ+ sequence alone) – This mutation operator picks 2 random modules and exchanges their positions only in the first sequence, Γ+, of the chosen individual. If the individual chosen for mutation is a legal Sequence Pair then this mutation operator does not introduce any new modules in the first sequence, and neither duplicates nor erases any module in the first
sequence. Also, the second sequence is not changed at all. Hence the Sequence Pair of the mutated individual will remain legal.

- **Mutation Operator 3** (Random module orientation change) – This mutation operator picks a random module, $b_i$, and changes its orientation ($\theta_i$) by exchanging the width and the height of the chosen module. This operator does not change the Sequence Pair at all and hence will maintain the legality of the Sequence Pair of the mutated individual.

- **Local Optimization** – The proposed genetic floorplanner uses a heuristic local optimization technique to find better solutions that are located near the generated offspring. Genetic algorithms are not very efficient in local neighborhood search [4]. Hybridization of genetic algorithms with local search operators will result in faster convergence to better solutions. The proposed genetic floorplanner uses a slack based local optimization operator, based on a strategy proposed in [21]. Although different strategies for slack-based moves were discussed in [21], no procedures were given for the implementation of the slack-based moves used in [21]. In this work, a single slack-based move is utilized which has been implemented using the procedure described in Figure 3.6. Local optimization using Procedure Slack_Move1 is a greedy procedure that accepts the modified individual if there is an improvement in either area or wirelength. If there is no improvement in either objective, the original individual from the population ($currInd$ in the procedure) is returned.

The local optimization move, Slack_Move1, uses the slack measures defined in Equation 3.3.2 and Equation 3.3.3 as whitespace estimates around the modules. It identifies the smallest module with zero whitespace surrounding it using lines 3 and 4 of the procedure. It then identifies a large module surrounded by a lot of whitespace in line 5 of the procedure. Using lines 6-8, the procedure moves the small module adjacent to the
larger module in either the horizontal or vertical direction based on the whitespace along those directions.

Procedure Slack_Move1 (currInd, newInd)
1. Copy (currInd, newInd);
2. Evaluate Slacks (newInd);
3. findLeastSlackModules (newInd, pList);
4. \( p = \text{findSmallestAreaModule (pList)} \);
5. \( q = \text{findLargestSlackModule (newInd)} \);
6. If (X-Slack(q) > Y-Slack(q)) then
   i. Move module \( p \) next to module \( q \) and arrange them in a horizontal fashion.
7. Else if (X-Slack(q) < Y-Slack(q)) then
   i. Move module \( p \) next to module \( q \) and arrange them in a vertical fashion.
8. Else
   i. Move module \( p \) next to module \( q \) and maintain their existing geometric relations.
9. Compute Fitness (newInd);
10. If (area(newInd) >= area(currInd) OR (wirelength(newInd) >= wirelength(currInd))
    i. copy(currInd, newInd);
11. return;

Figure 3.6. Procedure for Slack_Move1 used for local optimization by the proposed floorplanner

- Population Update – The new population for the next generation is formed by choosing the required number of individuals from the combined pool of the current population and the newly formed offspring population. The combined pool is sorted into non-dominated fronts and all the individuals belonging to a particular non-domination front are copied into the new generation starting from the non-domination front with rank zero. If the addition of all the individuals in a certain non-domination front results in violation of the population size, then the required number of individuals are chosen using the crowded
selection operator so that a uniformly distributed front, with respect to the crowding
distance, is obtained.

3.5 Proposed Crossover Operators

The proposed genetic floorplanner uses two novel crossover operators - namely Modified
Two-Point Order-based Crossover Operator (MTOX) and Heuristic One-Point Order-based
Crossover Operator (HOOX). The MTOX operator is an unbiased operator that attempts to search
the entire solution space for a good solution by randomly combining segments from the parents to
form the offspring. The Heuristic One-point Order Crossover operator (HOOX) tries to bias the
search towards promising regions of the solution space by promoting the transfer of good sub-
floorplans from the parents to the offspring.

3.5.1 Modified Two-Point Order-based Crossover (MTOX) Operator

The MTOX operator is a novel crossover operator proposed specifically to work on
Sequence Pairs. This operator is a combination of two classical crossover operators, namely the
two-point crossover operator and the order crossover operator.

The original two point crossover operator [3,4] was proposed for use with individuals that
were encoded as binary strings, as is illustrated in Figure 2.3. If this traditional method is used
with Sequence Pairs it will result in invalid $\Gamma^+$ and $\Gamma^-$ sequences due to duplication (violation of
uniqueness property) and deletion of modules (violation of existence property) as shown in Figure
3.7. These violations must be removed to obtain a Sequence Pair that represents a valid floorplan.
To eliminate the occurrence of such violations in the offspring's Sequence Pair, the traditional two point crossover operator is combined with the order crossover operator to yield the Modified Two-point Order Crossover (MTOX). The original order crossover operator [3] was proposed to work on individuals encoded as a single permutation as shown in Figure 3.8. To form the first offspring (Offspring 1 in Figure 3.8), the order crossover operator cuts the parent chromosomes at two points. For the genes present in between the two cuts of the second parent, the operator then identifies their positions in the first parent and fills these positions with holes (denoted by H in Figure 3.8). The holes are then slid to the spaces between the two cuts in a wrap-around fashion. Finally, the holes are filled with the genes in the order they are present in the second parent. Offspring 2 is produced by a similar process after exchanging the two parents. In this manner, order crossover preserves the relative positions of the genes from both the parents in the offspring.
The MTOX operator is not a simple extension of the two-point crossover to handle the two permutations of the Sequence Pair. The MTOX operator couples the crossover of the $\Gamma$-sequence with the crossover of the $\Gamma^+$ sequence to ensure that the relative positions of the modules contributed by the same parent are maintained in the offspring. The orientation of a module is contributed by the parent that dictates the position of the module. This ensures that a good configuration within a parent is preserved in the offspring. The MTOX crossover operator accepts two parents $P_1$ and $P_2$ as input and generates two offspring, $Off_1$ and $Off_2$. The MTOX crossover operation is formally described in the procedure below:

Procedure MTOX ($P_1$, $P_2$, $Off_1$, $Off_2$)

Step 1. Generate 2 random cut-points, $c_1$ and $c_2$, on $\Gamma^+$ sequence of the first parent individual, $P_1$.

Let $x_1^{P_1}$, $x_2^{P_1}$, and $x_3^{P_1}$ be the three segments obtained from the two-point cut of $\Gamma^+$ sequence of $P_1$.

Step 2. Find the order of the modules in segment $x_1^{P_1}$ in the first parent's ($P_1$) $\Gamma$- sequence.
Generate segment $y_1^{P1}$ using this ordering of modules. Similarly, generate segments $y_2^{P1}$ and $y_3^{P1}$ using P1’s $\Gamma^-$ sequence ordering of the modules in the segments $x_2^{P1}$ and $x_3^{P1}$.

Step 3. Find the order of the modules in segment $x_1^{P1}$ in the second parent's (P2) $\Gamma^+$ sequence.

Generate segment $x_1^{P2}$ using this ordering of modules. Similarly, generate segments $x_2^{P2}$ and $x_3^{P2}$ using P2’s $\Gamma^+$ sequence ordering of the modules in the segments $x_2^{P1}$ and $x_3^{P1}$.

Step 4. Find the order of the modules in segment $x_1^{P1}$ in the second parent's (P2) $\Gamma^-$ sequence.

Generate segment $y_1^{P2}$ using this ordering of modules. Similarly, generate segments $y_2^{P2}$ and $y_3^{P2}$ using P2’s $\Gamma^-$ sequence ordering of the modules in the segments $x_2^{P1}$ and $x_3^{P1}$.

Step 5. The concatenation of the sequences $x_1^{P1}$, $x_2^{P2}$, and $x_3^{P1}$ forms the $\Gamma^+$ sequence of the first offspring (Off1).

Step 6. The concatenation of the sequences $y_1^{P1}$, $y_2^{P2}$, and $y_3^{P1}$ forms the $\Gamma^-$ sequence of the first offspring (Off1).

Step 7. Module orientations in the offspring are copied over from the respective parent that contributes the position of the module.

Step 8. Steps 1-7 are repeated after exchanging the two parents to obtain the second offspring, Off2.

Figure 3.9 illustrates the MTOX operator with an example. It can be formally proven that the offspring produced by the MTOX procedure above always produce legal offspring as shown in Theorem 3.1.
Figure 3.9. Generation of the first offspring using the MTOX operator
Theorem 3.1. Given legal sequence pairs as the parents, the MTOX operator always produces legal sequence pairs as the offspring.

Proof:

Let $X^{p1}$ and $Y^{p1}$ denote the $\Gamma+$ and $\Gamma-$ sequences of the first parent. Let $X^{p2}$ and $Y^{p2}$ denote the $\Gamma+$ and $\Gamma-$ sequences of the second parent. Let $M = \{1, \ldots, n\}$ and without loss of generality, let us assume that there is a bijection between the module names and the set $M$.

Let $S = \{b: b \in X^{p1}\}$ and $T = \{b: b \in Y^{p1}\}$. Similarly let $Q = \{b: b \in X^{p2}\}$ and $R = \{b: b \in Y^{p2}\}$. Since both the parents correspond to legal floorplans, we have

$$S = T = Q = R = M \quad (3.5.1)$$

Step 1 of Procedure MTOX generates three segments $x_{1}^{p1}$, $x_{2}^{p1}$, and $x_{3}^{p1}$ from $X^{p1}$ using the two random cutpoints. Let $s_{1} = \{b: b \in x_{1}^{p1}\}$, $s_{2} = \{b: b \in x_{2}^{p1}\}$ and $s_{3} = \{b: b \in x_{3}^{p1}\}$. By construction and Equation 3.5.1,

$$s_{1} \cup s_{2} \cup s_{3} = s = M$$
$$s_{1} \cap s_{2} \cap s_{3} = \emptyset \quad (3.5.2)$$

Let sequence $s$ be an arbitrary subsequence of $\{1, \ldots, n\}$. We define $\Gamma+(s)$ [or $\Gamma-(s)$] as the subsequence of $\Gamma+$ (or $\Gamma-$) that contains the modules in $s$ in the order they occur in the sequence $\Gamma+$ (or $\Gamma-$). For example, assume $X^{p1} = \{5, 4, 1, 3, 2\}$ and $s = \{1, 2, 5\}$; then $X^{p1}(s) = \{5, 1, 2\}$.

Step 2 of Procedure MTOX generates three segments $y_{1}^{p1} = Y^{p1}(s_{1})$, $y_{2}^{p1} = Y^{p1}(s_{2})$, and $y_{3}^{p1} = Y^{p1}(s_{3})$. Let $t_{1} = \{b: b \in y_{1}^{p1}\}$, $t_{2} = \{b: b \in y_{2}^{p1}\}$ and $t_{3} = \{b: b \in y_{3}^{p1}\}$. By construction and Equation 3.5.1 we get,

$$t_{1} = s_{1}, \quad t_{2} = s_{2}, \quad t_{3} = s_{3} \quad (3.5.3)$$
$$t_{1} \cup t_{2} \cup t_{3} = t = M \quad (3.5.4)$$
$$t_{1} \cap t_{2} \cap t_{3} = \emptyset$$

Step 3 generates three segments $x_{1}^{p2} = X^{p2}(s_{1})$, $x_{2}^{p2} = X^{p2}(s_{2})$, and $x_{3}^{p2} = X^{p2}(s_{3})$. Let $q_{1} = \{b: b \in x_{1}^{p2}\}$, $q_{2} = \{b: b \in x_{2}^{p2}\}$ and $q_{3} = \{b: b \in x_{3}^{p2}\}$. By construction and Equation 3.5.1,
\[ q_1 = s_1, \quad q_2 = s_2, \quad q_3 = s_3 \]  
\[ q_1 \cup q_2 \cup q_3 = Q = M \]  
\[ q_1 \cap q_2 \cap q_3 = \emptyset \]  
(3.5.5)  
(3.5.6)

Step 4 generates three segments \( y_1^{p_2} = Y^{p_2}(s_1) \), \( y_2^{p_2} = Y^{p_2}(s_2) \), and \( y_3^{p_2} = Y^{p_2}(s_3) \). Let \( r_1 = \{ b : b \in y_1^{p_2} \} \), \( r_2 = \{ b : b \in y_2^{p_2} \} \) and \( r_3 = \{ b : b \in y_3^{p_2} \} \). By construction and Equation 3.5.1,

\[ r_1 = s_1, \quad r_2 = s_2, \quad r_3 = s_3 \]  
\[ r_1 \cup r_2 \cup r_3 = R = M \]  
\[ r_1 \cap r_2 \cap r_3 = \emptyset \]  
(3.5.7)  
(3.5.8)

Step 5 forms the \( \Gamma^+ \) sequence (\( X^{\text{Off}} \)) of the offspring using segments \( x_1^{p_1}, x_2^{p_2} \) and \( x_3^{p_1} \). Let \( F_1 = \{ b : b \in X^{\text{Off}} \} \). By construction, \( F_1 = s_1 \cup q_2 \cup s_3 \). From Equation 3.5.5, \( q_2 = s_2 \). Hence,

\[ F_1 = s_1 \cup s_2 \cup s_3 = M \]  
\[ s_1 \cap q_2 \cap s_3 = s_1 \cap s_2 \cap s_3 = \emptyset \]

Thus the \( \Gamma^+ \) sequence of the offspring is legal.

Step 6 forms the \( \Gamma^- \) sequence (\( Y^{\text{Off}} \)) of the offspring using the segments \( y_1^{p_1}, y_2^{p_2} \) and \( y_3^{p_1} \). Let \( F_2 = \{ b : b \in Y^{\text{Off}} \} \). By construction, \( F_2 = t_1 \cup r_2 \cup t_3 \). From Equation 3.5.3 and Equation 3.5.7, we get \( r_2 = t_2 \). This yields

\[ F_2 = t_1 \cup t_2 \cup t_3 = M \]  
\[ t_1 \cap t_2 \cap t_3 = t_1 \cap t_2 \cap t_3 = \emptyset \]

Thus, the \( \Gamma^- \) sequence of the offspring is also legal.

Hence, the offspring floorplan represented by the \( \Gamma^+ \) sequence and \( \Gamma^- \) sequence formed using the proposed MTOX crossover operator is always a legal floorplan

3.5.2 Heuristic One-Point Order-based Crossover (HOOX) Operator

The HOOX operator is a heuristic crossover operator, proposed specifically for Sequence Pairs, that identifies good sub-floorplans in the parents and preserves them in the offspring. The
HOOX operator uses the traditional One-point Crossover operator [3] to partition both the parents into two sub-floorplans as shown in Figure 3.10. The order crossover operator is used to eliminate any violations in the Sequence Pairs for the resulting sub-floorplans. The two sub-floorplans with the better area usage are then combined to form the offspring as shown in Figure 3.11. The HOOX crossover operation is described in the procedure below.

Procedure HOOX (P1, P2, Off1)

Step 1. Split the $\Gamma^+$ sequence of the first parent, P1. Let $x_1^{P1}$ and $x_2^{P1}$ be the two segments obtained from the one-point cut of $\Gamma^+$ sequence of P1.

Step 2. Find the order of the modules in segment $x_1^{P1}$ in the first parent's ($P1$) $\Gamma^-$ sequence. Generate segment $y_1^{P1}$ using this ordering of modules. Similarly, generate segment $y_2^{P1}$ using P1’s $\Gamma^-$ sequence ordering of the modules in the segment $x_2^{P1}$. The Sequence Pairs $(x_1^{P1}, y_1^{P1})$ and $(x_2^{P1}, y_2^{P1})$ correspond to two sub-floorplans, FP11 and FP21.

Step 3. Find the order of the modules in segment $x_1^{P1}$ in the second parent's ($P2$) $\Gamma^+$ sequence. Generate segment $x_1^{P2}$ using this ordering of modules. Similarly, generate segment $x_2^{P2}$ using P2’s $\Gamma^+$ sequence ordering of the modules in the segment $x_2^{P1}$. Similarly, generate segments $y_1^{P2}$ and $y_2^{P2}$ using P2’s $\Gamma^-$ sequence ordering of the modules in the segments $x_1^{P1}$ and $x_2^{P1}$. The Sequence Pairs $(x_1^{P2}, y_1^{P2})$ and $(x_2^{P2}, y_2^{P2})$ correspond to two sub-floorplans, FP12 and FP22.

Step 4. Sub-floorplans FP11 and FP12 contain the same modules in them and form two alternatives for building a sub-floorplan using the modules in segment $x_1^{P1}$. Sub-floorplans FP21 and FP22 form two alternatives for building a sub-floorplan using the modules in segment $x_2^{P1}$. The sub-floorplan alternatives with the better area usage are picked to form the offspring floorplan.

Step 5. Four different offspring configurations are possible using the two sub-floorplan alternatives. The best configuration is chosen as the final offspring.
Figure 3.10. Generation of sub-floorplans using single cut-point in the HOOX operator (steps 1-3 of Procedure HOOX)

The module orientations in the parents are preserved in the generated sub-floorplans and are copied over to the offspring. To speed up the area computations of the offspring configurations, each sub-floorplan is regarded as a super-module. The offspring can now be considered to be made of just two super-modules. It is to be noted that in step 5 of Procedure HOOX, no modules are added or deleted. Only the positions of the individual modules from the selected segments are determined. It can be formally proven that the offspring produced by the proposed HOOX operator is always legal as shown in Theorem 3.2.
Theorem 3.2. Given legal sequence pairs as the parents, the HOOX operator always produces legal sequence pairs as the offspring.

Proof:

Let $X^p_1$ and $Y^p_1$ denote the $\Gamma^+$ and $\Gamma^-$ sequences of the first parent. Let $X^p_2$ and $Y^p_2$ denote the $\Gamma^+$ and $\Gamma^-$ sequences of the second parent. Let $M = \{1, \ldots, n\}$. Without loss of generality, we can assume that there exists a bijection between the module names and the set $M$.

Let $S = \{b : b \in X^p_1\}$ and $T = \{b : b \in Y^p_1\}$. Similarly let $Q = \{b : b \in X^p_2\}$ and $R = \{b : b \in Y^p_2\}$. Since both parents correspond to legal floorplans, we have

$$S = T = Q = R = M \quad (3.5.9)$$

Step 1 of Procedure HOOX generates two segments $x_1^p$ and $x_2^p$ from $X^p_1$ using the single random cutpoint.
Let \( s_1 = \{ b : b \in x_1^{p1} \} \) and \( s_2 = \{ b : b \in x_2^{p1} \} \). By construction and Equation 3.5.9,

\[
\begin{align*}
    s_1 \cup s_2 &= S = M \\
    s_1 \cap s_2 &= \emptyset
\end{align*}
\] (3.5.10)

Step 2 of Procedure HOOX generates two segments \( y_1^{p1} = Y^{p1}(s_1) \) [the notation used here is the same as defined in the proof of Theorem 3.1] and \( y_2^{p1} = Y^{p1}(s_2) \). Let \( t_1 = \{ b : b \in y_1^{p1} \} \) and \( t_2 = \{ b : b \in y_2^{p1} \} \). By construction, and Equation 3.5.9,

\[
\begin{align*}
    t_1 &= s_1, \ t_2 = s_2 \\
    t_1 \cup t_2 &= T = M \\
    t_1 \cap t_2 &= \emptyset
\end{align*}
\] (3.5.11)

(3.5.12)

Step 3 of Procedure HOOX first generates two segments \( x_1^{p2} = X^{p2}(s_1) \) and \( x_2^{p2} = X^{p2}(s_2) \).

Let \( q_1 = \{ b : b \in x_1^{p2} \} \) and \( q_2 = \{ b : b \in x_2^{p2} \} \). By construction and Equation 3.5.9,

\[
\begin{align*}
    q_1 &= s_1, \ q_2 = s_2 \\
    q_1 \cup q_2 &= Q = M \\
    q_1 \cap q_2 &= \emptyset
\end{align*}
\] (3.5.13)

(3.5.14)

Step 3 of Procedure HOOX also generates two more segments \( y_1^{p2} = Y^{p2}(s_1) \) and \( y_2^{p2} = Y^{p2}(s_2) \). Let \( r_1 = \{ b : b \in y_1^{p2} \} \) and \( r_2 = \{ b : b \in y_2^{p2} \} \). By construction and Equation 3.5.9,

\[
\begin{align*}
    r_1 &= s_1, \ r_2 = s_2 \\
    r_1 \cup r_2 &= R = M \\
    r_1 \cap r_2 &= \emptyset
\end{align*}
\] (3.5.15)

(3.5.16)

Step 4 of Procedure HOOX picks either \( s_1 \) and \( t_1 \) or \( q_1 \) and \( r_1 \) for the first sub-floorplan of the offspring. Step 4 also picks either \( s_2 \) and \( t_2 \) or \( q_2 \) and \( r_2 \) for the second sub-floorplan of the offspring. Four different offspring configurations are possible. In all of the cases, let \( F_1 \) denote the set of modules used to form the offspring’s \( \Gamma^+ \) sequence, i.e. \( F_1 = \{ b : b \in X^{\text{Off}1} \} \) and let \( F_2 \) denote the set of modules used to form the offspring’s \( \Gamma^- \) sequence, i.e. \( F_2 = \{ b : b \in Y^{\text{Off}1} \} \).
Case 1: \(s_1, t_1, s_2\) and \(t_2\) are chosen to form the offspring

The \(\Gamma^+\) sequence of the offspring is formed using the modules in \(s_1\) and \(s_2\). Hence, 
\[ F_1 = s_1 \cup s_2. \] From Equation 3.5.10, we have \(F_1 = M\) and \(s_1 \cap s_2 = \emptyset\). Hence, \(\Gamma^+\) sequence of the offspring is legal.

The \(\Gamma^-\) sequence of the offspring is formed using the modules in \(t_1\) and \(t_2\). Hence, 
\[ F_2 = t_1 \cup t_2. \] From Equation 3.5.12, we have \(F_2 = M\) and \(t_1 \cap t_2 = \emptyset\). Hence, \(\Gamma^-\) sequence of the offspring is legal.

Case 2: \(s_1, t_1, q_2\) and \(r_2\) are chosen to form the offspring

The \(\Gamma^+\) sequence of the offspring is formed using modules in \(s_1\) and \(q_2\). Hence, 
\[ F_1 = s_1 \cup q_2. \] From Equations 3.5.10 and 3.5.13, we have \(F_1 = s_1 \cup q_2 = s_1 \cup s_2 = M\) and 
\[ s_1 \cap q_2 = s_1 \cap s_2 = \emptyset. \] Hence, \(\Gamma^+\) sequence of offspring is legal. The \(\Gamma^-\) sequence of the offspring is also formed using modules in \(t_1\) and \(r_2\). Hence, 
\[ F_2 = t_1 \cup r_2. \] From Equations 3.5.11, 3.5.12, and 3.5.13, 
\[ F_2 = t_1 \cup r_2 = t_1 \cup t_2 = M \] and 
\[ t_1 \cap r_2 = t_1 \cap t_2 = \emptyset. \] Thus, \(\Gamma^-\) sequence of offspring is legal.

Case 3: \(q_1, r_1, q_2\) and \(r_2\) are chosen to form the offspring

The \(\Gamma^+\) sequence of the offspring is formed using modules in \(q_1\) and \(q_2\). Hence, 
\[ F_1 = q_1 \cup q_2. \] From Equation 3.5.14, we have \(F_1 = M\) and \(q_1 \cap q_2 = \emptyset\). Thus, \(\Gamma^+\) sequence of the offspring is legal. The \(\Gamma^-\) sequence of the offspring is formed using the modules in \(r_1\) and \(r_2\). Hence, 
\[ F_2 = r_1 \cup r_2. \] From Equation 3.5.16, we have \(F_2 = M\) and \(r_1 \cap r_2 = \emptyset\). Thus, \(\Gamma^-\) sequence of the offspring is also legal.

Case 4: \(q_1, r_1, s_2\) and \(t_2\) are chosen to form the offspring

The \(\Gamma^+\) sequence of the offspring is formed using the modules in \(q_1\) and \(s_2\). Hence, 
\[ F_1 = q_1 \cup s_2. \] From Equations 3.5.10 and 3.5.13, we have \(F_1 = s_1 \cup s_2 = M\) and 
\[ q_1 \cap s_2 = s_1 \cap s_2 = \emptyset. \] Thus, \(\Gamma^+\) sequence of the offspring is legal.
The $\Gamma$-sequence of the offspring is formed using the modules in $r_1$ and $t_2$. Hence, $F_2 = r_1 \cup t_2$. From Equations 3.5.11 and 3.5.15 we have $F_2 = r_1 \cup t_2 = t_1 \cup t_2 = M$ and $r_1 \cap t_2 = t_1 \cap t_2 = \emptyset$. Thus, the $\Gamma$-sequence of the offspring is also legal.

Since the above four cases are the only ways in which the HOOX operator produces an offspring, it is proved that the proposed HOOX operator always produces legal offspring.

### 3.6 Empirical Determination of GA Parameter Settings

The values for all the GA parameters were determined from empirical studies conducted on a training set of circuits selected from both the MCNC and GSRC benchmark suites. The hp and ami33 circuits were used as training circuits from the MCNC suite while the n100 and n300 circuits were used as training circuits from the GSRC benchmark suite.

Table 3.2. Best area and total wirelength results obtained by the proposed genetic floorplanner for the n300 benchmark for various crossover and mutation rates (A – Area in sq.units, W – Total Wirelength in units)

<table>
<thead>
<tr>
<th>MUTN_RATE</th>
<th>CX_RATE</th>
<th>0.5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>W</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>0.01</td>
<td>328755</td>
<td>598997</td>
</tr>
<tr>
<td></td>
<td>0.05</td>
<td>325511</td>
<td>587297</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>319061</td>
<td>592562</td>
</tr>
</tbody>
</table>

Generally speaking, crossover rates are set high (in the range of $[0.5, 1.0]$) in genetic algorithms to allow the inheritance of good genetic information from parents to offspring while mutation rates are kept low (in the range of $[0.001, 0.1]$) to prevent the loss of good genetic information from the population [3, 4]. In the proposed genetic floorplanner, two crossover rates (0.5 and 1.0) and three mutation rates (0.01, 0.05, and 0.1) were used for the experimental studies.
Table 3.2 lists the results obtained by the proposed genetic floorplanner on the n300 GSRC benchmark for the different settings of the crossover and mutation rates. For all the training circuits, the best results were obtained when the crossover rate was set to 1.0 and the mutation rate was set to 0.1. The genetic floorplanner also exhibited good convergence for these values. Figure 3.12 through Figure 3.17 show the convergence plots of the proposed genetic floorplanner for various settings of the crossover and mutation rates.

Figure 3.12. Convergence plot of wirelength for n300 benchmark with cxRate=0.5, mutRate=0.1. Note: All values are in generic units.
Figure 3.13. Convergence plot of area for n300 benchmark with cxRate=0.5, mutRate=0.1. Note: All values are in generic units.

Figure 3.14. Convergence plot of wirelength for n300 benchmark with cxRate=1.0, mutRate=0.05. Note: All values are in generic units.
Figure 3.15. Convergence plot of area for n300 benchmark with cxRate=1.0, mutRate=0.05. Note: All values are in generic units.

Figure 3.16. Convergence plot of wirelength for n300 benchmark with cxRate=1.0, mutRate=0.1. Note: All values are in generic units.
3.7 Experimental Results

The proposed genetic floorplanner was implemented using C++/STL and compiled with g++ version 3.4.6 (using the -O2 flag). For all the experiments, the proposed genetic floorplanner was run with a crossover rate of 1.0 and mutation rate of 0.1. These GA parameter values were empirically determined as discussed in Section 3.6. All the experiments were run on a Linux machine with a 3.2GHz Intel Pentium 4 processor and 2GB RAM.
Table 3.3. Area and wirelength comparisons with the AdaptGA, QP-LFF, and Parquet floorplanners 
(A = Area in mm$^2$, W = Total Wirelength in mm, %S = Percentage Savings, “-” = not reported)

<table>
<thead>
<tr>
<th></th>
<th>Proposed Floorplanner</th>
<th>AdaptGA Floorplanner</th>
<th>QP-LFF Floorplanner</th>
<th>SA-based (Parquet) Floorplanner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>W</td>
<td>A</td>
<td>%S</td>
</tr>
<tr>
<td>apte</td>
<td>48.48</td>
<td>319.81</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>hp</td>
<td>10.052</td>
<td>146.94</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>xerox</td>
<td>20.56</td>
<td>424.48</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ami33</td>
<td>1.20</td>
<td>31.33</td>
<td>1.22</td>
<td>1.64</td>
</tr>
<tr>
<td>ami49</td>
<td>37.81</td>
<td>677.9</td>
<td>37.16</td>
<td>-1.75</td>
</tr>
<tr>
<td>Average Savings</td>
<td>-0.05</td>
<td>25.3</td>
<td>-2.63</td>
<td>26.9</td>
</tr>
</tbody>
</table>

55
Floorplan area is computed using the longest common subsequence method [16] proposed for floorplans represented by Sequence Pairs. Half-perimeter wirelengths (HPWL) are computed for all the nets to estimate the total wiring required for the floorplans. The positions of the module pins are available for the MCNC benchmarks. Hence, pin-to-pin half-perimeter wirelengths are computed for the MCNC benchmarks. Moreover, the positions of the pads on the chip’s periphery are also specified for the MCNC benchmarks. The pads located at the chip’s corners will define an outline for the chip. If the aspect ratio of the packing produced by the proposed genetic floorplanner for the benchmark is different from the chip’s outline, the pad locations are scaled along the chip boundary to obtain their new positions. This method has been previously used by many researchers as reported in [40]. The module pin positions are not specified for the GSRC benchmarks. Hence the module centers are used to measure HPWL in this case. The area and wirelength results shown for the proposed genetic floorplanner belong to a single valid floorplan chosen from the final pareto-optimal set. These results belong to the best floorplan solution found out of three independent runs (using different random number seeds) of the genetic floorplanner.

3.7.1 Performance of the Proposed Genetic Operators

To demonstrate the performance of the proposed crossover operators, the proposed genetic floorplanner is compared with another genetic algorithm based floorplanner [29] (referred to as AdaptGA in the rest of this chapter) that also uses the Sequence Pair based solution encoding. AdaptGA was chosen for comparison as it reports the best results, to the best of the authors’ knowledge, for a genetic floorplanner that encodes its solutions using the Sequence Pair representation. This ensures that the performance improvements obtained can be attributed to the proposed crossover operators as the solution encoding is the same for both the floorplanners. The proposed floorplanner easily outperformed AdaptGA as can be seen in Table 3.3 obtaining an average wirelength savings of 25.3% with almost no increase in area. In fact, the proposed genetic
floorplanner produces better area than AdaptGA for the ami33 benchmark but increases the floorplan area slightly (1.75%) for the ami49 benchmark. It is to be noted that the area and total wirelength results reported for the proposed genetic floorplanner belong to a single individual present in the best non-dominated front of the final population. It is to be noted that the AdaptGA does not perform simultaneous area and wirelength optimization. The results from the proposed genetic floorplanner are compared against the best area and wirelengths reported by AdaptGA.

The runtimes of the proposed GA were faster than AdaptGA but are not directly comparable as AdaptGA was run on an UltraComp model60 workstation (clock speed and memory not reported). Specifically, the average running time (over 3 runs) of the proposed genetic floorplanner per pareto front solution was approximately 0.433 seconds and 1.12 seconds for the ami33 and ami49 benchmarks respectively on the previously mentioned Linux machine.

3.7.2 Comparisons against State-of-the-art Floorplanners

To demonstrate the effectiveness of the proposed floorplanner for simultaneous area and wirelength optimization in outline-free floorplanning, the proposed floorplanner is compared with two state-of-the-art floorplanners that perform simultaneous optimization of area and wirelength. The proposed genetic floorplanner is compared with Sheqin et al [24] (referred to as QP-LFF in the rest of the chapter) which claims the best results for the MCNC benchmarks among floorplanners that simultaneously optimize both area and wirelength, outperforming Enhanced O-tree [18], TCG [19], and SA-LP [23] (refer to [24] for details). Comparisons are also made with the publicly available SA-based Parquet floorplanner which uses the single normalized weighted sum (SNWS) approach. Tables 3.3 and 3.4 summarize the results of the area and total wirelength comparisons for the MCNC and GSRC benchmarks respectively.

The proposed genetic floorplanner outperforms QP-LFF in terms of wirelength for both the ami33 and ami49 benchmarks. QP-LFF does not report results for the other MCNC
benchmarks. The proposed floorplanner yielded an average wirelength savings of 26.9% when compared to QP-LFF for a very marginal 2.63% increase in area, which is justifiable given the significant wirelength savings obtained.

- **Comparisons against Parquet** – The most recent version (4.5.23) of the publicly available Parquet tool [21] [22] was obtained, compiled (with the –O3 flag) and installed on the same Linux compute cluster mentioned above. The outline-free floorplanning results for Parquet were obtained using the following flag settings:

  ```
  -FPrep SeqPair –minWL –areaWeight 0.5 –wireWeight 0.5
  ```

  The population based approach and the NSGA-II template of the proposed genetic floorplanner ensure that multiple floorplan solutions belonging to a pareto-optimal set are available to the user. Thus the user can choose from numerous pareto-optimal solutions and analyze the trade-offs involved between area and wirelength objectives for the particular problem instance. But the Parquet floorplanner can only yield a single solution from each of its runs. Since the Parquet tool has to be run multiple times with different objective weight vectors to obtain different solutions, a more fair comparison of the tools’ runtimes should use the time ($t_{pg}$) to obtain a single solution of the final pareto-front for comparison against the runtime of the SA-based optimizer. The size of the final Pareto front was recorded for each run of the proposed floorplanner to compute the runtime per pareto front solution (last column of Table 3.4). Table 3.4 reports the area, wirelength and runtime results obtained by the proposed genetic floorplanner on the GSRC benchmarks. The Parquet tool was then run for the same time using the “-t” option. The area and wirelength of the best floorplan solution out of three independent runs are used for the comparisons shown in Table 3.5. The ratio of runtimes shown in Table 3.5 is the ratio of the runtime of the proposed genetic floorplanner by the runtime of the Parquet floorplanner to obtain the reported savings.
Table 3.4. Area, wirelength and runtime results for the proposed genetic floorplanner on the GSRC benchmarks

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Area Min</th>
<th>Area Avg</th>
<th>Wirelength Min</th>
<th>Wirelength Avg</th>
<th>Best Individual Area</th>
<th>Best Individual WL</th>
<th>Total Runtime (s)</th>
<th>RunTime /PFsoln (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n10</td>
<td>228492</td>
<td>242455.7</td>
<td>30375</td>
<td>32407.27</td>
<td>238120</td>
<td>32138.33</td>
<td>46.67</td>
<td>0.47</td>
</tr>
<tr>
<td>n30</td>
<td>224186</td>
<td>226355.3</td>
<td>86223</td>
<td>89287.27</td>
<td>227319.7</td>
<td>89065.33</td>
<td>651.48</td>
<td>2.17</td>
</tr>
<tr>
<td>n50</td>
<td>213498</td>
<td>219323.7</td>
<td>118757</td>
<td>122803</td>
<td>218961</td>
<td>122731.3</td>
<td>1745.55</td>
<td>3.49</td>
</tr>
<tr>
<td>n100</td>
<td>198592</td>
<td>200955</td>
<td>191626</td>
<td>195489.3</td>
<td>200776.3</td>
<td>197819</td>
<td>9176.95</td>
<td>9.18</td>
</tr>
<tr>
<td>n200</td>
<td>201600</td>
<td>204484</td>
<td>354726</td>
<td>359090.3</td>
<td>203908.3</td>
<td>366423.7</td>
<td>42619.37</td>
<td>21.31</td>
</tr>
<tr>
<td>n300</td>
<td>314130</td>
<td>320387</td>
<td>505139</td>
<td>521463.4</td>
<td>320045.6</td>
<td>517739.2</td>
<td>60822.62</td>
<td>1645.47</td>
</tr>
</tbody>
</table>

Table 3.5. Percentage savings in Area and Wirelength of the Proposed Genetic Floorplanner compared against the PARQUET Floorplanner for the GSRC benchmarks

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Best Individual Area</th>
<th>Best Individual WL</th>
<th>Ratio of Runtimes</th>
</tr>
</thead>
<tbody>
<tr>
<td>n10</td>
<td>0.44</td>
<td>16.90</td>
<td>0.9881</td>
</tr>
<tr>
<td>n30</td>
<td>-1.44</td>
<td>22.69</td>
<td>1.06104</td>
</tr>
<tr>
<td>n50</td>
<td>0.01</td>
<td>14.98</td>
<td>1.05365</td>
</tr>
<tr>
<td>n100</td>
<td>-0.78</td>
<td>15.38</td>
<td>1.08947</td>
</tr>
<tr>
<td>n200</td>
<td>-3.18</td>
<td>22.42</td>
<td>1.0373</td>
</tr>
<tr>
<td>n300</td>
<td>-7.82</td>
<td>12.94</td>
<td>1.06869</td>
</tr>
<tr>
<td></td>
<td>-2.13</td>
<td>17.55</td>
<td></td>
</tr>
</tbody>
</table>

For the MCNC benchmarks, the proposed floorplanner outperforms Parquet in terms of both area and wirelength producing 20.79% average wirelength savings and an average area savings of 4.88%. Fig. 18 shows one of the best floorplans obtained using the proposed GA for the ami33 benchmark.

For the GSRC benchmarks, the proposed genetic floorplanner outperforms Parquet in terms of total wirelength but at the cost of a small area penalty. Specifically, the proposed genetic floorplanner obtains 17.55% average wirelength savings for an average area increase of 2.13%. It is to be noted that the proposed genetic floorplanner produces better wirelength results for all the benchmarks. Fig. 19 shows one of the best floorplans
obtained by the proposed floorplanner for the n100 GSRC benchmark. Considering both the benchmark suites, the proposed genetic floorplanner obtains an average savings of 19.17% in wirelength and 1.375% in area over the Parquet floorplanner.

Figure 3.18. Floorplan of ami33 benchmark with area = 1.21 mm$^2$ and total wirelength = 35.43 mm

Figure 3.19. Floorplan of n100 GSRC benchmark with area = 205,758 sq.units and wirelength = 133,497 units
3.8 Summary

VLSI Floorplanning has transformed into a multi-objective optimization problem with the recent advances in integration technology. Genetic algorithms have been extensively used in different forms to solve various multi-objective optimization problems. In this work, the NSGA-II multi-objective genetic algorithm has been applied to tackle the VLSI floorplanning problem considering the floorplan area and total wirelength objectives. Novel crossover operators have been developed for effective floorplanning using Sequence Pairs. The hybridized multi-objective floorplanner achieves very good results for the MCNC and GSRC benchmark suites as compared to other floorplanners that perform simultaneous optimization of area and wirelength. Thus, genetic algorithms can be used effectively for multi-objective optimization in VLSI design when equipped with well designed genetic operators.
CHAPTER 4
DESIGN OF AN FPGA BASED GENERAL PURPOSE GENETIC ALGORITHM IP CORE

Genetic algorithms have been shown to be a robust search mechanism that can be used as an effective optimization engine in a wide variety of applications [6, 7], which include reconfigurable hardware applications and real-time applications. Genetic algorithms can explore multiple regions of the problem’s solution space concurrently but can incur significant runtimes due to their population based search mechanism. A natural solution to speed up the genetic algorithm is to implement it in hardware.

Another advantage of a hardware implementation of a GA is the elimination of the need for complex time and resource consuming communication protocols needed by an equivalent software implementation to interface with the main application. This is particularly advantageous to real-time applications such as reconfiguration of evolvable hardware. Moreover, with the rapidly increasing FPGA logic density, efficiently designed hardware genetic algorithms can be implemented on a single FPGA in addition to the target application resulting in a less bulky apparatus.

In this work, a robust parameterized genetic algorithm IP core is proposed that is readily synthesizable using standard FPGA design tools at both the RT-level and gate-level. The programmable IP core can be easily integrated with any application that requires a search engine and can also be implemented in a system-on-a-chip configuration. Its architecture is extremely flexible and easy to integrate with target applications allowing seamless integration of user defined
blocks such as fitness function modules. The core has been implemented on a Xilinx Virtex2Pro FPGA device (xc2vp30-7ff896). It has a very small footprint (only 13% slice utilization) and runs at a high speed (50MHz). This IP core is highly suitable for Evolvable Hardware (EHW) [59] applications. It is one of the building blocks of the Self-Reconfigurable Analog Array architecture and is used to compensate extreme temperature effects on VLSI electronics [68][69].

The novel contributions of this work are that the proposed GA IP core:

- is available at different design levels, RT-level and gate-level, to provide the end-user flexibility in choosing the design level at which to include the GA IP core,
- supports user-defined fitness functions without the need for re-synthesis of the entire design,
- allows programming of the initial seed for the Random Number Generator (RNG) that enables different convergence characteristics to be obtained for the same GA parameter settings,
- provides PRESET modes, allowing the user to readily experiment with a varied set of predefined GA parameter settings, and
- can be directly implemented as a digital ASIC using standard ASIC design tools with simple scan chain testability built into the core and with basic fault tolerance in the form of PRESET modes to bypass parameter initialization failure.

In addition, the proposed GA IP core has several highly desirable features:

- **Programmability** – values of important GA parameters including population size, number of generations, crossover rate, and mutation rate can be programmed to accommodate the requirements of a wide variety of applications,
- **High Probability of Convergence to Optimal Solution** – an elitist GA model is used that can converge to the global optimum [8, 9], and
• *Easy Interfacing* – simple two-way handshaking protocol to interface with user defined fitness evaluation module and the target application.

### 4.1 Background and Related Work

In this section, we will review previous hardware implementations of general purpose genetic algorithms. We will also review some background material on Evolvable Hardware and discuss how the previous hardware implementations fail to address key design issues of Evolvable Hardware.

#### 4.1.1 Prior Work

There have been many hardware implementations of both general-purpose [41-46] and application-specific [52] genetic algorithms. This section will review previous FPGA implementations of a general purpose genetic algorithm. Table 4.1 summarizes the existing works on FPGA implementation of general-purpose genetic algorithm.

Several application specific hardware implementations of a genetic algorithm [52] exist in literature, tailored to the particular application in terms of chromosome encoding, crossover, and mutation operations. These implementations will not be reviewed here as they cannot be re-used with other applications even for prototyping purposes.
Table 4.1. Review of existing literature on FPGA implementation of a general-purpose genetic algorithm (“N/A” : Not Applicable, “-“ : UNKNOWN)

<table>
<thead>
<tr>
<th>Work</th>
<th>Elitist</th>
<th>Pop. Size</th>
<th>No. Gens</th>
<th>Selection</th>
<th>Crossover/ Mutation rates</th>
<th>Crossover Operators</th>
<th>RNG/Seed</th>
<th>Preset Modes</th>
<th>Initialize Mode</th>
<th>FPGA platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scott et al [5]</td>
<td>N</td>
<td>fixed (16)</td>
<td>fixed</td>
<td>Roulette</td>
<td>fixed</td>
<td>1-point</td>
<td>CA/fixed</td>
<td>none</td>
<td>none</td>
<td>BORG board</td>
</tr>
<tr>
<td>Tommiska and Vuori [6]</td>
<td>N</td>
<td>fixed (32)</td>
<td>fixed</td>
<td>Round robin</td>
<td>fixed</td>
<td>1-point</td>
<td>LSHR/fixed</td>
<td>none</td>
<td>none</td>
<td>Altera</td>
</tr>
<tr>
<td>Shackleford et al [7]</td>
<td>N</td>
<td>fixed</td>
<td>fixed</td>
<td>Survival</td>
<td>fixed</td>
<td>1-point</td>
<td>CA/fixed</td>
<td>none</td>
<td>none</td>
<td>Aptix</td>
</tr>
<tr>
<td>Yoshida et al [8]</td>
<td>N</td>
<td>64 or 128</td>
<td>fixed</td>
<td>Simplified tourney</td>
<td>-</td>
<td>1-point</td>
<td>CA/fixed</td>
<td>none</td>
<td>none</td>
<td>SFL (HDL)</td>
</tr>
<tr>
<td>Tang and Yip [9]</td>
<td>-</td>
<td>prog.</td>
<td>prog.</td>
<td>Roulette</td>
<td>prog.</td>
<td>1-point, 4-point, uniform</td>
<td>fixed</td>
<td>none</td>
<td>-</td>
<td>PCI card based system</td>
</tr>
<tr>
<td>Aportewan et al [10]</td>
<td>N/A</td>
<td>fixed (256)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>CA/fixed</td>
<td>none</td>
<td>none</td>
<td>none</td>
<td>Xilinx Virtex1000</td>
</tr>
<tr>
<td>Proposed</td>
<td>Y</td>
<td>prog. (8-bit)</td>
<td>prog. (32-bit)</td>
<td>Roulette</td>
<td>prog. (4-bit)</td>
<td>1-point</td>
<td>CA/prog.</td>
<td>3 diff. modes</td>
<td>separate init. mode (two-way handshake)</td>
<td>Xilinx Virtex2Pro FPGA</td>
</tr>
</tbody>
</table>

65
The first FPGA implementation of a general purpose GA engine was proposed by Scott et al [41] who described a modular hardware implementation of a simple genetic algorithm that used roulette-wheel selection, one-point crossover with a fixed population size of 16, and member width of 3 bits. The genetic algorithm was broken into simpler modules and each module was described using behavioral VHDL. The overall GA design was implemented on multiple Xilinx FPGAs on a BORG board. The main goal of [41] was to illustrate the issues of hardware implementation of a general purpose GA.

Tommiska and Vuori [42] implemented a general purpose GA system with round-robin parent selection and one-point crossover and used a fixed population size of 32. The GA was implemented on Altera FPGAs mounted on PCI cards and connected to the host computer’s CPU using high performance PCI buses. Experimentation on various fitness functions involved rewriting the AHDL code and reprogramming the FPGAs.

Shackleford et al [43] implemented a survival-based, steady state GA in VHDL to achieve higher performance and tested it on set-covering and protein folding problems. The prototype GA machine used for the set-cover problem was designed using the Tsutsuji logic synthesis system and implemented on an Aptix AXB-MP3 Field Programmable Circuit Board (FPCB) populated with six FPGAs.

Yoshida et al [44] implemented a Genetic Algorithm Processor with a steady state architecture that supports efficient pipelining and a simplified tournament selection.

Tang and Yip [45] implemented a PCI-based hardware GA system using two Altera FPGAs mounted on a PCI board. The PCI-based GA system has multiple crossover and mutation operators implemented with programmable crossover and mutation thresholds. Tang and Yip also discussed different parallel implementations of the PCI-based GA system.

In contrast to the simple GA, Aportewan et al [46] implemented a compact GA in Verilog HDL as it is more amenable towards a hardware implementation. However, compact GAs suffer
from a severe limitation in that their convergence to the optimal solution is guaranteed only for the class of applications that possess tightly coded, non-overlapping building blocks [46].

Hardware acceleration techniques such as pipelining and parallel architectures have been applied to the design of hardware genetic algorithms [47-49]. Such advanced techniques are not the main focus of this work and will not be discussed here.

The motivations of all the previous FPGA implementations fall under one or more of the following categories:

- **Basic Hardware Acceleration** – to obtain speedup over the corresponding software implementation [41-45],
- **Novel GA Templates** – to propose a novel genetic algorithm template or architecture [46] that is more suited for a hardware implementation,
- **Advanced Hardware Acceleration Techniques** - to accelerate a genetic algorithm using pipelined, and/or parallel implementations of GA-architectures [47-49].

The primary goal of the above efforts is to demonstrate the speedup that can be achieved by a hardware GA implementation. As a result, the prototypes developed in the above FPGA-based implementations suffer from one or more of the following limitations:

- **Lack of programmability for GA parameters** – Some or all of the GA parameters are fixed. To the best of the authors’ knowledge, the only FPGA implementation that has customizable parameters is the GA machine proposed by Tang and Yip [45].
- **Lack of scalability in terms of fitness functions** – Only a single fitness function is supported. To accommodate a new fitness function, the entire design has to re-synthesized. All the previous implementations suffer from this limitation.
- **Pre-defined system architecture/organization** – The GA architecture is defined based on a specific development environment imposing serious restrictions on the target application. For example, in Tang and Yip [45], the GA machine can only be implemented on a PCI
card that contains two FPGAs with a local PCI bus providing the communication between the different modules.

The proposed GA IP core overcomes all of the abovementioned limitations with the following features:

- **Independent Parameter Initialization Phase** – The proposed GA core has a separate initialization phase that enables the user to program the desired GA parameters including population size, number of generations, crossover threshold, mutation threshold, and the initial seed used by the random number generator.

- **Compact IP core with Simple Communication Interface** – The proposed GA core does not impose any hardware requirements or system architecture on the user. It can be instantiated within the main application as a drop-in IP module and synthesized along with the application.

- **Support for External Fitness Functions** – The proposed core allows the user to provide external fitness values by multiplexing between the internal and external fitness values and the interfacing signals. It can select from the existing internal fitness function or an external fitness function supplied by the user using another FPGA or from a PC. This eliminates the need to re-synthesize the entire design just to accommodate a new fitness function. This is a very desirable feature especially for intrinsic EHW applications and other space applications that cannot re-program the on-board FPGA without significant effort and down-time.

Besides the FPGA implementations, ASIC implementations of GAs [50,51] have also been proposed to improve the performance of the GA using hardware acceleration. Wakabayashi et al [50] proposed a Genetic Algorithm Accelerator (GAA) chip that implements an elitist generational GA with on-the-fly adaptive selection between the two-point and uniform crossover operators. The GAA chip was fabricated using 0.5um standard CMOS technology.
Chen et al [51] developed a GA chip using 0.18um TSMC cell library. They developed a software tool called Smart GA to tailor the GA module to user specifications accepted through a front-end GUI. Any re-programming of the GA parameters will require re-synthesizing the GA netlist and repeating the physical design process to obtain the ASIC. This is a significant problem as, in most cases, the user cannot predict the best GA parameter settings for his/her application. If the current user settings do not offer the best performance, then the user has to re-synthesize the entire GA netlist with new parameter settings and re-design the entire ASIC. The programmable GA IP core proposed in this work eliminates the need for such re-synthesis.

4.1.2 Pseudo-random Number Generation and GA Performance

A genetic algorithm requires random numbers for generation of the initial population, crossover and mutation. True random numbers can be generated using specialized hardware that extract the random numbers from a non-deterministic source such as clock jitter in digital circuits [53]. Pseudo-random number generators (PRNG) use a deterministic algorithm to generate the random numbers. Hence, the sequence of random numbers can be predicted if the initial seed is known. The choice of random number generators depends upon the application at hand. Applications that require high security will use true random number generators. Applications that require a quick response and cannot afford the high area-overhead of true random number generators will use pseudo-random number generators.

The effect of the quality of the random number generators on the performance of genetic algorithms has been previously studied [54-56]. A high quality random number generator is generally characterized by a long period (before repetition of the random numbers), uniformly distributed random numbers, absence of correlations between consecutive numbers, and structural properties such as organization of the numbers in lattices. Meysenburg [54] and Meysenburg and Foster [55] reported little or no improvement of performance of GAs using good PRNGs over
those using poor PRNGs. However, later Cantu-Paz [56] found significant improvements on the performance of a simple binary GA when using a good PRNG. Cantu-Paz found that the quality of the random numbers used to generate the initial population has a major impact on the performance of the GA while it did not affect the performance of the GA significantly for all the other operations such as crossover and mutation.

The seed used by a Random Number Generator (RNG) influences the sequence of numbers generated. Although the RNG characteristics like cycle length and uniform distribution will remain the same with a different seed, the sequence of random numbers generated will differ. A poorly chosen seed can lead to a poor quality of random numbers generated by a good RNG. Guidelines to choosing a good seed can be found in Garfinkel and Spafford [57]. The performance of algorithms depending on random numbers has been shown to vary with the RNG seed. Elsner [58] studied the influence of the RNG seeds on the performance of four different graph partitioning algorithms. In a particular instance, Elsner observed that the performance worsened by up to 5 times by changing only the RNG seed.

Theoretically, a good RNG will produce random numbers that are uniformly distributed for a large enough sample size. But due to time constraints (of real-time applications), the distribution of the random numbers generated might be non-uniform. This might lead to poor results for resource-constrained hardware genetic algorithms. It has been observed by Meysenburg and Foster [55] and Cantu-Paz [56] that poor RNGs can sometimes outperform good RNGs for particular seeds. Thus, a user will have to experimentally determine the RNG seed value for his particular application. This is particularly necessary for hardware implementations where simple RNG implementations are preferred due to tight resource and response time requirements. Hence, the proposed IP core allows the user to program the RNG seed in addition to providing three in-built seeds to select from.
4.1.3 Basics of Evolvable Hardware

The basics of Evolvable Hardware [59] will be briefly discussed here as space applications are increasingly employing them to adapt on-board electronics to the changing environmental conditions. Evolvable Hardware (EHW) is a class of hardware that adapts itself to changing conditions using evolutionary algorithms. There are two major divisions of evolvable hardware, namely, extrinsic EHW and intrinsic EHW. Extrinsic EHW refers to hardware that is evolved using software simulations (and behavioral models of the hardware). The best configuration found in the simulations is then downloaded on to the actual hardware. Intrinsic EHW refers to the adaptation and re-configuration of previously configured hardware because of changes observed or required in the actual hardware.

The growing number of remote space applications has increased the demand for intelligent and adaptive space systems [67]. Thus, intrinsic EHW is becoming popular in space applications. Intrinsic EHW have been targeted for different platforms including ASICs [60-63], and specialized platforms [64]. Due to the flexibility and scalability requirements of space applications, most of the existing works on intrinsic EHW have been implemented on FPGAs [37, 38].

Intrinsic EHW can be classified into four different classes based on the location of the reconfigurable hardware and the evolutionary algorithm as proposed by Lambert et al [65]:

- **PC-based Intrinsic EHW** – The reconfigurable hardware application is located on an FPGA and the monitoring system is located in the PC. The reconfiguration of the evolvable hardware is done from the PC. This system suffers from a slow runtime because of the communication with the PC.

- **Complete Intrinsic EHW** – Both the reconfigurable hardware and the evolutionary algorithm are situated on the same (FPGA) chip. This system will yield the best performance as the communication delays are due to intra-chip wires.
• *Multi-chip Intrinsic EHW* – The reconfigurable hardware and the evolvable hardware are located on different (FPGA) chips. The performance of this system is lower than the complete intrinsic EHW solution due to the inter-chip communication delay.

• *Multi-board Intrinsic EHW* – The reconfigurable hardware and the evolvable hardware are located in FPGA chips on different boards. The performance of this system is lower than both multi-chip and complete intrinsic EHW solutions due to the inter-board communication delays.

Although the Complete Intrinsic EHW implementation (especially on an ASIC) yields the best performance and smallest form factor, it is not widely adopted as it suffers from low scalability and flexibility issues with respect to the fitness function computation. The multi-chip and multi-board implementations are considered better for intrinsic EHW due to the dynamic reconfiguration features available in FPGAs (see Lambert et al [65] for more details).

The proposed core alleviates this problem by supporting the interfacing of fitness functions housed on other chips (or boards) to the existing system, thus allowing the realization of a hybrid system, as shown in Figure 4.3. The proposed core allows the user to select between internal and external fitness functions. Hence, even if the existing system is implemented on an ASIC, new fitness functions can be added externally to the system. It is to be noted that the proposed IP core can be used to realize all classes of intrinsic EHW systems (excluding PC-based) both on an FPGA and on an ASIC without losing out on flexibility.

4.2 Proposed FPGA Based Genetic Algorithm IP Core

This section describes in detail the implementation and interfacing details of the proposed core and the design issues considered for ASIC development and space applications.
4.2.1 Implementation and Interfacing Details

In this sub-section, the design methodology, behavioral modeling, and interfacing of the proposed GA IP core are presented in detail.

- **Design Methodology** – The entire behavior of the proposed GA core was modeled in VHDL and simulated to test its correctness. A Register Transfer (RT) level VHDL model of the GA core was synthesized from the behavioral model using an in-house High Level Synthesis tool called Automatic Design Instantiation (AUDI). The RT-level VHDL model was simulated thoroughly to test the correctness of the synthesized netlist. A gate-level Verilog model was then synthesized from the RT-level model using in-house flattening scripts and the Berkeley SIS tool [66]. The gate-level Verilog model uses simple Boolean gates such as NAND, NOR, AND, OR, XOR, and SCAN_REGISTER. The gate-level Verilog model was also simulated using Cadence NC-Launch to verify the functionality and the timing. This design methodology ensures that the RT-level and the gate-level netlists are completely synthesizable by standard synthesis tools such as the Xilinx ISE tool. Thus, a synthesizable Genetic Algorithm FPGA IP core is available to the user at two levels of design abstraction, namely RT-level and gate-level.

- **Behavioral Modeling** – The proposed GA core implements the GA optimization cycle shown in Figure 4.1. An initial population of randomly generated individuals is formed. A 16-bit cellular automaton based Random Number Generator, similar to the implementation in [41], is used to generate all the required random numbers. In each generation, a new population of candidate solutions is generated using crossover and mutation operators. Elitism is provided by copying the best individual in the current generation into the next generation.
Figure 4.1. High level view of the implemented GA optimization cycle

- **Parent Selection** – The parent individuals required for crossover are selected from the current population using the Proportionate Selection scheme [3]. A threshold value is computed by scaling down the sum of the fitnesses of all the individuals in the current population using a random number. A cumulative sum of the fitnesses of the individuals in the new population is computed and compared to the threshold value. The individual whose fitness causes the cumulative sum to exceed the threshold is chosen as the parent. This ensures that highly fit individuals have a selection probability that is proportional to their fitness. To speed up computations, the sum of the fitnesses of the new population is accumulated when the offspring’s fitness is computed.

- **Single Point Binary Crossover** – The GA core implements the single-point binary crossover technique [3], illustrated in Figure 2.2, to combine parents from the current
generation and produce offspring for the next generation. The GA core performs crossover only if the random number generated is less than the specified crossover threshold. Since the 4-bit crossover threshold is user-programmable, the user can control the crossover rate in the GA core. The single point crossover is implemented by using a bit mask vector that generates the first portion of the offspring from the first parent and the other portion of the offspring from the second parent. A random number $n$ is generated to denote the random cutpoint on the parents’ chromosomes. A mask is then generated with 1s from position 0 to $n-1$ and 0s after $n$. This mask is logically ANDed with the first parent’s chromosome to obtain the first part of the offspring. The mask is then logically inverted and ANDed with the second parent’s chromosome to obtain the second part of the offspring.

- **Mutation** – Mutation is performed after crossover in the proposed GA core. The GA generates a 4-bit random number and compares it with the selected mutation threshold to decide if mutation should be performed. If the random number is smaller than the mutation threshold, a random bit mutation is performed. A randomly chosen mutation point dictates the appropriate bit mask to be used in an XOR operation with the candidate solution. This XOR operation essentially flips the bit at the mutation point.

The fitness of the resultant offspring is then computed using a simple two-way handshaking communication between the GA core and the fitness evaluation module. The candidate and its fitness are then stored in the GA memory as part of the new population. The above cycle of parent selection, crossover, and mutation is repeated in every generation until the new population is completely filled. The GA optimization ends when the generation index is equal to the user-programmed number of generations. Then, the GA core exits the optimization cycle and outputs the best individual found.
Table 4.2. Port interface of the proposed GA core

<table>
<thead>
<tr>
<th>No.</th>
<th>Port</th>
<th>Input/Output</th>
<th>Width in bits</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>reset</td>
<td>I</td>
<td>1</td>
<td>System reset</td>
</tr>
<tr>
<td>2</td>
<td>sys_clock</td>
<td>I</td>
<td>1</td>
<td>System clock</td>
</tr>
<tr>
<td>3</td>
<td>ga_load</td>
<td>I</td>
<td>1</td>
<td>Load GA parameters</td>
</tr>
<tr>
<td>4</td>
<td>index</td>
<td>I</td>
<td>3</td>
<td>Index of GA parameter</td>
</tr>
<tr>
<td>5</td>
<td>data_valid</td>
<td>I</td>
<td>1</td>
<td>Initialization Handshake signal</td>
</tr>
<tr>
<td>6</td>
<td>data_ack</td>
<td>O</td>
<td>1</td>
<td>Initialization Handshake signal</td>
</tr>
<tr>
<td>7</td>
<td>fit_value</td>
<td>I</td>
<td>16</td>
<td>Fitness value bus</td>
</tr>
<tr>
<td>8</td>
<td>fit_request</td>
<td>O</td>
<td>1</td>
<td>Fitness request signal</td>
</tr>
<tr>
<td>9</td>
<td>fit_valid</td>
<td>I</td>
<td>1</td>
<td>Fitness value validity signal</td>
</tr>
<tr>
<td>10</td>
<td>candidate</td>
<td>O</td>
<td>16</td>
<td>Candidate solution bus</td>
</tr>
<tr>
<td>11</td>
<td>mem_address</td>
<td>O</td>
<td>8</td>
<td>GA Memory address</td>
</tr>
<tr>
<td>12</td>
<td>mem_data_out</td>
<td>O</td>
<td>32</td>
<td>Data to GA memory</td>
</tr>
<tr>
<td>13</td>
<td>mem_wr</td>
<td>O</td>
<td>1</td>
<td>GA Memory Write signal</td>
</tr>
<tr>
<td>14</td>
<td>mem_data_in</td>
<td>I</td>
<td>32</td>
<td>Data from GA memory</td>
</tr>
<tr>
<td>15</td>
<td>start_GA</td>
<td>I</td>
<td>1</td>
<td>GA Start signal</td>
</tr>
<tr>
<td>16</td>
<td>GA_done</td>
<td>I</td>
<td>1</td>
<td>GA completion signal</td>
</tr>
<tr>
<td>17</td>
<td>test</td>
<td>I</td>
<td>1</td>
<td>Scan chain Test signal</td>
</tr>
<tr>
<td>18</td>
<td>scanin</td>
<td>I</td>
<td>1</td>
<td>Scan chain input</td>
</tr>
<tr>
<td>19</td>
<td>scanout</td>
<td>O</td>
<td>1</td>
<td>Scan chain output</td>
</tr>
<tr>
<td>20</td>
<td>preset</td>
<td>I</td>
<td>2</td>
<td>Preset Mode Selector</td>
</tr>
<tr>
<td>21</td>
<td>rn</td>
<td>I</td>
<td>16</td>
<td>Random number</td>
</tr>
</tbody>
</table>

- **Programmable GA parameters** – The proposed GA core has a port interface as shown in Table 4.2. The performance and runtime of a genetic algorithm depend on GA parameters, namely population size, number of generations, crossover rate, and mutation rate. Large values for the population size and the number of generations generally yield the best results at the expense of long runtimes. But if the target application is simple, a few generations and a small population size may suffice to find the best solution.
Table 4.3. Index values of the GA core’s programmable parameters

<table>
<thead>
<tr>
<th>Index</th>
<th>Programmable Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Number of Generations [15:0]</td>
</tr>
<tr>
<td>1</td>
<td>Number of Generations [31:16]</td>
</tr>
<tr>
<td>2</td>
<td>Population Size</td>
</tr>
<tr>
<td>3</td>
<td>Crossover Rate</td>
</tr>
<tr>
<td>4</td>
<td>Mutation Rate</td>
</tr>
<tr>
<td>5</td>
<td>RNG Seed</td>
</tr>
</tbody>
</table>

An efficient GA core implementation must have the ability to cater to the needs of the individual applications by allowing the user to change these parameters according to the application. The crossover and mutation rates that produce the best results in the shortest amount of time also vary with the application. Hence, the proposed GA core also has the capability to program both the crossover and mutation rates. The quality of the random numbers generated for the execution of the genetic operators also has an impact on the performance of the GA. The proposed core allows the user to program the initial seed of the Random Number Generator (RNG) which enables the user to obtain different sequences of random numbers using the same RNG module.

All the programmable parameters of the GA core must be initialized before it can be used. Initialization of the GA core is done using a simple two-way handshake. The user first asserts the `init_GA` signal to put the GA core in the initialization mode. Then, all the programmable parameters can be initialized using the handshaking process described below. Each programmable parameter has an index associated with it as shown in Table 4.3. The user places the value of the programmable parameter on the `fit_value` bus and the corresponding index value on the `index` bus. The user then asserts the `data_valid` signal. The GA core reads the `fit_value` bus, decodes the `index` and stores the value in the appropriate register. The GA core then asserts the `data_ack` signal and waits for `data_valid` to be de-asserted before de-asserting `data_ack`. 

77
Interfacing details of the GA core – The overall GA optimizer consists of three modules, namely, GA core, GA memory, and a random number generator (RNG). The GA core communicates with a fitness evaluation module and the actual application using simple two-way handshaking protocol. A typical system with the communication between all these modules is shown in Figure 4.2.

The GA memory module is a single port memory module that stores both the individuals and their fitnesses. To store an individual and its fitness, the GA core places the memory contents on the memory bus and asserts the memory write signal. To read an individual and its fitness, the GA core places the memory address on the address bus and reads the memory contents in the next clock cycle.

The RNG module is implemented using a cellular automaton. It is to be noted that the operation of the GA core is independent of the RNG implementation. The initial seed of the RNG module can be provided by the user. One of three preset initial seeds can also be selected in the PRESET mode. The GA core reads the output register of the RNG module when it needs a random number. Based on the number of random bits needed, the GA selects the bits from pre-defined positions.

The GA core uses a simple two-way handshaking protocol for its communication with the fitness evaluation module. When the GA core requires the fitness of a candidate solution, it places the individual on the candidate bus and then asserts the fit_request signal. The FEM module of the target application should then read the candidate port and compute the fitness of the individual. The computed fitness is then placed on the fit_value port of the GA core and the fit_valid signal is asserted by the target application. On assertion of the fit_valid signal, the GA core reads the fitness value and de-asserts the fit_request signal. The simplicity of the interfacing protocol is a major advantage to the user as it reduces timing issues during implementation of the entire application.
Figure 4.2. Typical system showing the communication between the different modules and the GA core (signal numbers are in reference to Table 4.2)

- **Usage details of the GA core** – The Genetic Algorithm core starts its optimization cycle when it receives the \textit{start\_GA} pulse from the target application. If the programmable parameters of the GA have been initialized, it uses these values. Otherwise, the GA core can use one of the three preset modes. During its optimization cycle, the GA core requests fitness computations for candidate individuals from the fitness evaluation module using the handshaking protocol described. Once the GA core has computed the best candidate, it is placed on the \textit{candidate} bus and the \textit{GA\_done} signal is asserted.

4.2.2 **Design Considerations for ASIC Implementation and Space Applications**

The proposed GA core is well suited for ASIC development as it is available as a Verilog gate-level netlist, and also has three preset modes and a scan chain built into the design.
Table 4.4. Preset modes available in the proposed GA core

<table>
<thead>
<tr>
<th>Mode</th>
<th>Pop size</th>
<th>No. of generations</th>
<th>Thresholds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>crossover</td>
</tr>
<tr>
<td>User</td>
<td>00</td>
<td>&lt; 256</td>
<td>&lt; 2^32</td>
</tr>
<tr>
<td>Preset</td>
<td>01</td>
<td>32</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>64</td>
<td>1024</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>128</td>
<td>4096</td>
</tr>
</tbody>
</table>

- **Preset Modes** – The proposed GA core has three preset modes as shown in Table 4.4. The values for the programmable GA parameters in the preset modes have been set so that the GA core can be used for a varied set of applications without compromising on performance or runtime. The user can select any one of the three different preset modes based on the target application. When the 2-bit preset signal value is set to “00”, the GA core uses the user-programmed values for all the programmable parameters.

- **Scan Chain Testing** – The proposed GA core has a scan chain connecting all the registers used in the design. A scan chain test can be run on the core by asserting the test signal and feeding the user test pattern in the scanin port. The output of the scan chain can be observed on the scanout port. This scan chain can also be connected to a top-level scan chain in a system level design.

The increasing number of remote space missions has necessitated autonomous space crafts that are capable of handling unexpected situations and adapting to new environments [67]. This requires deployment of intrinsically evolvable hardware whose adaptation and re-configuration are controlled by on-board evolutionary algorithms. In [67], Stoica et al identify the following characteristics as the most critical in space-oriented EHW:

- **Systems Approach to EHW design** – The EHW system only helps to reconfigure and adapt the higher level application to the changing environment.
• *Flexibility in Fitness Calculation* – The means of computing and the context of the fitness of a candidate solution need to be considered.

• *Response Time* – Most of the space applications are time-critical applications that must adapt to the changing environment quickly before serious damage is done to the system and/or the mission itself.

• *Safety* – Space systems are very expensive systems that are highly sensitive to even small errors and/or environmental changes. Hence, safety of the space systems is the most critical characteristic as they can be permanently lost or damaged with the slightest of problems.

The proposed GA core addresses these issues in the following ways:

• Design of the GA core as a drop-in IP module and its capability to be integrated at various design levels enables a systems approach to EHW design.

• The proposed GA core supports external fitness functions.

• By providing the user the ability to program the number of generations according to the criticality of the application, the runtime of the GA can be controlled. Moreover, the best candidate of every generation is always output to the application to use in case of an emergency.
Implementation of a hybrid Intrinsic EHW system – The proposed GA core can be used to implement a scalable hybrid intrinsic EHW system as shown in Figure 4.3. The GA core enables the user to multiplex between an internal fitness function and an external fitness function. The fitness value (shown in bold in Figure 4.3) and the handshaking signals are available to the external fitness function module, which may be on a different chip/board and can be added on later by the user to expand the functionality of the system. The external fitness function can be housed on a reconfigurable fabric such as an FPGA if more external fitness functions are to be supported.
4.3 Experimental Results

The GA core was simulated and tested thoroughly at each level of design abstraction (behavioral, RT-level, and gate-level). Moreover, an analogous GA optimization cycle was implemented in software (C programming language) to compute the speed-up obtained by the FPGA implementation. This section will discuss in detail the various experiments conducted at each design level and the results obtained from simulation and hardware execution runs.

4.3.1 RT-level Simulations

At the RT-level, the GA core was simulated using Cadence NC-Launch to verify the functionality. The effectiveness of the GA core was tested by optimizing three maximization test functions shown below using various parameter settings. All the experiments use a chromosome length of 16. Hence all the single variable experiments have an X-variable range of 0 to 65535 and the two variable experiments have equal ranges (0 to 255).

- Test Function #1 (Binary F6) - \( BF6(x) = ((x^2 + x) \cos(x) / 4000000) + 3200 \)

  This is a very difficult test function that has numerous local maxima as can be seen in Figure 4.4 and has exactly one global maxima with a value of 4271 when \( x = 65522 \). This is a standard test function used to test the effectiveness of genetic algorithms and other optimization algorithms [5].
Figure 4.4. (Zoomed in) Plot of the modified Binary F6 [5] test function

- Test Function #2 \( F_2(x, y) = 8^*x - 4^*y + 1020 \)

  This is a simple minimax test function that has to maximize one variable \( x \) and minimize the other variable \( y \) to obtain the optimal objective function value of 3060.

- Test Function #3 \( F_3(x, y) = 8^*x + 4^*y \)

  This is a simple maximax test function that has to maximize both the variables \( x \) and \( y \) to obtain the optimal objective function value of 3060.

Table 4.5 summarizes the best results obtained for the three test functions under various parameter settings. The “convergence” column shows the generation number when the difference in average fitness between the current generation and next generation is less than 5%. It can be clearly seen that the proposed GA core finds the optimal values for all the test functions. But the optimum is found only for certain parameter settings underlining the need for programmability of the GA core’s parameters. It can also be seen that the random numbers used by the GA play a vital
role in determining the performance. For instance, in run #1 from Table 4.5, the GA core converges prematurely to a local optimum. But when the RNG seed is changed from 45890 to 10593 (Run#3), the convergence of the GA is better and the global optimum is found under the exact same settings for the other parameters.

Table 4.5. RT-level simulation results obtained for the three test functions (BF6, F1, and F2) under various GA parameter settings

<table>
<thead>
<tr>
<th>Test Function</th>
<th>Run Number</th>
<th>Initial RNG Seed</th>
<th>Population Size</th>
<th>Crossover Threshold</th>
<th>Best Fitness Value</th>
<th>Best Fitness Generation</th>
<th>Convergence (gen. num.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF6</td>
<td>1</td>
<td>45890</td>
<td>32</td>
<td>10</td>
<td>4047</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>45890</td>
<td>64</td>
<td>10</td>
<td>4271</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10593</td>
<td>32</td>
<td>10</td>
<td>4271</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1567</td>
<td>32</td>
<td>10</td>
<td>4146</td>
<td>2</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1567</td>
<td>32</td>
<td>12</td>
<td>4047</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>F2</td>
<td>6</td>
<td>45890</td>
<td>32</td>
<td>10</td>
<td>3060</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>45890</td>
<td>64</td>
<td>10</td>
<td>2096</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>F3</td>
<td>8</td>
<td>10593</td>
<td>32</td>
<td>12</td>
<td>3060</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>10593</td>
<td>64</td>
<td>10</td>
<td>3060</td>
<td>10</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1567</td>
<td>32</td>
<td>10</td>
<td>3060</td>
<td>16</td>
<td>20</td>
</tr>
</tbody>
</table>

The convergence plots for the three test functions under different parameter settings are shown in Figures 4.5 through 4.11. In these plots, the X-axis plots the generation number and the Y-axis plots an individual’s fitness value. Each point $P(i, j)$ is a population member in generation ‘$i$’ with a fitness value of ‘$j$’. For the sake of clarity, the plots show only one of multiple members with the same fitness in any generation. Hence, as the population converges to the best few candidates in the latter generations, the number of points reduces. Although many inferior members are present in the initial random population, the final generations contain highly fit individuals and very few inferior individuals (due to mutation).
Figure 4.5. Convergence plot for the BF6 test function using number of generations=32 – Run #3 of Table 4.5

Figure 4.6. Convergence plot for the BF6 test function with initial seed for RNG=1567 – Run #4 of Table 4.5
Figure 4.7. Convergence plot for the BF6 test function with crossover rate=0.75 – Run #5 of Table 4.5

Figure 4.8. Convergence plot for the test function F2 with population size=32 – Run #6 of Table 4.5
Figure 4.9. Convergence plot for the test function F2 with population size=64 – Run #7 of Table 4.5

Figure 4.10. Convergence plot for the test function F3 with initial seed for RNG=10593 – Run #9 of Table 4.5
Figures 4.8 through 4.11 show the convergence characteristics for the “hard” binary F6 test function in Figures 4.5 through 4.7 show that finding the optimal parameter settings for a difficult problem is non-trivial and that the initial seed for the RNG module is an important factor in GA convergence. For instance, changing the initial seed for the RNG module from 45890 in run #1 to 10593 in run #3 (while using the same values for all the other programmable parameters) improved the best solution found for the BF6 test function by about 5.5%, while for test function F2 the optimal result was found quickly with the initial RNG seed set at 45890. Thus, it is clear that the optimal GA parameter settings differ widely from function to function re-iterating the need for a customizable GA core.
4.3.2 FPGA Implementation Results

The gate-level Verilog netlist of the GA core was synthesized using the Xilinx ISE 10.1i tool and mapped to a Virtex2Pro (xc2vp30-ff896, speed grade -7) device. Table 4.6 shows the area utilization, clock speed, and block memory utilization for the placed and routed GA core on this Xilinx device. Block memory utilization is reported as it is not included in the logic utilization computation. It is to be noted that the dedicated block memory in the Xilinx Virtex-II Pro device implements both the GA memory module and the lookup based fitness evaluation module. The post place-and-route simulation model for the designed GA IP core was extracted and simulated using ModelSim to verify the functionality and timing. The design was then downloaded on to the FPGA device and its functionality was verified using Chipscope Pro 10.1i tools.

Table 4.6. Post place-and-route statistics for the proposed GA core on Virtex II Pro device (xc2vp30-7ff896)

<table>
<thead>
<tr>
<th>Design Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Utilization (% Slices Used)</td>
<td>13%</td>
</tr>
<tr>
<td>Clock Period</td>
<td>50MHz</td>
</tr>
<tr>
<td>Block Memory Utilization (GA Memory)</td>
<td>1%</td>
</tr>
<tr>
<td>Block Memory Utilization (Fitness Lookup Module)</td>
<td>48%</td>
</tr>
</tbody>
</table>

The effectiveness of the GA core was then tested by optimizing the three difficult maximization test functions shown below. The RT-level simulations used simple mini-max and maxi-max functions, and a difficult optimization test function. These simulations verified the functionality and the convergence characteristics of the proposed GA core. For the FPGA experiments, more complex test functions have been used to test the effectiveness of the GA core. These functions have been modified to enable easy hardware implementation.
• Modified and Scaled Binary F6 –

\[ m BF6 - 2(x) = 4096 + \left( \frac{\left( x^2 + x \right) \cos(x)}{2^{20}} \right) \]
\[ 0 \leq x \leq 65535. \]

This function is a modified and scaled version of the maximization test function from Haupt and Haupt [5]. It has a single globally optimal solution at \( x = 65521 \) with a value = 8183.

• Modified Binary F7 –

\[ m BF7 - 2(x, y) = 32768 + \left( 56^* [x^* \sin(4x) + 1.25^* y^* \sin(2y)] \right), \]
\[ 0 \leq x, y \leq 255. \]

This function is a modified version of the minimization function in [5]. It has been modified into a maximization function that has a single optimal solution with a value=63904 at \( x=247 \) and \( y=249 \).

• Modified 2D Shubert Function –

\[ m Shubert 2D(x_1, x_2) = 65535 - 174^* \left[ 150 + \left( \prod_{k=1}^{2} \sum_{i=1}^{5} \{ i \cdot \cos(i+1) \cdot x_k + i \} \right) \right], \]
\[ 0 \leq x_1, x_2 \leq 255. \]

This function is a minimization function (derived from [50]) modified into a maximization function with a global optimal value=65535. The function has 48 global optimal solutions and numerous local maxima.

The experimental setup is similar to the one shown in Figure 4.2. The Xilinx ISE 10.1i tool achieved a clock speed of 50MHz for the GA module (GA core, RNG module, and the GA memory). The initialization module and the application (fitness) module are separate entities that communicate with the GA module using handshaking. The Xilinx ISE tool was able to achieve a
clock speed of 200MHz for these modules. A Digital Clock Manager (DCM) core is used to generate the two clocks from the on-board 100MHz clock.

The initialization module consists of a simple Finite State Machine (FSM) to perform the two-way handshaking operation using the “data valid” and “data ack” signals to initialize the various GA parameters one-by-one. The application module contains a simple FSM to perform the two-way handshaking with the GA core and the hardware implementation of the fitness function. A lookup based implementation has been used for the fitness functions as this resulted in better operational speed than a combinational implementation. In the lookup based fitness computation method, block-ROMs within the FPGA device are populated with the fitness values corresponding to each solution encoding. The approach is used only to demonstrate the effectiveness of the proposed GA IP core in optimizing difficult maximization test functions without having to implement the actual test functions in hardware.

The entire experimental setup was implemented on the Xilinx Virtex2Pro (xc2vp30-7ff896) FPGA device. Chipscope Pro 10.1 tools were used to build cores to observe and record the “best fitness” and “sum of fitness” values for each generation on the FPGA. The proposed GA core was run with 12 different parameter settings as shown in Table 4.7 through Table 4.9. It is to be noted that mutation rate and number of generations were set to 0.0625 and 64 respectively for all the experiments. The number of generations was set to 64 as the population converged within 64 generations for all three fitness functions.

Table 4.7 tabulates the results obtained by the GA core for the mBF6_2 test function using different settings for the programmable GA parameters. In the experiments conducted, the best solution found by the proposed GA core for the mBF6_2 test function was 65345. This solution evaluates to a fitness of 8135 which is approximately 0.59% lesser than the globally optimal fitness value of 8183. It is to be noted that the best solution found by the proposed GA core lies within approximately 0.27% distance of the globally optimal solution in the solution space.
Table 4.7. Best fitness values obtained by the GA for the mBF6_2 function for different parameter settings (XR = Crossover Rate)

<table>
<thead>
<tr>
<th>RNG_Seed (hexadecimal)</th>
<th>PopSize=32</th>
<th>PopSize=64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XR=10</td>
<td>XR=12</td>
</tr>
<tr>
<td>2961</td>
<td>7999</td>
<td>7813</td>
</tr>
<tr>
<td>061F</td>
<td>6175</td>
<td>7578</td>
</tr>
<tr>
<td>B342</td>
<td>7612</td>
<td>7497</td>
</tr>
<tr>
<td>AAAA</td>
<td>7534</td>
<td>7534</td>
</tr>
<tr>
<td>A0A0</td>
<td>8104</td>
<td>7406</td>
</tr>
<tr>
<td>FFFF</td>
<td>7291</td>
<td>7623</td>
</tr>
</tbody>
</table>

Table 4.8. Best fitness values obtained by the GA for the mBF7 function for different parameter settings (XR = Crossover Rate)

<table>
<thead>
<tr>
<th>RNG_Seed (hexadecimal)</th>
<th>PopSize=32</th>
<th>PopSize=64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XR=10</td>
<td>XR=12</td>
</tr>
<tr>
<td>2961</td>
<td>56835</td>
<td>56835</td>
</tr>
<tr>
<td>061F</td>
<td>59648</td>
<td>53432</td>
</tr>
<tr>
<td>B342</td>
<td>55000</td>
<td>59928</td>
</tr>
<tr>
<td>AAAA</td>
<td>55560</td>
<td>52704</td>
</tr>
<tr>
<td>A0A0</td>
<td>58136</td>
<td>53040</td>
</tr>
<tr>
<td>FFFF</td>
<td>60880</td>
<td>61384</td>
</tr>
</tbody>
</table>

Table 4.8 tabulates the results obtained by the GA core for the mBF7_2 test function using different settings for the programmable GA parameters. The best candidate found by the proposed GA core for the mBF7_2 test function was 65516. The corresponding solution is $y=(\text{FF})_{16}$ and $x=(\text{EC})_{16}$ with a fitness of 61496. This is approximately 3.7% lesser than the globally optimal fitness value of 63904. The best solution found by the proposed GA core lies within 4.3% and 2.35% distance of the globally optimal solution along the x-direction and y-direction respectively of the solution space.

Table 4.9 tabulates the results obtained by the GA core for the mShubert2D test function using different settings for the programmable GA parameters. The proposed GA core found more than one globally optimal solution for many different parameter settings as seen in Table 4.9. The GA core found two different globally optimal solutions, $(x_1=C2, y_1=4A)$ and $(x_2=DB, y_2=4A)$, during the experimental run with RNG seed=$(\text{AAAA})_{16}$, population size=64, and crossover threshold=10.
Table 4.9. Best fitness values obtained by the GA for the Shubert function for different parameter settings (XR = Crossover Rate)

<table>
<thead>
<tr>
<th>RNG_Seed (hexadecimal)</th>
<th>PopSize=32</th>
<th></th>
<th>PopSize=64</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XR=10</td>
<td>XR=12</td>
<td>XR=10</td>
<td>XR=12</td>
</tr>
<tr>
<td>2961</td>
<td>56835</td>
<td>56835</td>
<td>48135</td>
<td>56835</td>
</tr>
<tr>
<td>061F</td>
<td>56835</td>
<td>55095</td>
<td>65535</td>
<td>58227</td>
</tr>
<tr>
<td>B342</td>
<td>56487</td>
<td>56487</td>
<td>54051</td>
<td>63795</td>
</tr>
<tr>
<td>AAAA</td>
<td>63795</td>
<td>56487</td>
<td>65535</td>
<td>65535</td>
</tr>
<tr>
<td>A0A0</td>
<td>56835</td>
<td>63795</td>
<td>65535</td>
<td>53355</td>
</tr>
<tr>
<td>FFFF</td>
<td>53355</td>
<td>65535</td>
<td>48135</td>
<td>56835</td>
</tr>
</tbody>
</table>

Figures 4.12 through 4.15 plot the data collected from the hardware runs and illustrate the convergence of the GA optimizer for the three test functions. Both the best fitness and average fitness values are plotted for every generation. It can be seen that the GA core finds the best solution within the first 20 generations for all three test functions. From Figures 4.12 and 4.13, it can be seen that the GA core evaluates at most \((10 \text{ generations} + 1\text{-initial population} = 11) \times \{\text{population size} = 64\}\) 704 candidate solutions before finding the best solution. Although the size of the entire solution space is only 65536, the GA core evaluates less than 1.1% of the solution space before finding the best solution. This is a major speedup over an exhaustive search and is very important for real-time applications and other applications that have time-consuming fitness evaluation procedures such as EHW.
Figure 4.12. Convergence plot for the test function mBF6_2(x) with initial RNG seed=(061F)$_{16}$, crossover threshold=10, and popSize=64 (data collected from hardware execution)

Figure 4.13. Convergence plot for the test function mBF6_2(x) with initial RNG seed=(A0A0)$_{16}$, crossover threshold=10, and popSize=64 (data collected from hardware execution)
Figure 4.14. Convergence plot for the test function mBF7_2(x,y) with initial RNG seed=(AAAA)_{16}, crossover threshold=12, and popSize=64 (data collected from hardware execution)

Figure 4.15. Convergence plot for the test function mShubert2D(x1,x2) with initial RNG seed=(AAAA)_{16}, crossover threshold=10, and popSize=64 (data collected from hardware execution)
From Figure 4.14, it can be seen that the GA core evaluates at most \((18\text{-generations} + 1\text{-initial population} = 19) \times \{\text{population size} = 64\}\) 1216 candidate solutions before finding the best solution for the \textit{mBF7_2} test function.

From Figure 4.15, it can be seen that the GA core evaluates at most \((12\text{-generations} + 1\text{-initial population} = 13) \times \{\text{population size} = 64\}\) 832 candidate solutions before finding the best solution for the \textit{mShubert2D} test function.

Thus, it can be seen that the GA core quickly converges to a good solution after evaluating a small fraction of the solution space even for difficult test functions. It is expected for the GA to find good solutions quickly due to its population-based search mechanism. The GA then converges towards the good solutions and tries to find better solutions in their vicinity. However, it has to be noted that the GA core finds the optimal solutions only for certain settings of the GA parameters. This re-iterates the necessity for the ability to change the values of these parameters according to the application at hand.

4.3.3 Runtime Comparison with Software Implementation

A software implementation of a GA optimizer, similar to the GA optimization algorithm in the IP core, was developed in the C programming language. Genetic algorithms, when used for hardware applications such as EHW, need to communicate with the application to evaluate the fitness of the candidate solutions. This communication overhead can be effectively modeled using the Xilinx Virtex2Pro board as it contains PowerPC processor IP cores that can execute software programs. The experimental setup consists of the GA software running on the PowerPC processor in the Xilinx Virtex2Pro board and the fitness evaluation module (implemented as the same lookup table using block RAM) on the Xilinx Virtex2Pro FPGA. This setup gives a fair comparison between the software and hardware implementations as both are implemented using the same technology node. The runtime, averaged over 6 runs, for the GA program for a
population size of 32, crossover rate of 0.625, mutation rate of 0.0625, and running for 32 generations to optimize the modified binary F6 ($mBF6_2$) function was 37.615 milliseconds.

In hardware, a 32-bit counter was implemented that is clocked using the 50MHz clock used for the GA IP core. The GA execution time on the hardware was computed as the product of the counter value and the clock period. The hardware GA implementation achieved a speedup of approximately 5.16x over the software implementation.

4.4 Summary

With the rapid increase in integration technology, entire systems are now implemented on a single chip. Many systems require a stochastic optimization engine and it is highly desirable that the optimization engine is also implemented in hardware. In this work, a readily synthesizable, robust genetic algorithm core for FPGAs has been designed that is easy to interface and use with a wide range of applications. The efficiency of the designed core is illustrated by the low area utilization and the high clock speed. The effectiveness of the designed GA core is evident from the convergence characteristics obtained for the standard test function. The gate-level Verilog implementation of the GA core is advantageous in that it can be directly used by commercial layout tools such as Cadence First Encounter for chip layout generation. The availability of preset modes and scan chain testability provides some basic fault tolerance to an ASIC designed using the proposed GA design.
CHAPTER 5
A DIGITAL FRAMEWORK FOR EVOLUTIONARY DESIGN, AUTONOMOUS
MONITORING AND PERFORMANCE COMPENSATION
OF EXTREME ENVIRONMENT ELECTRONICS

The functionality and performance of analog electronics degrades with temperature
variations and the presence of radiation in the operating environment. Extreme environment
electronics are necessary in many applications including automotive, geo-thermal, oil-well drilling,
and space applications. The degradation of operational characteristics such as amplitude gain or
slew rate of analog components can affect the operation of the entire system severely. Identifying
and compensating the performance degradation of these analog electronics is essential to guarantee
the proper functionality of the entire system. Moreover, many of these systems operate in
inaccessible environments. Hence, failure of these electronic components may have disastrous
consequences and lead to loss of the entire system.

A simple method to avoid performance degradation is to place the temperature sensitive
electronics in protective enclosures that provide a controlled operating environment regardless of
the surroundings. But such protective enclosures are bulky, consume a lot of power, and might fail
themselves.

An alternative method is to allow the performance degradation to occur but monitor the
performance characteristics and compensate when the performance degradation exceeds specified
limits. Without the volume and weight of the protective cover, or the power needed for thermal
control, electronics can be placed close to sensors and actuators in miniature probes, sensing
arrays, and smart structures. This offers a great increase in the electronic processing capability of the higher level systems, and results in unprecedented accessibility (information and control) of extreme environments encountered in human and robotic space exploration.

In this work, a digital system is proposed for autonomous monitoring and compensation of the performance and functionality of an analog system. The proposed system continuously monitors the performance characteristics of an analog system and compensates any significant performance degradation using either a model based compensation technique or a genetic algorithm based compensation technique. The proposed digital system has been tested at both the RT-level and the layout level. It has been implemented as a digital ASIC using Honeywell’s 0.35 μm rad-hard SOI-V technology. The novel contributions of the proposed digital system include:

- A digital framework for evolutionary design, autonomous monitoring, and compensation of analog electronics,
- Two methods of performance compensation – Genetic Algorithm (GA) based compensation and Model Lookup Table based compensation,
- Parameterizable hardware GA module for GA-based compensation,
- Support for external fitness function modules to provide scalability for an intrinsic EHW system,
- Module-level isolation and by-passing schemes that provide basic testability and fault tolerance, and
- Simple external interfaces to ensure ease-of-integration and ease-of-use.

5.1 Related Work

There have been previous design techniques for compensation of extreme environment effects, specifically extreme temperature effects. A survey of compensation design techniques for extreme temperature electronics has been presented in [73] where these techniques have been
classified into compensation design techniques and technological design techniques.

Most of the previous compensation schemes are specific to the target performance characteristic. In [74], current mirrors were proposed for leakage current compensation. In [75], input stage compensation schemes were proposed for operational amplifiers (op-amps). Supply voltage division schemes were proposed in [76] for compensation of middle gain stages in op-amps.

But all the compensation schemes proposed still face temperature limits beyond which their operation is degraded. Hence, an adaptive solution is necessary that can modify the necessary system parameters and recover any loss in performance. A simple adaptive solution may assume that the changes in the environmental conditions will cause a sub-set of a list of operational changes in the system and that the list of possible operational changes can be well-specified during the design phase. Different solutions can then be created for these specific conditions, all of which are available at runtime and applied as the conditions are changed. A compensation system can then switch between those predefined solutions. However, even if different temperatures are known in advance and the suitable design solutions created, the real-time switching of the design may not be trivial. Also, depending on the application in hand, it is not always possible to specify the temperature conditions at design time. To adapt to changing temperature conditions at runtime involves creating a new design solution tailored to the changing conditions. One possible runtime adaptive solution is to apply an evolutionary algorithm to search for the new solution.

Stoica [78] proposed Field Programmable Transistor Arrays, a hardware concept for reconfiguration at the transistor level. Both digital and analog circuits can be mapped on to the FPTA architecture. Performance recovery of circuits including multipliers, Gaussian-shape curve generators, filters and logic gates were successfully demonstrated [79][80]. In the initial experiments, the circuits were subject to specific high temperatures and compensation was manually triggered. The performance recovery algorithms were implemented in software and were
Further experiments by Zebulum et al [81] revealed that only filters with modest roll-off characteristics could be evolved or recovered by FPTAs. They proposed a Reconfigurable Analog Array (RAA) architecture that uses Gm-C filters to implement a wider range of analog filters even at extreme temperatures. The Gm-C filters are built using Wide Range Transconductance Amplifiers (WRTA) [82]. They tested the behavior of the WRTA for a temperature range of -180°C to +120°C. They also evolved a low-pass filter using the RAA and were able to recover its functionality for the same temperature range. The functionality recovery was done by manually tuning the bias voltages of the component WRTAs of the low-pass filter. Later, a hill-climbing algorithm was implemented on an FPGA to recover the performance of a low-pass filter using the RAA [83].

Zebulum et al [84] later proposed a Self-Reconfigurable Analog Array (SRAA) architecture that used multiple building blocks to build different analog circuits including Pulse Width Modulator, Power Switch Control, Shaft Encoder, and Instrumentation Amplifier. They identified a list of essential building blocks such as Op-Amps, Low offset Op-Amps, High Voltage Op-Amp, Current Source, Comparator, and High-speed Comparator by analyzing previous implementations of the analog circuits. The building block analog cells were replicated in four rows to form a 4 x 6 array of functional analog cells. An extra copy of each type of the analog cell is also provided to serve as a reference analog cell. This enables online monitoring and compensation of the reference analog cell while the functional analog cells continue their operation. The functionality of the analog cells can be programmed by setting their bias voltages using configuration Digital-Analog Converters (DACs). A switchbox array is used to interconnect the different analog cells and to provide external access to the analog cells. The compensated bias voltage values are applied to the analog cells using the bias voltage configuration DACs of the analog cells.
In all the previous efforts, user interaction is needed for one or more of the following:

- to evolve the required analog circuit using the field programmable analog chips,
- to monitor the performance of the evolved analog circuits and identify any significant performance degradation,
- to recover the performance or functionality of the evolved circuit under changing conditions.

In this work, a digital framework is proposed that interacts with a field programmable analog array to evolve custom analog circuits, autonomously monitor their performance and automatically compensate any performance degradation using either a model based lookup technique or a genetic algorithm based technique. Thus, the proposed system removes the need for any user interaction with the entire system. The monitoring and compensation system has been implemented as a digital ASIC so that the form factor of the entire system is reduced.

5.2 Architecture and Implementation of Proposed Digital Framework

This section presents a detailed discussion of the design methodology and behavioral modeling of the digital framework. It also describes its various features and interfacing details.

5.2.1 Overall System Level Architecture and Operation

This section will describe the overall architecture and operation of an entire system that contains the proposed digital system as the performance monitoring and compensation subsystem.
Figure 5.1. Block diagram of overall system (including analog and digital systems)

The architecture of the top-level system containing both the analog electronics and the digital monitoring and compensation system is shown in Figure 5.1. The analog system shown in the figure is a generalization of the reconfigurable analog array architectures proposed by Stoica et al [83] and Zebulum et al [84]. The analog sub-system consists of:

- Analog electronic components/cells, arranged in an array fashion,
- Switching and configuration logic to interconnect the various components in the analog sub-system and also to realize different functionalities, and
- Digital-Analog and Analog-Digital converters to interface with the digital sub-system so that the performance characteristics of the analog cells can be measured.

The digital sub-system is meant to monitor the performance of the analog cells in the analog sub-system and compensate any significant degradation using the user-specified compensation technique. It comprises of:
• **Monitoring logic** – to select the analog cells, excite them with appropriate input signals, collect their response, and calculate the performance degradation, if any,

• **Compensation logic** – to compensate the degraded performance of the selected analog cell using the user-selected technique,

• **Synchronization and Control logic** – regulates the communication between modules and controls the triggering of the monitor and compensate cycles, and

• **Interfacing logic** – contains the communication interface between the analog sub-system and the digital sub-system.

Figure 5.2 shows the high-level operational flow of the entire system. The operation of the proposed system can be divided into three operational modes:

• **INIT mode** – In the INIT mode, the entire system is reset and all the customizable parameters in the system are programmed.

• **MONITOR mode** – In this mode, an analog cell is selected by providing its address to the analog sub-system. The appropriate analog performance characteristic is measured by sending an excitation signal to the selected analog cell, recording the response, and evaluating the digitized response on the digital sub-system.

• **COMPENSATE mode** – The proposed system enters this mode when the digital sub-system has identified that the currently selected analog cell suffers from significant performance degradation. The digital sub-system uses either the model based lookup technique or the GA-based technique for compensation based on the user specification.
5.2.2 Detailed Architecture of the Proposed Framework

The proposed system has been designed with a modular architecture as shown in Figure 5.3 to enable ease-of-design, and module-level isolation and testing. The proposed system consists of the following modules:

- Initialization Module
- Digital Controller Module
- System Monitor Module
- GA-based Compensation Module
- Internal fitness evaluation module, and
- Model based lookup table (LUT) module.

The following section describes in detail the design of each of these modules and the operation of the entire proposed system.
Initialization Module – The initialization module contains logic to initialize all the modules in the digital system. When the user asserts the “init_digASIC” signal of the initialization module, all the other modules of the digital sub-system are reset. The initialization module can then be used to:

- Program the number of Analog cells to be monitored by the System Monitor module.
- Load pre-characterized correction voltage values into the model based lookup table module.
- Customize the programmable parameters of the GA optimization engine in the GA-based compensation module.
- Customize the programmable parameters of the internal fitness evaluation module.
- **Digital Controller Module** – The Digital Controller module acts as the main controlling unit that synchronizes and triggers the operations of all the other modules in the proposed system. It is a simple state machine, as shown in Figure 5.4, which loops the entire digital system in monitoring and compensation cycles. On system reset, the digital ASIC controller module enters an IDLE state. In the IDLE state, the digital ASIC controller waits for the analog cell array to start some operation. Once the analog cells are programmed to realize some required functionality, the “app_valid” signal is asserted. This triggers the controller module to start the monitor-and-compensate cycles for the analog cells. In the START_MONITOR state, the controller triggers the system monitor...
module to commence the monitoring cycle by asserting the “start_monitoring” signal. After triggering the system monitor module, the controller waits in the IDLE2 state for a compensation request. When it receives a compensation request, the controller triggers the LUT-based or GA-based compensation technique based on the user programmed “comp_type” signal. In the LOAD_BEST_CANDIDATE state, the controller asserts the “prog_FAC” signal to indicate that the analog system should apply the bias voltages found in the “candidate” bus to the selected analog cell. The controller then waits in the COMPENSATION_DONE state for the analog system to complete applying the bias voltages to the selected analog cell. Then the controller returns to the IDLE2 state while the system monitor module proceeds to the next analog cell to continue its monitoring cycle.

- **System Monitor Module** – The System Monitor (SM) module interfaces with the analog electronics subsystem and determines if the functionality and performance of the system is acceptable. The System Monitor module has also been designed as a finite state machine as shown in Figure 5.5. On system reset, the system monitor module goes into the INIT state and waits for a monitoring request. When the “start_monitoring” signal is asserted, the system monitor module starts monitoring the analog cells for performance degradation. In the MONITOR_MODE state, the system monitor signals to the entire system that the digital system is entering the monitoring cycle by asserting the “monitor mode” signal. It also selects the first analog cell by setting the “RAC_ID” address bus to 0 and asserting the “RAC_ID_valid” signal in the VALIDATE_RAC_ID state. In the WAIT_FOR_FITNESS state, the system monitor module waits for the fitness evaluation module to compute the performance of the selected analog cell. When the “fit_valid” signal is asserted, the system monitor reads the “fit” signal and then goes to the TEST_FITNESS state. If the “fit” signal
is de-asserted, it implies that the selected analog cell suffers from significant performance degradation and needs compensation. The system monitor notifies the controller module about the need for compensation in the COMPENSATE state by asserting the “compensate” signal and signals the entire system that the system is in the compensation phase by de-asserting the “monitor_mode” signal. The system monitor then waits for the compensation phase to complete in the WAIT_FOR_COMP_DONE state. When the “compensation_done” signal is asserted, the system monitor enters the monitoring phase in the MONITOR_NEXT_RAC state by asserting the “monitor_mode” signal and de-asserting the “compensate” signal. If the “fit” signal is asserted, it implies that the selected analog cell has an acceptable performance and does not need compensation. If compensation is not required for the current analog cell, the system monitor module proceeds to the next analog cell in the MONITOR_NEXT_RAC state and continues the monitoring cycle.
• **GA module** – The proposed system has a Genetic Algorithm (GA) module to perform GA-based compensation. The GA module consists of three sub-modules, namely, the GA optimizer core, the GA memory, and the random number generator (RNG).
Figure 5.6. Chromosome encoding used by the GA based compensation algorithm

The GA optimizer in the GA module is built using the customizable GA core described in [81] which follows the optimization cycle shown in Figure 4.1. A 16-bit binary string is used to encode the chromosome used by the GA optimizer. This binary string is the concatenation of two voltage values that serve as the bias voltages for the selected analog cell as shown in Figure 5.6. In Zebulum et al [84], these digital values are applied to the configuration DACs of the selected analog cells to obtain the required functionality. A 16-bit chromosome was chosen as Zebulum et al [84] were able to realize all their desired analog building block functionalities by programming the analog cells using two 8-bit bias voltages.

In each generation, a new population of candidate solutions is generated using crossover and mutation operators. The GA optimizer follows an elitist strategy by preserving the best individual found in each generation, since Rudolph [8,9] proved that elitist GAs have better probability of convergence to the global optimum. The GA module uses the Proportionate Selection scheme to select parents for the crossover operation. This ensures that highly fit individuals have a selection probability that is proportional to their fitness. The GA core implements the single-point binary crossover technique [3]
illustrated in Figure 4.2 to combine parents from the current generation and produce offspring for the next generation. The GA core performs crossover only if the random number generated is less than the specified 4-bit crossover threshold. Since the crossover threshold is user-programmable, the user can control the crossover rate in the GA core. The GA core uses random bit flipping as its mutation operator and uses a 4-bit mutation threshold. The RNG module implements a cellular automaton based random number generator as described in [41]. The initial seed of the RNG module can be provided by the user. The GA memory module is a single port memory module that can store both the individuals and the fitnesses of the individuals in the population.

On system reset, the GA module waits in the IDLE state for either an initialization request or a GA-based compensation request. When the “load_GA” signal is received, the GA module enters an initialization mode in which the customizable GA parameters such as population size, crossover rate, mutation rate, number of generations and random number generator seed can be programmed. When the “start_GA” signal is asserted, the GA module enters the GA optimization cycle. The GA module uses either the on-board or the external fitness evaluation module to compute the fitness of the individuals that it generates during the optimization cycle. When the GA optimization cycle is done, the GA module outputs the best solution found on the “candidate” bus and asserts the “GA_done” signal. The GA module de-asserts the “GA_done” signal when the “start_GA” signal is de-asserted which indicates that the best candidate has been read.

The GA core uses a simple two-way handshaking protocol for its communication with the fitness evaluation module. When the GA core requires the fitness of a candidate solution, it places the individual on the candidate bus and then asserts the fit_request signal. The FEM module of the target application should then read the candidate port and compute the fitness of the individual. The computed fitness is then placed on the fit_value bus.
port of the GA core and the fit_valid signal is asserted by the target application. On assertion of the fit_valid signal, the GA core reads the fitness value and de-asserts the fit_request signal. The simplicity of the interfacing protocols is a major advantage as there are no real timing issues because of the two-way handshaking protocols used for communication.

- **Fitness Evaluation Module** – The proposed digital system uses a fitness evaluation module to calculate the value of performance characteristics for the analog cells and determine if there exists significant performance degradation. The proposed system supports both internal and external fitness evaluation modules, which improves the scalability of the system. The internal fitness evaluation module implemented in the proposed system computes the slew rate of an analog cell. It compares the computed slew rate to a user-programmed threshold value to determine if the analog cell needs compensation with respect to the slew rate characteristic.

The internal fitness evaluation module has a modular architecture and consists of three modules:

- FEM_Controller module,
- Fitness Computation module, and
- Excitation module.

The FEM_Controller module contains the interfacing logic to communicate with the analog sub-system and the other modules in the digital sub-system. It has been implemented as a simple state machine as shown in Figure 5.9. On system reset, the FEM_Controller module enters the INIT state and then waits in the IDLE state for a fitness evaluation request from either the system monitor module or the GA module. When either “fit_req_mon” signal or “fit_req_ga” signal is asserted the FEM_Controller module checks for assertion of the “onchip_fiteval” signal to determine if the fitness
evaluation is to be done by the internal fitness evaluation module. It then reads the address of the analog cell in the READ_ANALOG_CELL_ID state and triggers the Excitation module and the slew rate module to evaluate the fitness of the selected analog cell. When the evaluation is complete, the FEM_Controller module asserts the “fit_valid” signal to indicate to the fitness request module that a valid fitness value is available. It then waits for the fitness value to be read in the WAIT_FOR_FIT_READ state. When the “fit_req” signal is de-asserted, the controller returns to the IDLE state and waits for the next fitness evaluation request.

![Excitation Input Diagram](image)

![Analog Cell Response Diagram](image)

Figure 5.7. Slew rate measurement by the internal fitness evaluation module of the proposed digital framework

The Excitation module is used to send an appropriate input signal to the selected analog cell so that its response can be collected and used to obtain a value for the desired performance characteristic. The excitation module within the internal fitness evaluation module is used to send a step-input signal to the selected analog cell. It is implemented as a state machine that can be programmed to send out either a rising edge or a falling edge. The start and end voltage levels and the duration of these voltage levels can also be
programmed by the user/application. Figure 5.8 shows a rising edge excitation input being sent to the analog cell and the response expected from the analog cell.

![Figure 5.8](image)

Figure 5.8. FSM of the FEM_Controller module in the internal fitness evaluation module

The Fitness Computation module within the internal fitness evaluation module is used to compute the output slew rate of the selected analog cell. The output slew rate is a measure of the rate of change of the analog cell’s output. The time taken for the output signal to change from 10% of its maximum value to 90% of its maximum value is used as
a measure of the analog cell’s output slew rate. The response of the analog cell to the rising-edge input signal from the excitation module passes through an Analog-Digital converter (ADC) and the digitized response is read by the Fitness module. The fitness module decrements the maximum value of an internal counter to measure the time elapsed between the ADC code for the 10% voltage value and the 90% voltage value. The remaining counter value is used as a fitness measure for the slew rate characteristic of the selected analog cell.

- **Model Based Lookup Table Module** – The model based lookup table module is essentially a memory module that stores the best voltage values for all the analog cells at different temperatures. This lookup table is addressed by concatenating the analog cell address and the current temperature value. The bias voltage values that produce the best performance for the analog cells at different temperatures can be obtained by pre-characterization experiments. Such pre-characterization experiments can build functional models for the analog cells and find the bias voltage values that obtain the best performance for different temperature settings. These voltage values can then be loaded into the lookup table during the initialization phase of the proposed system and read from the module during model based compensation. The lookup table in the proposed system stores the best voltage values for eight different analog cells over a temperature range of -180°C to +120°C in 5°C temperature increments.

### 5.3 Digital ASIC Implementation

In this section, we will discuss the design methodology and tools used to develop the proposed system as a digital ASIC.
5.3.1 Design Methodology

The design flow used to develop the digital ASIC from the behavioral models is shown in Figure 5.10. The behavior of each of the modules in the digital system was implemented in behavioral VHDL and simulated to test the correct functionality at the behavioral level. An RT-level VHDL description of each module was then obtained from the behavioral VHDL description using an in-house High Level Synthesis tool called Automatic Design Instantiation (AUDI) tool. The RT-level description is built using a set of simple components. The RT-level VHDL
description was simulated thoroughly using Cadence NC-Launch tools to test the functionality of the GA core. A gate-level Verilog description was generated for each of the modules from the RT-level VHDL description using in-house flattening scripts and the Berkeley SIS tool [66]. The gate-level Verilog description uses simple gates such as NAND, NOR, AND, OR, XOR, and SCAN_REGISTER. The gate-level Verilog description was also simulated using Cadence NC-Launch to verify the functionality and the timing details of all the modules. The functionalities of the RT-level and gate-level systems were also tested by developing and testing a top-level design. This design methodology ensures that the entire system is available to the user at two design levels, namely RT-level and Gate-level.

The gate-level netlists of each of the modules were used to produce module layouts using the Cadence Silicon Ensemble Standard Cell Place-and-Route tool. The top-level ASIC layout was obtained by performing block placement of all the modules using the Cadence First Encounter tool. The placed and routed layout was then imported into the Cadence Virtuoso tool and various design checks such as DRC, ERC, Antenna-check, Soft Power check and LVS were performed on the layout using the Mentor Graphics Calibre tool. The digital ASIC passed all the tests and was fabricated using a commercial SOI-based rad-hard technology.

5.4 Simulations and Results

The digital ASIC was extensively tested at multiple design levels, namely, behavioral level, RT-level, gate level, and layout level. The functionality of the individual modules was tested at the behavioral level using Cadence NC-launch simulations. The RTL descriptions of the individual modules and the top-level design were simulated for correctness using Cadence NC-launch tools. Finally, the gate level Verilog descriptions of the individual modules and top-level design were simulated and verified using ModelSim. The gate level netlists of the individual modules were then used to produce the module layouts using Cadence Silicon Ensemble. The
individual layouts were then placed and routed to form the entire ASIC layout using Cadence SOC Encounter tools.

5.4.1 Layout-level Simulations

Spice netlists were extracted for each of the individual modules and the top-level design. These netlists were then simulated using HSpice for simple test cases as complete tests are exorbitantly time-consuming. Simulation snapshots for the individual modules are shown in Figure 5.10 through Figure 5.13.

Figure 5.10. Layout level simulation of the digital framework controller module
Figure 5.11. Layout-level simulation of the system monitor module (in a monitoring cycle)

Figure 5.12. Layout level simulation of the system monitor module (in a compensation cycle)
Figure 5.13. Layout level simulation of the internal fitness evaluation module (containing FEM_Controller, excitation, and slew rate modules)

5.5 Summary

Evolvable analog electronics are useful in many space applications. In this work, a digital framework is proposed for the realization of self reconfigurable electronics that performs autonomous monitoring and compensation of analog electronics operating under extreme environments. The proposed system has been implemented as a digital ASIC to reduce the form factor of the overall system and is also scalable with respect to the number of fitness functions.
CHAPTER 6
CONCLUSIONS

The complexity of designing hardware systems has increased significantly with technological advances. Irrespective of the choice of hardware implementation, technology scaling has affected the design process by increasing the complexity of the designs that can be implemented on a single chip, and by introducing new challenges in the form of interconnect, thermal, and reliability issues. Effective optimization techniques are required in all the stages of the hardware design cycle. Genetic algorithms have been shown to be a robust search mechanism in a wide variety of problem domains. In this dissertation, it has been shown that genetic algorithms can be used successfully to address the optimization needs of the hardware design process. Specifically, this dissertation has used genetic algorithm based optimizers to address the following hardware design problems:

- **Layout optimization of VLSI ASICs** – A genetic algorithm based multi-objective floorplanner has been developed for solving the outline-free macro-cell based ASIC design problem. The proposed floorplanner outperforms all existing floorplanners that perform simultaneous optimization of floorplan area and wirelength.

- **Reconfigurable Hardware Design of Optimization Applications** – A customizable FPGA IP core of a general purpose genetic algorithm has been developed to alleviate the design of hardware applications that need an effective optimization engine.

- **Design, Monitoring and Performance Compensation of Extreme Environment Electronics** – A digital framework has been developed for the evolutionary design of analog
electronics. The proposed digital framework also performs autonomous monitoring and automatic performance compensation of the evolved analog electronics when operating in extreme environments.
REFERENCES


ABOUT THE AUTHOR

Pradeep Ruben Fernando received a Bachelor of Engineering Degree in Computer Science and Engineering from the University of Madras in 2002 and a Master of Science Degree in Computer Engineering from the University of South Florida in 2006. He was admitted into the Ph.D. program at the University of South Florida in the summer of 2005. While in the Ph.D. program, Pradeep has co-authored several technical publications and presented papers at technical conferences.