A New Architecture for Adaptive Digital Logic

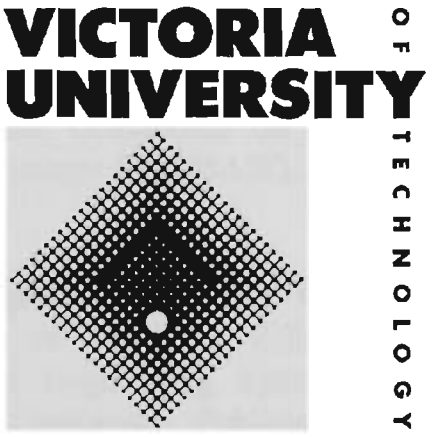
Mehrdad Salami, M.Sc.

A thesis submitted for the degree of

Doctor of Philosophy

in the

Department of Electrical and Electronic Engineering
Faculty of Engineering



1996

Preface

This work is conducted under guidance of Dr. Greg Cain as supervisor. Some of the research results presented here were included in the following papers [Salami M. and Cain G.], published in conference proceedings or currently under review for journal publication:

[1] "The Quest for a New Computing Architecture Based on Genetic Algorithms", Proceedings of the Electrical Engineering Congress (EEC94), The Institution of Engineers Australia, Canberra, Australia, November 1994, pp. 635-640.

[2] "An Adaptive Control System Based on Genetic Algorithms", Proceedings of the First International Workshop on Intelligent Adaptive System (IAS-95), Melbourne Beach, Florida, April 1995, pp. 63-77.

[3] "A Genetic Algorithm Processor", Proceedings of the Iranian Conference on Electrical Engineering (ICEE95), Iran University of Science and Technology, Tehran, Iran, May 1995, pp. 233-239.

[4] "Adaptive Hardware Optimization Based on Genetic Algorithms", Proceedings of The Eighth International Conference on Industrial Application of Artificial Intelligence & Expert Systems (IEA95AIE), Melbourne, Australia, June 1995, pp. 363-371.

[5] "Multiple Genetic Algorithm Processor for the Economic Power Dispatch Problem", Proceedings of The First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'95), IEE Conference Publication No. 414, The University of Sheffield, Sheffield, UK, September 1995, pp. 188-193.

[6] "An Adaptive PID Controller Based on Genetic Algorithm Processor", Proceedings of The First IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications

(GALESIA'95), IEE Conference Publication No. 414, The University of Sheffield, Sheffield, UK, September 1995, pp. 88-93.

[7] "A Multiple Genetic Algorithm Processor for a PID Controller System", Proceedings of The International Conference on Genetic Algorithms 95 (MENDEL'95), University of Brno, Brno, Czech Republic, September 1995, pp. 67-71.

[8] "Genetic Algorithms for Solving the Economic Power Dispatch Problem", Proceedings of The Electrical Engineering Conference 1995 (EEcon95), The Institution of Engineers Australia, Adelaide, Australia, September 1995, pp. 59-64.

[9] "A PID Controller Based on a Multiple Genetic Algorithm Processor", Proceedings of Control 95 Conference (Control'95), The Institution of Engineers Australia, University of Melbourne, Melbourne, Australia, October 1995, pp. 359-362.

[10] "Multiple Genetic Algorithms Processor for Engineering Applications", Poster Proceedings of The Eighth Australian Joint Conference on Artificial Intelligence (AI'95), The University of New South Wales, Canberra, Australia, November 1995, pp. 79-86.

[11] "Implementation of Genetic Algorithms on Reprogrammable Architectures", Applications Stream Proceedings of The Eighth Australian Joint Conference on Artificial Intelligence (AI'95), The University of New South Wales, Canberra, Australia, November 1995, pp. 121-128.

[12] "Application of Multiple Genetic Algorithm Processor in Complex Systems", Proceedings of The Second New Zealand International Conference on Artificial Neural Network and Expert System (ANNES'95), IEEE Computer Society Publication, University of Otago, Dunedin, New Zealand, November 1995.

[13] "Genetic Algorithm Processor for Adaptive IIR Filters", Proceedings of The Second IEEE International Conference on Evolutionary Computing (ICEC'95), The University of Western Australia, Perth, Australia, December 1995, pp. 423-428.

[14] "A Robust Genetic Algorithm", Proceedings of The First Annual CSI (Computer Society of Iran) Computer Conference (CSICC'95), Sharif University of Technology, Tehran, Iran, December 1995, pp. 542-548.

[15] "Genetic Algorithms Processor for Adaptive Engineering Systems", Proceedings of The First International Conference on Fuzzy Logic and the Management of Complexity 1996 (FLAMOC'96), The University of Sydney, Sydney, Australia, pp. 265-269.

[16] "Genetic Algorithms Toolbox for Matlab", Proceedings of The 1996 Australian MATLAB Conference, CEANET Inc., The University of Melbourne, Melbourne, Australia, January 1996, pp. 1-4.

[17] "Genetic Algorithm Processor on Reprogrammable Architectures", Proceedings of The Fifth Annual Conference on Evolutionary Programming 1996 (EP96), MIT Press, San Diego, CA, March 1996.

[18] "Application of Genetic Algorithm Processor in a PID Controller System", to appear in the Proceedings of The Fourth Iranian Conference on Electrical Engineering (ICEE96), The University of Tehran, Iran, May 1996.

[19] "Genetic Algorithm Processor for the Frequency Assignment Problem", to appear in the Proceedings of The Ninth International Conference on Industrial Application of Artificial Intelligence & Expert Systems (IEA96AIE), Fukuoka Institute of Technology, Fukuoka, Japan, June 1996.

[20] "Hardware Implementation of Genetic Algorithms", paper submitted to the Journal of Evolutionary Computing, Reference number EC-KD-9602-0160 January 1996.

[21] "Application of Genetic Algorithm Processor in Engineering", paper submitted to the IEEE Transactions on Industrial Electronics Magazine, February 1996.

[22] "Hardware Genetic Algorithms and Their Applications", to appear in the Proceedings of IEEE International Conference on Industrial Technology, Shanghai, China, 2-6 December 1996.

Research related activities

1 Member of the program committee in The International Conference on Genetic Algorithms 95 (MENDEL'95), University of Brno, Brno, Czech Republic, September 1995.

2 Member of the program committee in The International Conference on Genetic Algorithms 96 (MENDEL'96), University of Brno, Brno, Czech Republic, June 1996.

3 Invited paper for the Special Session on Genetic Algorithms Applications in the IEEE International Conference on Industrial Technology, Shanghai, China, December 1996.

4 Paper review for IEEE Transactions on Industrial Electronics:
Paper Number: 1728 Review A,
Authors: Park J.H. and Choi Y.K.,
Title: "An On-line Control Scheme with Evolution Strategy for Unknown Nonlinear Dynamic Systems",
Date sent: February 7, 1996.

Declaration

I hereby declare that this thesis is the result of my own research and has not been submitted for a degree to any other university.

Mehrdad Salami
Department of Electrical and Electronic Engineering
Faculty of Engineering
Victoria University of Technology
Melbourne, Australia

Abstract

This thesis reports research into the hardware implementation of Genetic Algorithms and engineering applications. These algorithms are significant to engineering as a means of providing additional adaptive capability to known and existing control mechanisms.

The first part of the thesis is concerned with the underlying mechanisms of Genetic Algorithms and a model of a computing architecture which directly executes these algorithms in a generic form. The model has been developed and simulated using the hardware descriptive language VHDL and the Mentor Graphics tools running on SUN systems. It has been tested using a standard software test suite and synthesised into Field Programmable Gate Array (FPGA) technology. Test results demonstrate the performance of the Genetic Algorithm Processor (GAP) on a number of standard problems and the speedup achievable in comparison with software Genetic Algorithms.

The second part of the thesis is concerned with the applications of the GAP to engineering problems including economic power dispatch, PID controllers and adaptive digital filters. A new hardware configuration based on multiple units of the original design is introduced as a means of handling applications where long bit strings are required. Finally, the ability of the GAP to adapt an existing controller or filter to a dynamically changing environment is investigated.

Acknowledgments

I would like to thank my supervisor Dr. Greg Cain for his constant support and guidance throughout my thesis work. For his enthusiasm to basic research, willingness to discuss and challenge ideas at any time as well as his kindness and generosity, I feel indebted to him and find it hard to imagine a better supervisor.

I also thank my co-supervisor Dr. Roman Malyniak for his great help on several occasions. I also would like to thank John Chlond for his support in software and hardware design and for providing the necessary tools.

I would like to thank my following colleagues for their support and help during research and writing: Adrian Stoica, Reza Berangi, Omar Ghanayem, Mahmood Zonoozi, Nasser Hossainzadeh and Dr. Osama Ata.

I gratefully acknowledge the financial support from the Ministry of Culture and Higher Education (MCHE) in Iran throughout the research period.

Finally, I thank my parents for their support and great understanding. Above all I am grateful to my wife, for her continuous support, without which this work would not have been possible. I dedicate this thesis to her.

Contents

Appendices

List of Figures

List of Tables

List of Abbreviations

The following definitions of the abbreviation are applied throughout this thesis.

Abbreviation	Description	Defined in Section
ADC	Analog to Digital Converter	4.2.3.9
ASIC	Application Specific IC	2.3
CA	Cellular Automata	4.2.3.1
CLB	Configurable Logic Block	6.2.3
CM	Crossover Module	4.2.3
CMOS	Complementary Metal-Oxide Semiconductors	6.1.1
CUPL	Compiler for Universal Programmable Logic	1.4
DAC	Digital to Analog Converter	4.2.3.9
DAP	Distributed Array Processor	2.6
DPE	Dynamic Parameter Encoding	3.5.5
EEPROM	Electrically EPROM	1.3
EPD	Economic Power Dispatch	2.2
EPROM	Erasable Programmable ROM	1.3
FIR	Finite Impulse Response	2.2
FM	Fitness Module	4.2.3
FPGA	Field Programmable Gate Array	1
FPID	Filed Programmable Interconnect Device	2.4
FU	Fitness Unit	4.2

GAP	Genetic Algorithm Processor	1
GAs	Genetic Algorithms	1
IIR	Infinite Impulse Response	2.2
IOB	Input Output Block	App. D.1
LCA	Logic Cell Array	6.2.3
LFSR	Linear Feedback Shift Register	4.2.3.1
LMS	Least Mean Squares	7.3.1
MIM	Memory Interface Module	4.2.3
MIMD	Multiple Instruction Multiple Data	2.6
MM	Mutation Module	4.2.3
MPGA	Mask-Programmed Gate Array	1.3
MSE	Mean Square Error	7.3
MSI	Medium-Scale Integration	App. D.2
MU	Memory Unit	4.2
NP	Non Polynomial	1
nP	nano-Processor	2.4
PAM	Programmable Active Memory	2.4
PID	Proportional Integrator Differentiator	2.2
PLA	Programmable Logic Array	6.2.3
PLD	Programmable Logic Device	1.4
PNG	Pseudorandom Number Generator	4.2.3
RAM	Random Access Memory	1.3
RM	Read Module	4.2.3
ROM	Read Only Memory	1.3
RTL	Register Transfer Language	1.4

sGA	simple Genetic Algorithm	4.2.3
SGA	Software Genetic Algorithms	6.3.2
SIMD	Single Instruction Multiple Data	2.6
SLA	Stochastic Learning Automata	7.3.1
SM	Selection Module	4.2.3
SRAM	Static RAM	2.4
SSI	Small-Scale Integration	App. D.2
SU	Setup Unit	4.2
SUS	Stochastic Universal Sampling	3.3
TTL	Transistor-Transistor Logic	App. D.2
UUT	Unit Under Test	4.2.3.9
VHDL	VHSIC Hardware Description Language	1.2
VHSIC	Very High Speed IC	1.2
VLSI	Very Large Scale IC	1.1

List of Symbols

The following definitions of the symbol are applied throughout this thesis.

Symbol	Description	Defined in Section
addressm	Width of address line for Memory Unit	4.3
B_i	The transmission loss coefficient	7.2.1
C	Crowding factor	3.4
$C_s(g)$	Number of copies of schema *s* in the population at generation g	3.5.2
d	The number of delay cycles in the Fitness Unit	5.2
d(n)	Desired response for the plant in IIR filter	7.3.1
d_s	Defining length of schema *s*	3.5.2
e(n)	Prediction error in IIR filter	7.3.1
E(S)	Error signal function in PID controller	7.1.1
F_i	The flow rate of stage *i*	5.2
F_{out}	The flow rate out of the pipeline	5.2
$\overline{f}$	Average fitness of the whole population	3.5.2
f_s	Fitness of schema *s*	3.5.2
$F_T(P)$	The total fuel cost for all generators	7.2.1
g	Number of generations	3.5.2
geneparam	Package file name for the GAP model	4.3
h(n)	The impulse response of IIR filter	7.3.1

$H_A(z)$	Rational transfer function for IIR filter	7.3.1
$H_D(z)$	Desired response function for the plant in IIR filter	7.3.1
k	Number of GA parameters in GAP memory	4.2.3.2
K_d	Derivative term in PID controller	7.1.1
K_i	Integration term in PID controller	7.1.1
K_p	Proportional term in PID controller	7.1.1
l	Length of schema	3.5.2
l_s	Maximum useful schema length	3.5.2
m	Population size	3.5.2
M	Nominator degree of IIR filter	7.3.1
N	Denominator degree of IIR filter	7.3.1
$ngen$	Maximum number of generations	4.3
n_s	Number of usefully processed schemata	3.5.2
o_S	Order of schema s	3.5.2
$p(s)$	Probability that the schema s survives the reproduction operators	3.5.2
$P(S)$	Plant transfer function in PID controller	7.1.1
P	Problem space dimension	1
P0	Population at bottom of memory	4.2.3.3
P1	Population at top of memory	4.2.3.3
p_c	Probability of crossover	3.5.2
P_i	The power generated by the unit i	7.2.1
$P_{i_{max}}$	The upper limits of permitted power generation for unit i	7.2.1

$P_{i_{\min}}$	The lower limits of permitted power generation for unit i	7.2.1
P_L	The total transmission loss	7.2.1
p_m	Probability of mutation	3.5.2
P_R	The total load demand	7.2.1
P_t	Population at generation t	4.2.3.3
P_T	The total power generation	7.2.1
r	The total number of cycles to read from the memory	5.2
R	Clock cycle rate per generation per population member	5.2.6
$R(S)$	Reference signal in PID controller	7.1.1
R_a	Actual clock cycle rate per generation per population member	5.2.7
randomsize	Size of maximum random number	4.3
s_i	The actual service time of pipeline stage i	5.2
S_{norm_i}	The normalised service time of stage i	5.2
T	The total number of clock cycles	5.2
$U(S)$	Output of the PID controller	7.1.1
$v(n)$	Nominator factors for IIR filter	7.3.1
$V(n)$	Additive noise	7.3.1
valuem	Width of data line for Memory Unit	4.3
w	The total number of cycles to write to the memory	5.2
$w(n)$	Denominator factors for IIR filter	7.3.1
$x(n)$	Input to IIR filter	7.3.1
$y(n)$	Output of IIR filter	7.3.1

Y(S)	Output of the plant in PID controller	7.1.1

Chapter 1

Introduction

In recent times some new classes of probabilistic search algorithms have been developed to handle difficult or intractable problems. Principal among these developments are Genetic Algorithms [Holland, 1992; Goldberg, 1989b] and the method of Simulated Annealing [Kirkpatrick et al., 1983]. The search method used in Genetic Algorithms (GAs) mimics some of the mechanisms and principles of natural evolution in biological systems. They have exhibited an almost unique ability to solve difficult problems in discrete configuration spaces where solutions are found as unconnected points in a P-dimensional space rather than as points on differentiable surfaces. Examples include adaptive game-playing, biological cell simulation, machine learning, pattern recognition, VLSI microchip layout and job scheduling. These discrete configuration problems usually have an enormous set of candidate solutions that expands in a non-polynomial fashion with the problem dimension (P). Such problems

are thus known as *NP Hard* and algorithms which seek exact solutions are generally only used when P is small.

GAs handle large configuration spaces by sampling and processing randomly selected points in the space. The nature of GAs and their wide applicability make them excellent candidates for hardware implementations, thus obtaining a great speedup over software implementations. This speedup would allow hardware GAs to be applied to much more complex problems.

Because a general-purpose GA engine requires certain parts of its design to be easily changed (e.g. the operators), a Genetic Algorithm Processor (GAP) was not feasible until Field-Programmable Gate Arrays (FPGAs) were developed. FPGAs allow for reprogrammability which is an essential concept behind the development of the GAP model.

This thesis describes the GAP, an implementation of a hardware genetic algorithm. Because of the reprogrammability of FPGAs, the GAP is a general purpose GA engine which is useful in many applications where conventional GA implementations are too slow and expensive. The GAP works as a optimiser with the system under test and gives its user the ability to specify many of the GA parameters.

1.1 Introduction to Genetic Algorithms

The Darwinian theory of evolution depicts biological systems as the product of ongoing process of natural selection. During the 1950s researchers became interested in genetic processes and the possibility of emulating them in computer systems. The foundations of Genetic Algorithms theory were initially developed by John Holland and his students and in recent years have been applied to problems as diverse as pattern recognition and optimisation. GAs are probabilistic algorithms and their behaviour is still in many ways not well understood. It can be said that genetic algorithms are probabilistic algorithms

which start with an initial population of likely problem solutions, and then evolve towards better solution version. New solutions are generated with the use of genetic operators patterned upon the reproductive processes observed in nature. Also from the area of genetics come the names of the concepts we use. Each element of a current solution space (population) is called a chromosome, and its components are called genes. Genetic operators also have names originating in genetics: cross-over, mutation and inversion. Genetic algorithms allow engineers to use a computer to evolve solutions over time, instead of designing them by hand. Because almost any method, theory, or technique can be encoded on a computer, this implies an approach to problem solving that can be automated by a computer. More specifically, computer science has long been interested in how the design, development, and debugging of computer programs could be automated, and genetic algorithms provide one avenue toward this goal.

There are four major differences between GA-based approaches and conventional problem solving methods.

1. GAs use probabilistic transition rules, not deterministic rules.

2. GAs use payoff (objective function) information. Other supplementary knowledge of the problem may be useful but is not essential.

3. GAs search from a population of points, not a single point.

4. GAs work with a coding of the parameter set, not the parameters themselves.

These four properties make GAs robust, powerful, and data-independent [Goldberg, 1989b]. The GA operations, selection, crossover and mutation, primarily involve random number generation, copying, and partial string exchange. Thus they are powerful tools which are simple to implement. They have been applied to many areas, including VLSI layout optimisation, job shop scheduling, function optimisation and the travelling salesman problem.

1.2 Introduction to VHDL

VHDL is an acronym for VHSIC Hardware Description Language (VHSIC is an acronym for Very High Speed Integrated Circuit). It is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. The complexity of the digital system being modelled could vary from that of a simple gate to a complete digital electronic system, or anything in between. The digital system can also be described hierarchically. Timing can also be explicitly modelled in the same description.

The language not only defines the syntax but also defines very clear simulation semantics for each language construct. Therefore, models written in this language can be verified using a VHDL simulator. It is a strongly typed language and is often verbose to write. It inherits many of its features, especially the sequential language part, from the Ada programming language. Because VHDL provides an extensive range of modeling capabilities, it is often difficult to understand. Fortunately, it is possible to quickly assimilate a core subset of the language that is simple to understand without learning the more complex features. This subset is usually sufficient to model most applications. The complete language, however, has sufficient power to capture the descriptions of the most complex chips to a complete electronic system.

VHDL is a hardware description language and, therefore, VHDL descriptions are generally used to model hardware components and system (i.e. gates, chips, boards, etc). However, VHDL provides an abstract framework for describing hardware which is easily extended into other domains. A VHDL digital device can range from a gate to a microprocessor, to a complete system and beyond. The guiding factor is that the underlying system be based on a general stimulus-response model that uses discrete (i.e. non-continuous) values.

1.3 Introduction to Field Programmable Gate Arrays

A field-programmable gate array (FPGA) is an inexpensive hardware component, usually costing on the order of $100, which allows the user to program its functionality quickly and inexpensively. This allows for cheaper prototyping and shorter time-to-market of hardware designs. FPGAs are slower and have a lower gate density than full-custom (customised VLSI chips) and semi-custom (Mask-Programmed Gate Arrays (MPGAs)) design methodologies. However, FPGA costs per chip and turnaround times for low-volume designs and prototypes are much better than for MPGAs and full custom designs [Xilinx, 1994].

In general, FPGAs consist of *logic blocks*, *I/O cells* and *interconnection lines*. The *logic blocks* implement the actual logic of the FPGA using primitives such as NAND gates, multiplexers or lookup tables. The *I/O cells* allow the FPGA's logic blocks to connect to the pins. The *interconnection lines* connect logic blocks to each other and to the I/O cells. The routing done by these lines is implemented with wire segments and a system of programmable switches. The switching technology can be any one of pass-transistors controlled by static RAM cells, anti-fuses, EPROM transistors or EEPROM transistors.

FPGAs were first created by Xilinx Incorporated in 1984. Since that time, many other companies have marketed FPGAs, the major ones being Xilinx, Actel and Altera. Actel FPGAs use an anti-fuse technology which is programmable only once. Reprogrammable FPGAs use EPROM, EEPROM or static RAM technology. Xilinx FPGAs, using static RAM technology, are used in this thesis and in many other design projects which require hardware reprogrammability.

1.4 Mapping GAs to FPGAs through VHDL

Genetic algorithms are currently used in many applications as a robust general-purpose optimisation technique. For optimisation problems in which a solution must be computed quickly by genetic algorithms a hardware implementation may be necessary.

The nature of GA operators is such that GAs lend themselves well to pipelining and parallelisation. This capability for parallelisation and pipelining makes a GA an excellent candidate for mapping to hardware.

There are many ways to map a GA to an FPGA device, including Register Transfer Language (RTL), CUPL (for PLD devices) and VHDL. Synthesis tools are available for each method to create a PLD or FPGA device file from a source file. However, the basic operations of a GA require addition and multiplication that make it difficult to use RTL or CUPL as language tools or PLDs as a device for implementation. Fortunately in VHDL it is possible to use integer arithmetic without being concerned with the hardware implementation details. VHDL is an excellent environment for algorithmic implementation as long as there is not too much arithmetic in the design.

1.5 Thesis outline

The remainder of this thesis is organised as follows. Chapter 2 reviews the background for this work and methods of using reconfigurable hardware to speed up general applications. Chapter 3 explains the theory behind GAs and different types of GAs and genetic operators. For those who are not familiar with Genetic Algorithms, Appendix A explains GAs with a simple example. Chapter 4 describes the GAP model, hardware prototype and verification of the design and its performance analysis. In Chapter 5, the GAP will be analysed mathematically to find out the bottlenecks of the model. Chapter 6 explains the implementation of the GAP on FPGA devices including discussion about functionality and limitations of the design. Appendix D demonstrates a brief explanation of the Xilinx FPGAs and their internal structure. Chapter 7 deals with applications including a PID control system, dynamic power dispatch and an adaptive IIR filter. Several GAP configurations are tested to find the best configuration for each application. Chapter 8 demonstrates the multiple GAP, a parallel configuration for handling more complicated applications. Chapter 9 describes the adaptive capabilities of

the GAP and its application in the three problems. Chapter 10 presents conclusions and possible avenues for future work. Appendix A describes Genetic algorithms with a simple example. Appendix B shows Gray code tables and the VHDL code for the GAP modules is given in Appendix C. Appendix D describes Xilinx FPGAs and their internal structures.

Chapter 2

Background and related works

This chapter continues the basic description of genetic algorithms that was provided in Chapter 1. A much more detailed description is provided in Chapter 3. Next, VHDL is described and the advantages and disadvantages of using VHDL are discussed. This is followed by a review of related work in mapping frequently used software routines into configurable hardware and finally a brief review of research into previous hardware GA models.

2.1 Basic idea of Genetic Algorithms

Genetic Algorithms were developed initially by John Holland in the 1960's [Holland, 1975] as a form of search technique modelled on Darwinian evolution. The most accessible introduction is by Goldberg [1989b]. Other sources are Davis [1991b], Fogel [1995] and the Proceedings of the GA and PPSN conferences [Grefenstette, 1985;

Grefenstette, 1987; Schaffer, 1989; Belew and Booker, 1991; Forrest, 1993; Schwefel and Manner, 1990; Manner and Manderick, 1992].

GAs are relatively a new class of search algorithms in which good solutions to a problem are sought using an objective function. The search process in GAs is based on the natural evolution of biological organisms in which successive generations are given birth and are raised until they themselves are able to reproduce. GAs are becoming increasingly important mathematical tools for nonlinear optimisation problems.

For a genetic algorithm to improve a solution, it is necessary to reject the poor solutions and only allow reproduction from the best ones. This is analogous to the so called law of survival in which only organisms that adapt best to the natural environment tend to survive. In this case, the role of environment is played by a so called evaluating function, measuring the degree of fitness of an attempted solution to problem requirements. This function is equivalent to testing whether a given state is close to optimal. The use of a population of trial solutions helps the GA avoid converging to false peaks (local optima) in the search space.

A detailed description of different GAs will be discussed in Chapter 3 and a simple genetic algorithm definition with one example is included in Appendix A.

2.2 Applications of GAs

Genetic Algorithms have been employed in a wide variety of combinatorial optimisation and job scheduling problems including

1 - Travelling salesman problem [Homaifar et al., 1993].

2 - Job shop scheduling, rescheduling and open shop scheduling [Fang et al., 1993].

3 - Vehicle routing to service a set of customers with demands and least time for servicing [Thangiah et al., 1993].

4 - Pallet loading which involves the optimal packing of a predetermined number of cartons onto pallets [Jullif, 1993].

A complete list of applications can be found in the proceedings of the genetic algorithms conferences [Belew and Booker, 1991], [Forrest, 1993] and [Eschelman, 1995]. The use of GAs in this research has been inspired by a number of specialised applications of GAs in electronic engineering design including

1 - PID controller widely used in industry. In a PID (Proportional-Integrator-Differentiator) controller we attempt to drive a plant accordance to a given reference signal. The design objective is to determine a set of gains for the controller to match the set of roots of the closed loop control equation chosen by designer. PID controller design is often carried out by an experienced operator using a trial and error procedure. In applying GAs to a PID controller, a GA tries to estimate the three gain parameters of the PID controller while ensuring that transient response specifications are met [Hwang and Thompson, 1993].

2 - Economic Power Dispatch (EPD) which is used in power stations to schedule the supply of fuel to meet the system load demand at minimal cost. Conventional optimisation techniques become very complicated when dealing with complex dispatch problems and are limited by their lack of robustness and efficiency in practical applications. In this problem GAs are used to minimise an objective function, usually the total cost of generators, while satisfying both equality and inequality constraints [Walters and Sheble, 1993].

3 - Adaptive communication filtering in which the objective is to determine the optimum setting of parameters defining the system to minimise a suitably defined error function. There are two types of adaptive filters: adaptive FIR filters and adaptive IIR filters [Willsky, 1985]. Algorithms relating to the adaptation of FIR filters are well established. The role of GAs in adaptive IIR filter design is in the approximation of a

desired function (H1) by a rational transfer function (H2) for different values. This approximation is achieved by minimising an error surface between H1 and H2 [Roberts and Mullis, 1987].

4 - Channel Assignment in Cellular Mobile Networks: Frequency assignment is an important problem in cellular radio networks. Early methods used algorithms based on regular hexagonal arrays but as real cellular networks are far from regular, these algorithms are not suitable. GAs are used to find a good frequency assignment allowing for frequency reuse by non-adjacent cells which allows the number of communication channels over the network to be maximised with a limited number of frequencies [Kunz, 1991].

5 - Genetic Synthesis of Neural Networks: Neural networks are a technology in which computers learn directly from data, thereby assisting in classification, function estimation and similar tasks. Most classical learning algorithms for neural networks aim at finding weights for a neural network whose architecture is frozen. On the other hand a GA generates and tests a population of different architectures on a specific problem [Gruau, 1993]. The objective is to discover an optimal network for the problem [Davis, 1991b].

The above research has demonstrated that genetic algorithms can be used to produce near optimal solutions without regard to the complexity of the algorithms or the computing resources required. The approach in this thesis is to investigate whether a simple genetic algorithm embedded in hardware is effective on problems like these. This research also considers the real-time performance of these hardware devices.

2.3 The VHDL language

VHDL was developed to address a number of recurrent problems in the development, exchange and documentation of digital hardware. For instance, a typical microprocessor

would include thousands of pages of documentation to be sorted through during design and testing and referred to throughout the maintenance life of the component. When the component needs to be replaced, it takes substantial effort to reconstruct its intended behaviour. A good HDL design solves this problem because the documentation is executable and all elements are tied into a single model.

While there have been many hardware description languages, before VHDL, there was no accepted industry standard. Many of the existing languages have been developed to serve the simulators that run them, and are often proprietary developments of particular companies. Others target a particular technology, design level, or design methodology. VHDL is technology independent, is not tied to a particular simulator or value set, and does not enforce a strict design methodology. It allows the designer the freedom to choose technologies and methodologies while remaining within a single language. No one can foresee the changes that will take place in digital hardware technology. Therefore, VHDL provides abstraction capabilities that facilitate the insertion of new technologies into existing designs.

VHDL is used to describe a model for a digital hardware device. This model specifies the external view of the device and one or more internal views. The internal view specifies the functionality or structure of the device, while the external view specifies the interface of the device through which it communicates with the other models in its environment.

The following are the major capabilities that the language provides along with the features that differentiate it from other hardware description languages [Coelho, 1989]:

1 - VHDL supports hierarchical design. A digital system can be modelled as a set of interconnected components. Each component can then be modelled as a set of interconnected subcomponents.

2 - VHDL can be used for various digital modeling techniques such as finite-state machine descriptions, algorithmic descriptions, and boolean equations.

3 - VHDL is an IEEE and ANSI standard, and therefore, models described using this language are portable. There is a strong interest in maintaining this as a standard so that re-procurement and second-sourcing may become easier.

4 - The language is publicly available, easy to understand, readable, and above all, it is not proprietary.

5 - VHDL supports a behavioural description of hardware from the digital level to the gate level. One of the primary advantages of VHDL lies in its ability to capture the operation of a digital system on a number of descriptive levels at once, using a coherent syntax and semantics across these levels, and to simulate this system using any mixture of these levels of description. It is therefore possible to simulate designs that mix high-level behavioural descriptions of some subsystems with detailed implementations of other subsystems in the model.

6 - VHDL is not technology-specific, but is capable of supporting technology-specific features. It can also support various hardware technologies, for example, it may define new logic types and new components, it specifies technology-specific attributes. By being technology independent, the same behaviour model written in VHDL can be synthesised to utilise different vendor libraries (Such as PLDs, FPGAs or ASICs).

7 - VHDL is modelled on a philosophy similar to that of many modern programming languages that design decomposition aids are just as important as detailed descriptive capabilities. Packages, configuration declarations and the concept of multiple bodies exhibiting different implementations of an entity are all present in this language to support design sharing, experimentation and design management.

Chapter 6 explains how a VHDL model can be implemented on FPGA architectures.

2.4 Previous work in reconfigurable hardware

Recently there has been a sharp increase in work with reconfigurable hardware systems based on Field Programmable Gate Arrays (FPGAs) technology (see Chapter 6).

Gokhale et. al. [1991] developed a programmable linear logic array called SPLASH with several documented applications including one-dimensional pattern matching between a DNA sequence and a library of such sequences. SPLASH consisted of 32 Xilinx XC3090 FPGAs and 32 memory chips and greatly outperformed several supercomputers including a CM-2 and CRAY-2.

Athanas [1992] has been researching a series of reconfigurable computing architectures based on Xilinx FPGA technology. He developed the PLADO hardware platform which includes an array of Xilinx XC3090 FPGAs to assist in computation. The PLADO hardware platform worked in conjunction with the PLADO configuration compiler designed to analyse candidate hardware segments of a C program, choose the best segments for hardware implementation, and map these segments to the files necessary for programming Xilinx FPGAs. Candidate hardware segments were marked by the programmer and analysed by the compiler for feasibility of execution in a single clock cycle when implemented on the FPGA array.

At Digital Equipment Corporation's Paris Research Lab, Bertin et al. [1993] worked with a Programmable Active Memory (PAM) architecture which is a 5 x 5 array of Xilinx XC3090 FPGAs and supporting hardware, all combined to act as a coprocessor to a host system. Compiling and running an application on the PAM architecture consists of:

- identifying the critical computations best suited for hardware implementation,
- implementing and optimising the hardware part on the PAM,

- implementing and optimising the software part on the host system.

The PAM was tested on ten applications including data compression, string matching and binary 2D convolution. In each of the ten applications, the performance of the PAM implementation was competitive with a supercomputer implementation but was up to 100 times cheaper in cost per operation per second. The key, according to Bertin et al., is to choose an application with a single inner loop which is implementable on the PAM and which accounts for a vast percentage of the software run time. Thus many complex supercomputer applications are beyond the reach of current PAM technology.

The above research has inspired the production of commercial prototyping boards for implementing and testing FPGA designs. Virtual Computer Corporation [Casselman, 1993] now markets the line of Virtual Computers which consist of arrays of Xilinx XC4010 FPGAs and ICUBE IQ160 Filed Programmable Interconnect Device (FPID). XC4010s do the processing for the Virtual Computer while the IQ160s allow the user to program the interconnect between the XC4010s. The number of XC4010s ranges from 22 to 52. While other FPGA-based boards exist on the market today as prototyping boards, the Virtual Computer is intended to act as a reconfigurable coprocessor.

Another line of commercial products is from National Technologies Incorporated (NTI) [McLeod, 1994]. Their X-12 system uses a dozen XC3000 FPGAs, each with a 32K x 8 Static RAM (SRAM) at its disposal. Like the products from Virtual Computer Corporation, the X-12 system is intended for reconfigurable hardware use rather than prototyping.

As an example of an application, Eldredge and Hutchings [1994] used an NTI X-12 board to develop an FPGA based neural network which utilised run-time reconfiguration. To save the FPGA space, evaluation of different stages of the neural network (feed-forward, back propagation and updating) occupied the same FPGAs at different times during the run. When a new stage of the run was to be made, a master

system reconfigured FPGAs as needed and started the next stage. FPGA requirements were reduced to one fifth.

Another example of a reconfigurable system is given by Gilson [1993] who used an X-12 reprogrammable logic board from NTI to build a Nano-Processor (nP). The nP is a customisable stored-program processor which occupies less area than a fully customised reconfigurable hardware system still retaining a significant speed advantage over a conventional microprocessor due to the nP's customisable instruction set. The small size of the nP allows for some application-specific hardware to also occupy valuable FPGA space.

2.5 Hardware implementation of optimisation algorithms

Optimisation algorithms involve either a search for an optimum path through a network of discrete points or a search along a performance surface for a point of minimum error or maximum performance. This task is often time consuming and expensive for computers and can be addressed by specialised hardware. Algorithms like simulated annealing or genetic algorithms can be implemented on special hardware to improve the optimisation speed.

Annealing is a term borrowed from metallurgy to describe how nature can produce ordered structures in discrete systems of interacting particles by slow and careful cooling. As the system cools the small interaction between particles can lead to an ordered configuration representing the minimum energy state. Rapid cooling causes 'quenching' in which irregularities become frozen in and the system remains disordered.

Simulated annealing models this process on a computer in an attempt to solve some large scale optimisation problems. The method of simulated annealing can be applied to many problems of combinatorial optimisation. Such problems involve a search through a data structure for a path which optimises some property of the data. In many

mathematical problems we are not dealing with 'energy' levels or forces between particles but we can still employ the same principles to achieve optimum minimisation [Kirkpatrick et al., 1983].

Abramson [1992] described a special purpose machine for solving an integer programming problem using simulated annealing. The hardware executed the code about 100 times faster than the same program running on a workstation and cost less than a personal computer to build. Interestingly, this machine gained its performance from two main sources. First, it utilised very low level concurrency which cannot be extracted by vector and parallel computers. Second, it avoided all address arithmetic normally required for matrix manipulation. The board was implemented using conventional logic devices and was hosted by a PC or workstation. It was controlled by a simple finite state machine which implemented the annealing algorithm, and contained no fast logic or pipelined stages.

Abramson et al. [1995] are now working on a project called Guess to develop a class of computer architectures which deliver very high performance on solving integer optimisation problems. These architectures will be designed to support both simulated annealing and branch-and-bound algorithms through system reconfiguration. Guess makes use of an Aptix AP4 reconfigurable logic board which contains up to 16 Xilinx 4010 Field Programmable Gate Arrays, plus a number of Aptyx switch chips. The switches make it possible to connect the pins of the Xilinx parts together, and thus the board is totally reconfigurable.

2.6 Previous work in hardware Genetic Algorithms

So far little work has been done in implementing a hardware-based GA. Husband and Mill [1991] have implemented a version of a GA on a transputer based parallel machine to optimise a manufacturing scheduling problem.

Spiessens and Manderick [1991] implemented a genetic algorithm on a Distributed Array of Processors (DAP). The DAP from Active Memory Technology Ltd is a fine grain massively parallel Single Instruction Multiple Data (SIMD) computer. The processors are arranged in a 2- dimensional cyclic mesh. The size of the dimensions is 32*32 for the DAP-510 and 64*64 for the DAP-610. Each processor has a direct connection to its own local memory. The DAP attaches to a host computer which is used for program development, debugging, loading and controlling DAP program. The DAP is mainly programmed in a version of the FORTRAN language which includes extensions for dealing with vectors and arrays as single objects.

DCP Research Corporation in Edmonton, Alberta has implemented a suite of proprietary GAs in a text compression chip [Wirbel, 1992].

Chen et al. [1993] have implemented a GA for diversity minimisation on a Thinking Machines Corporation Connection Machine CM-5 in the Computer Science Department at the University of Wisconsin-Madison. This machine consists of 64 SPARC processors each with 32 megabytes of local memory connected by a "fat free" network. In Multiple Instruction Multiple Data (MIMD) mode, the processors run asynchronously and communicate via calls to the routines in a message passing library.

Tetsuya Higuchi et al. [1994] at the Electrotechnical Laboratory in Tsukuba are developing self-adapting hardware which uses a GA to modify hardware configuration bit strings that control the connections in programmable logic devices. The evaluation function in this case is a string's performance in particular tasks, e.g. controlling a robot arm.

Chapter 3

Principles of Genetic Algorithms

This chapter is provided as an introduction to genetic algorithms, a class of optimisation algorithms that draw their inspiration from evolution and natural selection. The intension is to describe the features and variations of GAs and to give the reader an idea of the sophistication that may be employed to enhance performance. In the design of hardware GAs, there is a need to avoid complexity and some of the features described in this chapter are very difficult to implement in hardware.

GAs were defined by John Holland in his 1975 book: "Adaptation in Natural and Artificial Systems" [Holland, 1975]. Since then the GA community has gradually grown, mostly in the USA, with a series of international conferences starting in 1985. However, Holland's book is rather theoretical, and a more accessible book is Goldberg's "Genetic Algorithms in Search, Optimisation and Machine Learning" [Goldberg, 1989b]. With the arrival of this and Davis's "Handbook of Genetic Algorithms" [Davis,

199lb], interest in GAs looks set to increase further. What follows is a review of the art of GAs, recovered both from the literature and experimentation.

3.1 The simple principles

A GA operates on a problem that is specified in terms of a number of parameters. For a function optimisation, these may be the values of coefficients for the real time operation of an industrial plant, the control settings for a neural network, the numbers of units or the learning rates. One key feature of GAs is that they hold a population of such parameters, so that many points in the problem space are sampled simultaneously. The population is generated either at random or by some heuristic. The former is usual when the aim is to compare different algorithms. The latter may be more appropriate if the object is to solve a real problem. Each set of parameters may be regarded as a vector, but the traditional name is a string. Another key feature of Holland's GA is that these parameters are bit strings, with real or integer valued problem parameters being coded by an appropriate number of bits. The nature of this coding is functionally extremely important and is discussed further in Section 3.5.1. Each string is rated, by running the system that is specified. In the case of a function evaluation, this may be very quick. For an aircraft simulation [Bramlette and Bouchard, 1991] or a neural network, the evaluation might take minutes or even hours. A new population is then generated, by choosing the best strings preferentially. A simple way of doing this is to allocate children in proportion to the test performance (or rather, in proportion to the ratio of a string's test performance to the average of all the strings). With no other operators affecting the population, the result of this is that the best string increases in number exponentially, and hence rapidly takes over the whole population.

Novel structures are generated by a process resembling sexual reproduction. Two members of the new population are chosen at random, and new offspring are produced by mixing parameters from the parents. In the earliest work [DeJong, 1975], a single

crossover was used, where parameters were copied from one parent up to some randomly chosen point, and then taken from the other. Thus the strings ABCD and EFGH might be crossed to produce AFGH and EBCD. Much subsequent work on GAs has studied the relative merits of different recombination algorithms. The preferred form of recombination is problem and coding-dependent and some other possibilities will be discussed further below.

A second operator that introduces diversity is mutation in which the value of a parameter is changed arbitrarily. This process is not the major source of new structures, that is the role of recombination, but it serves to produce occasional new "ideas", and to replace combinations that might be lost in the stochastic selection processes. The precise role of mutation depends on the coding used in the genes and is also discussed further below.

The cycle for a basic genetic algorithm is as follows. Generate a population of parameter sets, test them against the problem, select for reproduction on the basis of performance, recombine pairs of parameter sets and mutate a few to generate the new population and restart the cycle. We shall now look at each aspect of the algorithm in more detail.

First a note about terminology. GAs are inspired by biological evolution, and exponents often borrow terms from the study of natural genetics. Some workers refer to strings as chromosomes, their natural analogue. Genotype and phenotype may be used to describe the genetic string and the decoded parameter set respectively. We need to distinguish between the parameters of the target problem and the components of the genetic string. The term *gene is* often used for the components. This is an inaccurate interpretation, since in biology a gene is usually taken to be something that codes for a whole trait, such as blue eyes. However, the application of GAs has not advanced to the point where this meaning of gene would be useful. Therefore the term will be adopted here to mean the individual components of a string, while parameter refers to the target problem. A

real-valued parameter might be coded directly by a real-valued gene, or by a number of binary genes. Possible values of a gene are commonly known as *alleles*: 0 and 1 for a bit string. The set of possible alleles is known as the *alphabet.* Finally, a distinction will be made between *crossover* and the more general *recombination.* Crossover is the traditional form of recombination, simply selecting between the parent strings and not affecting gene values. The simplest form of crossover changes from one parent to the other at a single point.

3.2 Evaluation

There may not seem much to discuss about evaluation of the parameter set. If the task is an artificial one, such as a function evaluation that is being used to test the GA, then there should indeed be no problem, provided the function is deterministic. Where the function is stochastic, as many real-world processes are, there is the issue of how much to try and reduce the noise. GAs are relatively immune to noisy evaluations, compared with, for instance, gradient ascent methods that may be thrown right off course by an odd result. However, it is still naturally the case that accurate evaluations are to be preferred to noisy ones. The accuracy can be improved by doing q evaluations and averaging, the noise decreasing with $\sqrt{q}$. However this may not be the best approach, particularly if the evaluation takes a long time. There is evidence [Grefenstette and Fitzpatrick, 1985; Fitzpatrick and Grefenstette, 1988] that it is better to do a fast, noisy evaluation and get on to the next generation, rather than spend time accurately assessing each individual.

Another important aspect of the evaluation procedure is that it should reflect the desired target problem. One part of this is simple accuracy. Suppose the aim is to improve the design of a jet engine. The parameters might be values such as the angle and size of fan blades. Clearly the real engines will not be tested as specified, it would be done by computer simulation. However, the end product can only be as good as the simulation.

A rather more subtle aspect of the simulation has to do with constraining it sufficiently. This became apparent in some work on tuning neural network parameters [Spears, 1989]. The only information the GA gets is the evaluation result, usually a simple scalar value. When, for reasons of evaluation time, the test is a reduced version of the real task, it must be very carefully constructed.

One difficulty is the need to optimise more than one aspect of performance simultaneously, or to optimise one subject to some constraints. For instance a neural network may be required to do as well as possible, but quickly, or without exceeding some size. It may be possible to build such constraints into the operators that produce new strings. This is usually to be preferred, since it both avoids the problem at evaluation time and concentrates search in fruitful areas. However, such operators may not be feasible, either because it is simply very difficult to satisfy all the constraints, or because the various factors, test score and run time in the neural network case, only become available after evaluation.

The standard GA requires a scalar evaluation value for the parent selection process, so the various test values and constraints need to be combined. The easiest method is some linear combination. If the balance between the components is not good, the GA will surely optimise the easiest one at the expense of the others. It may be that the only way to discover the correct combination is by trial and error. A possibility that might merit investigation is to alter the balance dynamically. For instance if, during the GA run, the evaluation time dropped below some limit, the time element in the evaluation function could be reduced.

Richardson [1989] has looked at various ways of handling penalty functions for constraint satisfaction. It might be thought that violation of constraints should be harshly penalised. However, Richardson argues that this may cause the GA to fail, especially if it is difficult to satisfy the constraints. His suggested solution is to try and

construct a penalty function that is proportional to the distance of the string from feasibility, rather than simply counting the number of constraints that have been violated.

In some cases there is more than one potential measure of the same aspect of a string's performance. In the application of GAs to learning the weights for a neural network, the error of a network may be measured in a number of ways, for instance the sum squared error across all the training set, or the worst individual bit error. While the aim of training is usually taken to be minimising the squared error, the real target for a binary training set is to get each individual bit the correct side of 0.5. However, if this was set as an evaluation target when using the traditional sigmoidal output function, the GA always become stuck with all the values just above 0.5. If the squared error alone was used, the GA tended to minimise it quickly by solving the easy bits, and letting the hard ones go to 1.0 error. It was then unable to correct the remaining bits and a combination had to be used.

GAs are by no means reliable, and sometimes no progress is made on a problem. Perhaps there are too many constraints, or the area of the possible search space that gives scores significantly better than zero, is too small. A possible approach, used in some work on parameter tuning [Hancock, 1989], is to alter the evaluation function during the GA run. The problem is initially made easier, perhaps by relaxing some of the constraints, so that the GA is able to make some progress. When some performance level is achieved, the task is gradually made harder. This approach makes strong assumptions about the presence of a fairly continuous path in the search space as the task changes, which may be unjustified. While a GA may be expected to do a reasonable job of finding a way past some discontinuities there can be no guarantees.

In some optimisation procedures, it is natural to talk about the optima being small values. Others are more naturally described as hill-climbing algorithms. It makes no real

difference to a GA whether it is aiming to go up or down. However, descriptions of strings as being high-ranking, or having high fitness, suggest that hill-climbing is the natural target. Except where stated otherwise, this will be the case in this work.

3.3 Scaling and selection

Having evaluated the strings, the best need to be selected in some way to form the new population. There are two aspects to this process: how to decide what proportion of the new population should come from each string, and, how to cope with the reality of a finite population size.

The simplest means of allocating strings to the new population is in proportion to the ratio of their evaluated fitness to the average of the whole population. Thus if a particular string has twice the average fitness, it would be expected to be chosen twice to act as a parent. This was the method used in the first thorough experimental work on GAs, reported in DeJong's thesis [DeJong, 1975]. While it works well enough for nicely behaved functions, it can cause problems if the function has large areas of poor performance, with localised good spots. Once one string finds a good area, its fitness will be far above the average. It will dominate the next generation, with consequent loss of diversity, a phenomenon known as premature convergence. Conversely, towards the end of an optimisation, most of the population should be highly rated. Those that are slightly better than average get little selective advantage, and the search stagnates.

The traditional approach to this, implemented in Grefenstette's public domain GA program Genesis [Grefenstette, 1987], is to use a movable baseline for the evaluation. This is typically set to the evaluation score of the worst string, either in the current generation or within some small (5-10) window of recent generations. The baseline may be set somewhat below the worst value, to ensure that even the worst string gets some chance to reproduce. This can be important, both as a general guard against premature convergence and because poor strings may be poor because they are on the shoulder

between different maxima. Indeed allowing poor individuals to reproduce entitles the evolutionary system to escape local maxima. The baseline re-expands the fitness scale such that, for instance, the ratio between 0 and 1 is the same as that between 99 and 100. The problem of exceptionally good strings is handled by using a scaling algorithm that ensures a constant fitness ratio, typically about 2, between the best and the worst.

A more radical approach suggested by Baker [1985] is to use the fitness scores only to give a ranking and then assign a fixed hierarchy of selection probabilities. It is possible to use a geometric scaling [Montana and Davis, 1989], such that the best string is assigned a fitness of say 0.9, the second, 0.9^2, the third, 0.9^3 and so on. The scaling factor can be varied during the run so as gradually to increase the selection pressure, perhaps starting at 0.95 and ending at 0.85. One potential advantage of this method is that the evaluation no longer needs to return a single scalar value.

A disadvantage of the method is that the selection pressure, in terms of the ratio of selection probability of best to worst, is dependent on the population size. This must be remembered when comparing different GA runs. Whitley [1989] has suggested an alternative algorithm for use in Genitor that avoids this effect. However, this implements a linear scaling rather than the geometric scale proposed in [Montana and Davis, 1989]. The latter gives relatively more reproductive opportunities to the better strings.

Having decided the ideal proportions, some finite number of copies of each string must be chosen for reproduction. The simplest method of doing this is to add up the total fitness (whether scaled or not). Then, for each string to be selected, pick a random number between 0 and that total and work through the list of strings, summing their fitness values until a number bigger than the random one is reached. Each string will then be chosen with a probability that reflects its share of the total fitness. The process is known as roulette wheel selection, it being equivalent to spinning a wheel where the

sectors are allocated according to each string's fitness. However, Baker [1987] showed that the random nature of the algorithm can result in significant inaccuracies in the selection process. He suggested a more accurate algorithm, called Stochastic Universal Sampling (SUS) that guarantees the correct whole number of offspring for each string. Fractional numbers of expected offspring are allocated proportional, so if 1.7 are expected, 1 will be obtained with probability 0.3 and 2 with probability 0.7. Another way of looking at the algorithm is as a modified roulette wheel, with as many, equally spaced pointers as strings to be selected and only one spin is required. This algorithm can make a remarkable difference in performance, particularly in small populations. In some cases it has produced an order of magnitude improvement in solution time.

3.4 Generation and crowding

The simplest method of running a GA is to replace the whole population each generation. In this case, therefore, the generation size (the number of strings evaluated in each generation) is equal to the population size. This was the method used for most of DeJong's main work [DeJong, 1975]. A more conservative method is to ensure that the best string from the previous generation survives, by simply adding it to the pool of the new generation if necessary. DeJong calls this the elitist strategy, and he showed that it generally improves performance on unimodal functions. The elitist strategy ensures the best string survives the whole generation procedure. On multimodal functions the strategy may be less beneficial, since it can make escape from a local maximum more difficult. A compromise that has been used by many people is to keep the best for a few, perhaps 5, generations, but then delete it if no further progress has been made.

The generation size may be smaller than the population, in which case some method must be used to decide which of the old population to delete. This may be done at random, or weighted to make the worst number most likely or even certain to go. An interesting alternative, intended to reduce premature convergence, was introduced by

DeJong when tackling a function designed to have multiple local maxima. For each member of the new generation, a small number C (the crowding factor) of the old population are chosen at random. The one with the highest number of bits in common is replaced by the new string. This effectively introduces competition between strings that are close together in the parameter space, discouraging convergence on one good spot. The strategy gave significantly enhanced performance on the multi-modal function. The required value of the crowding factor C is surprisingly small - 2 or 3 for a population of 100. If it is much larger then the system will have difficulty converging on any maximum.

A significantly different GA model uses a generation size of just one. This was introduced by Whitley with his Genitor system [Whitley and Kauth, 1988], and termed steady-state reproduction by Syswerda [1989]. Genitor is very conservative, the offspring is only added to the population if its performance exceeds the current worst, which is then deleted. An apparent drawback of this method is that the one-at-a-time selection procedure inevitably suffers from the same kind of sampling error as roulette wheel selection. This is unlikely to affect good strings, since they will in any case survive for many evaluations (until they become the worst), but may result in the weak strings getting less chance to breed than they should. This potential loss of diversity is moderated by ensuring that there are no duplicate strings. However, the potential sampling error on poor strings combined with the very conservative memory of the good ones suggests that the system may have difficulty in escaping from local minima. This is supported by Whitley's results on DeJong's original test set [Whitley, 1989].

3.5 Reproduction and coding

This section discusses the various operators used to create a new generation from the strings selected to be parents. The key to the explorative power of GAs is held to be recombination. The numerical arguments in favour of recombination are easy to see

[Davis, 1991b]. Suppose two new alleles are required to cause a big fitness improvement in a population. Such new alleles can only come from mutation, which happens infrequently, say with a probability of 10^{-6} per reproduction. If each new allele presents some advantage, then without recombination, strings containing one or other will eventually appear and prosper, but still have to wait for another rare mutation to acquire both. With recombination it requires only that two strings each with one of the alleles interbreed.

Despite this, there have been claims that recombination contributes nothing to the optimisation process [Fogel and Atmar, 1990]. Whether or not it does contribute usefully depends very much on how it interacts with the underlying coding of the strings.

3.5.1 Coding

One of the important differences between the other methods in evolution strategies and Holland's GA is the form of coding of the parameters. Other approaches hold the parameters as normal computer variables: integer or real as appropriate. While some work on GAs also uses this form of coding, Holland specified a bit-string coding. Some problems contain boolean parameters for which such a coding is ideal. However, real or integer parameters may be coded with arbitrary precision by using sufficient bits. Any digital computer will have such a bit coding in any case, but the details are usually hidden from high-level languages.

Whether or not to use bit coding is a contentious issue. GA-purists tend to regard real-coded algorithms as not being proper GAs. Meanwhile more pragmatic experimenters have produced good results with real coding. We shall first consider the advantages claimed for bit coding.

Access to the bit level gives the crossover operator the ability to explore the whole search space. Given just the strings containing all ones and all zeros, repeated application of simple crossover can in principle produce any desired bit pattern, and therefore any parameter values. If the parameters held directly as real values, then crossover can only explore new combinations of the values existent in the population. Actually the same is true of bit-coded GAs, crossover cannot affect a bit that has the same value in every member of the population. It is the role of mutation to replace bit values that may have been lost so that crossover may form new combinations. Real-coded algorithms depend more heavily on mutation to provide new values. Since mutation is random, it will destroy good parameter values as well as improving bad ones.

A possibly more important reason for using bit coding has to do with the way the search space is sampled. It maximises the effect known as intrinsic parallelism, a prediction of schema theory, to which we now turn.

3.5.2 Schemata

A problem with any optimisation procedure is credit assignment. Suppose we have a good result: which of the parameters caused it? Most likely several in combination. A *similarity template* or *schema,* in this context, specifies some of the parameters, leaving others as "don't care" (usually shown as "*"). Schemata provide a way of describing the underlying similarities between successful strings. There are many such schemata contained within even a short binary string. For instance 1101 contains 11**, *10*, *1*1: 16 (2^4) in all. All 16 of these schemata are selected and evaluated when the complete string is. The reproduction operators are processing not only the basic strings but also all the constituent schemata. Each schema is likely to be represented by many strings, so it is possible to work out an average score for each. Such explicit calculation is unnecessary, however, as the process is automatically handled by the selection of whole strings, good schemata will thus tend to increase in numbers. Using f_s as the

fitness of schema s, $C_s(g)$ as the number of copies in the population at generation g and $\bar{f}$ as the average fitness of the whole population, we can write an expression for the expected number of copies in the next generation:

$$E(C_s(g+1)) = C_s(g)\frac{f_s}{\bar{f}}p(s) \tag{3.1}$$

The unexplained term $p(s)$ is the probability that the schema survives the reproduction operators. The likelihood that a schema is affected by mutation depends on the number of defined bits in the schema, known as the *order* of the schema. The likelihood that a schema of order o_S survives mutation is $(1-p_m)^{o_S}$, where p_m is the probability of mutation at each bit. For the typically small values of p_m that are used, this may be approximated by $(1 - o_S p_m)$. Note that some users take p_m to be the probability that a bit is randomly reset, so that the chance of it being changed is actually half p_m. This is implemented in Grefenstette's public domain Genesis package [Grefenstette, 1987], though it has been changed back to the more natural use in Schraudolph's GAucsd development of Genesis [Grefenstette and Schraudolph, 1992].

The probability that a schema survives crossover is a function of its *defining length* d_s. This is the distance between the first and last defined bits. Thus the schema 1*1** has d_s = 2. There are two possible positions where a crossover could come between the defining bits. Short schemata have a proportionately better chance of surviving crossover than longer ones. For a string of total length l, the chance of a schema surviving a single point crossover with paobabilty p_c is

$$p(S) \geq 1 - p_c\frac{d_s}{(l-1)} \tag{3.2}$$

because there are (l - 1) possible positions for the cross site. The inequality exists because, unlike mutation, crossover does not imply loss of the schema. In the limit,

crossing two identical strings will have no effect on any schemata. So the calculated loss is a worst case.

$$E(C_s(g+1)) \geq C_s(g)\frac{f_s}{\bar{f}}\left(1-p_c\frac{d_s}{(l-1)}-o_s p_m\right) \tag{3.3}$$

This is the *schema theorem*, originally from "*The fundamental algorithm of Genetic Algorithms*" by Goldberg [1989b]. If the fitness of a schema is sufficiently above average to outweigh the loss terms, its proportion in the population will grow exponentially. This is most likely for short defining length, low order schemata.

One source of the power of GAs is that many schemata are being processed simultaneously. The number may be estimated [Holland, 1975; Goldberg, 1989b] as the order of m^3, where m is the size of the population. This phenomenon is known as *intrinsic parallelism,* and has been described as the only case where an exponential explosion works to our advantage.

The order of m^3 estimate hides an assumption as to the value of m, which is chosen to expect one copy of each schema being processed. The derivation of the estimate is given by Goldberg [1989b]. The chance that a schema survives crossover is related to its defining length. Depending on the selection pressure within our GA, we may set a required survival probability for schema that will be processed usefully. By using the survival probability equations from above, we may calculate a maximum useful schema length l_s. The estimate for the number of usefully processed schemata n_s is then [Goldberg, 1989b]:

$$n_s \geq m(l-l_s+1)2^{l_s-2} \tag{3.4}$$

The order of m^3 estimate arises from assuming a population size of $2^{l_s/2}$. This is done to prevent over estimating the total number of schemata by having many copies of each in a large population.

For a given small population, clearly n_s is highly dependent on l_s. This is the origin of the desire for a *low cardinality alphabet*, i.e. a coding where each gene has few possible alleles, preferably 2, since then the length of the string, and therefore the value of l_s will be maximal.

Unfortunately, the success of this parallel search is not guaranteed. It requires that two good genes in combination will produce a better result than either alone. This is known as the *building block hypothesis*. It rates lots of schemata in parallel, but how does the performance of a given schema, say 1****, relate to the performance of more defined schemata that incorporate it, such as 1***1? For an extremely simple optimisation problem such as maximising the number coded by the binary string, the combination is straightforward. 1**** will be highly rated, ****1 much less so, but higher on average than ****0, so its numbers should increase. If it does not already exist, crossover will soon produce 1***1, which will be better than either parent.

Although tasks like that have been used for testing GAs, it is not immediately clear that the hypothesis applies so well in other problems. It is possible to test the behaviour of GAs under such circumstances by designing problems that are deliberately deceptive and thus might be expected to mislead the algorithm [Goldberg, 1987; Vose, 1990]. Suppose that an integer parameter happens to have a maximum at 8. If the function is smooth, then 7 will also get a good score, but its binary coding will be very different. The schema **111 may be quite highly rated, but it is unlikely to reach 01000. The use of Gray codes is one possible solution for this "Hamming cliff" problem that we turn to that in the following section.

3.5.3 Gray codes

Gray codes have the property that the binary codings of adjacent integers differ in only one bit, see Appendix B. For instance, the Gray code for 7 is 0100, for 8 it is 1100. This means that such changes can always be made by a single mutation. The use of Gray coding might therefore be expected to improve the hill-climbing ability of a GA. Its use was suggested by Hollstien [1971], who reported tentative benefits, and by Bethke [1981], who also reported empirical success. Caruana and Shaffer [1988] report that it improves performance on DeJong's classic 5 problem test suite [DeJong, 1975]. Some authors have therefore adopted Gray coding as standard, while it is an option on the Genesis package.

There are also arguments against the use of Gray codes, to do with the schema theorem. Goldberg [1989a] hints at a problem in his analysis of the use of Walsh codes in deception, but it is quite easy to demonstrate. With simple binary coding, a given bit always makes the same contribution to the value of the external parameter. With Gray coding, this is not the case. Thus, coding integers from 0 - 15 in normal binary, the schema ***0 has an average value of 7, while ***1 has an average of 8, reflecting the value of the least significant bit. With Gray coding, both schemata have the same average, 7.5. There is no longer any information about the merits of setting this bit from the overall averages, which suggests that the degree of implicit parallelism will be reduced. Such interdependence between bits is commonly known as *epistasis,* another term borrowed from biology.

The effects of a single bit mutation clearly differ between the two coding strategies. Mutating the most significant bit in a standard binary coding causes a big change in the number being represented. In Gray coding, adjacent numbers differ in only one bit, so it might appear that a single bit mutation will cause less dramatic effects. However, there are still highly significant bits in a Gray code. For instance, the Gray code for 0 is 0000, as with ordinary binary, but 15 is 1000. The possible big changes balance out the small

ones, so that the expected average change caused by a single mutation is the same for both codings 3.75 over the range 0-15.

If the use of Gray coding interferes with the parallel search, but makes mutation-driven improvement easier, a Gray coded GA should be more sensitive to the mutation rate. This prediction was tested by looking at DeJong's test set, using the Genesis package [Grefenstette and Schraudolph, 1992].

The test set, although quite carefully constructed to include a variety of problems, is now showing its age. The five functions are given in Table 3.1. They have been heavily criticised by Davis [1991a], who shows that a simple bit climbing algorithm out-performs standard GAs on all but one of them. This is because they are rather regular, for instance the optima are conveniently placed at zero in F1 and F4 and at one end of the range in F3. F5 looks frightful, being a plane with 25 sixth order fox-holes, differing only slightly in depth. However, the holes are laid out on a regular grid, that actually makes solution rather easy since a change in only one parameter can cause the move to the adjacent hole. It seems clear from Davis's results that these functions should no longer be used for comparison of new algorithms. They are used here simply to demonstrate some of the differences caused by changes in coding strategy.

Except for F4, all the runs used a population and generation size of 100, the elitist strategy and a two-point crossover probability of 0.6 (if not selected for crossover, a string is passed to the next generation unaltered except for possible mutation). They were run for 4000 evaluations, or, since strings that were simply duplicates of a parent were not re-evaluated, until two generations had passed with no evaluations. F4 has a much longer string than the others, and showed a tendency to premature convergence with a population of 100. It was run with a population of 400, for 20000 evaluations, but with duplicates being re-evaluated, since the function has noise added. The results are shown in Figure 3.1a to 3.1e. The graphs give the best value obtained, averaged over 10

experiments, at a variety of mutation rates. Note that DeJong's functions are defined as minimisation tasks, so low values are good.

Function Number	Function	Range
F1	$\sum_{i=1}^{3} x_i^2$	$-5 \le x_i \le 5$
F2	$100\left(x_1^2 - x_2\right)^2 + (1 - x_1)^2$	$-2 \le x_i \le 2$
F3	$\sum_{i=1}^{5} \text{integer}(x_i)$	$-5 \le x_i \le 5$
F4	$\sum_{i=1}^{30} i\, x_i^4 + Gauss(0,1)$	$-1.3 \le x_i \le 1.3$
F5	$0.002 + \sum_{i=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6}$	$-65 \le x_i \le 65$

Table 3.1: DeJong's five test functions.

The most remarkable result is given by the simplest unimodal task, F1. The Gray coded algorithm showed a very marked dependence on mutation rate, being significantly better than normal binary coding only over a fairly narrow band, and much worse if the mutation rate was too low. This fits with the hypothesis that Gray coding is more dependent on mutation for progress, but is potentially better at hill climbing. The results for F2 and F3 show a similar mutation-dependency for Gray coding, for which the results are somewhat worse than simple binary coding. Gray coding appears better for F4 and F5. Note that the optimum of F4 is undefined, because of the Gaussian noise term, which accounts for the negative values shown on the graph. In F4, mutation appears to have no beneficial effect at all.

These results have been includes to show a number of effects:

• Even when averaging over 10 runs, there is a fair amount of noise in the data. GAs are stochastic algorithms, and any paper claiming comparative results based on just one run, of which there are a surprising number, should be treated with suspicion.

• Mutation, on the whole, has the expected effect, too little or too much being harmful. Too little usually results in premature convergence, too much is disruptive. However, the ideal rate varies between different problems. It is related to string length l, a reasonable first approximation being $1/l$.

• Gray coding is certainly not a universal solution. Even in the simple hill-climbing case of F1, precisely the sort of problem it is intended to address, its relative sensitivity to mutation rate means that its advantage over simple binary coding is not consistent.

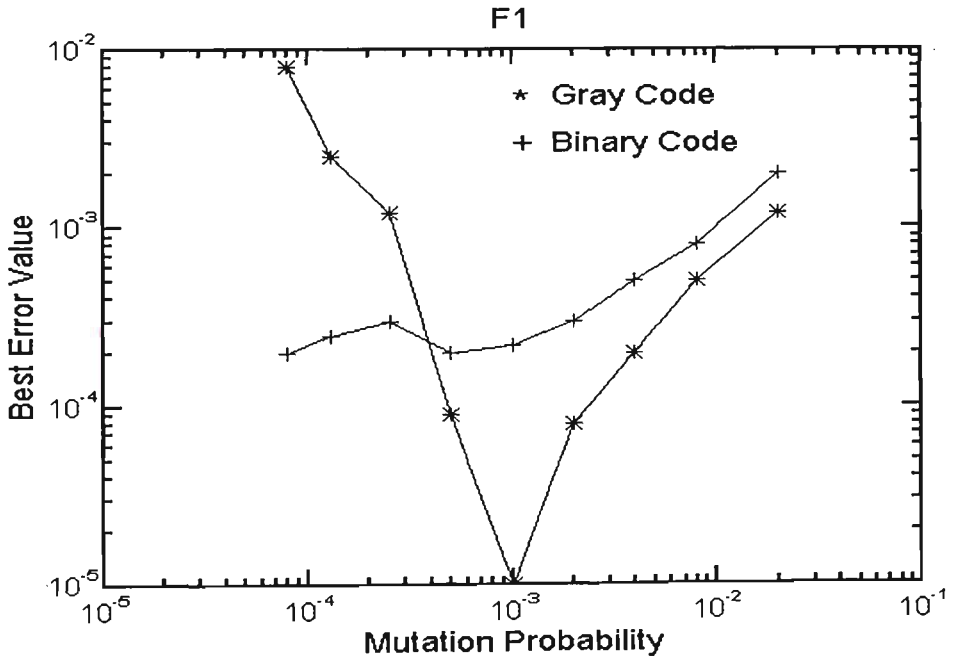


Figure 3.1a: Comparison of best performance of Binary coding "+" and Gray Coding "*" on DeJong's F1 function at a variety of mutation probabilities.

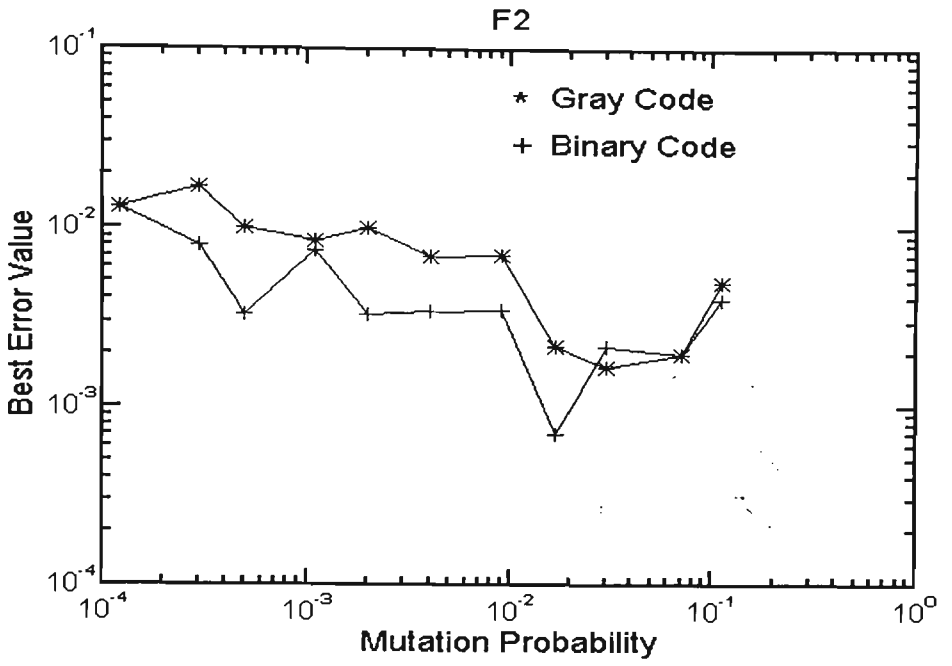


Figure 3.1b: Comparison of best performance of Binary coding "+" and Gray Coding "*" on DeJong's F2 function at a variety of mutation probabilities.

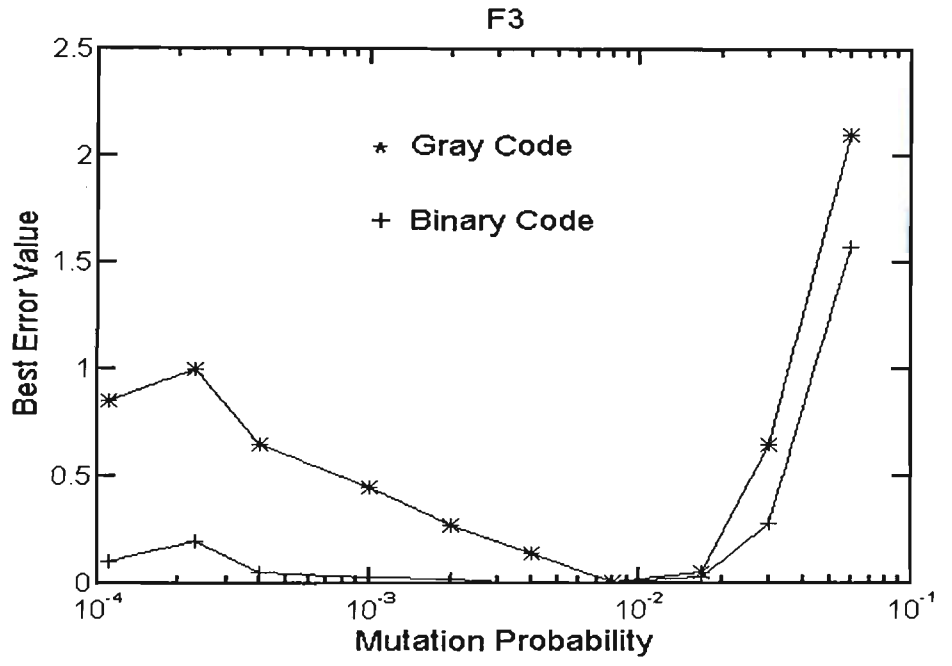


Figure 3.1c: Comparison of best performance of Binary coding "+" and Gray Coding "*" on DeJong's F3 function at a variety of mutation probabilities.

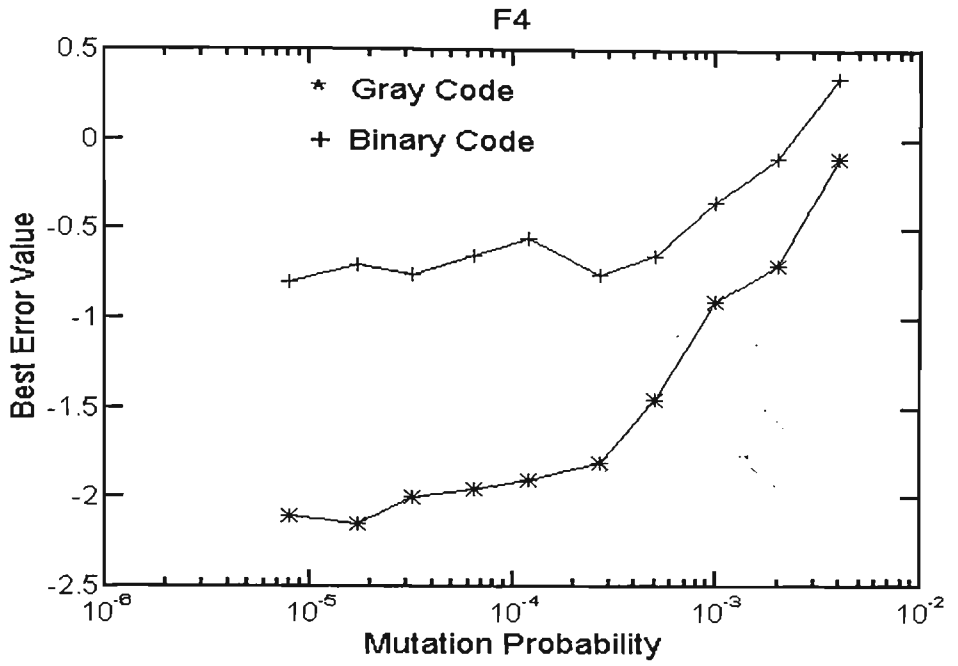


Figure 3.1d: Comparison of best performance of Binary coding "+" and Gray Coding "*" on DeJong's F4 function at a variety of mutation probabilities.

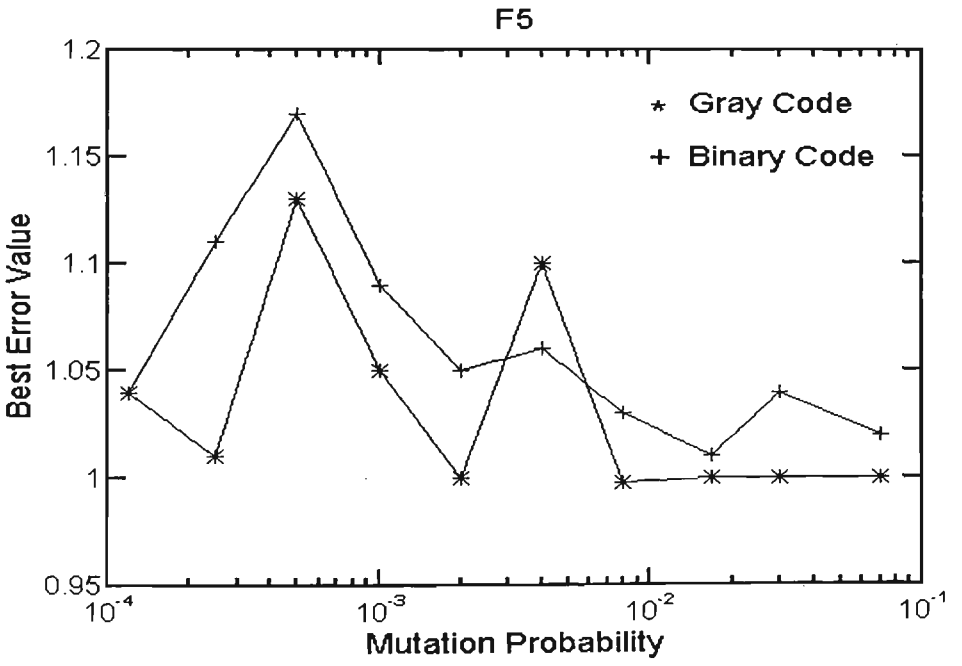


Figure 3.1e: Comparison of best performance of Binary coding "+" and Gray Coding "*" on DeJong's F5 function at a variety of mutation probabilities.

It is not surprising that Gray codes are not consistently better. It can be shown that all fixed coding schemes of a given length work out equally well when all functions are averaged over [Caruana and Schaffer, 1988]. However, this result depends on averaging over literally all functions, including those which are random look-up tables. Any given coding then simply reshuffles the entries in the table. Attempting to optimise such functions is rather fruitless, real-world problems usually have at least some continuity. The relative empirical success of Gray coding suggests that it matches the GA to the continuities of such problems, at least better than the inherently discontinuous binary coding.

3.5.4 Real value genes

Goldberg [1990] has discussed arguments for and against real coding. He argues that selection by the GA rapidly reduces the range of parameter values present in the population, to form a *virtual alphabet,* which is then used for further processing. His reasoning may be summarised as follows. In the early generations of a GA, each parameter can be treated individually, since there has not been time to collect much information about combinations of parameter values. An average fitness can be calculated (in principle) for all values of each parameter, by integrating over all values of all the other parameters. Goldberg calls this a *mean slice.* Unless the parameter has no effect on the function, some parts of its range will be better than others.

Goldberg and Deb [1991] show that these above-average regions will come to dominate the population very quickly, in the order of *log(log(m))* generations, where m *is* the population size and *log* is in base 2. Thus for a typical population of 100, only above-average alleles are left after 3 or 4 generations. Note that an allele in this case may be a sizeable region of the parameter's range, and that there may be several disconnected regions within the range. After the initial selection has taken place, the action of crossover is limited to exploring combinations of these sub-ranges, Figure 3.2. Goldberg argues that, presented with a high (infinite) cardinality alphabet, the system effectively

produces its own lower-cardinality virtual alphabet, one specifically tailored to the problem in hand. This may explain the empirical success of real-coded GAs.

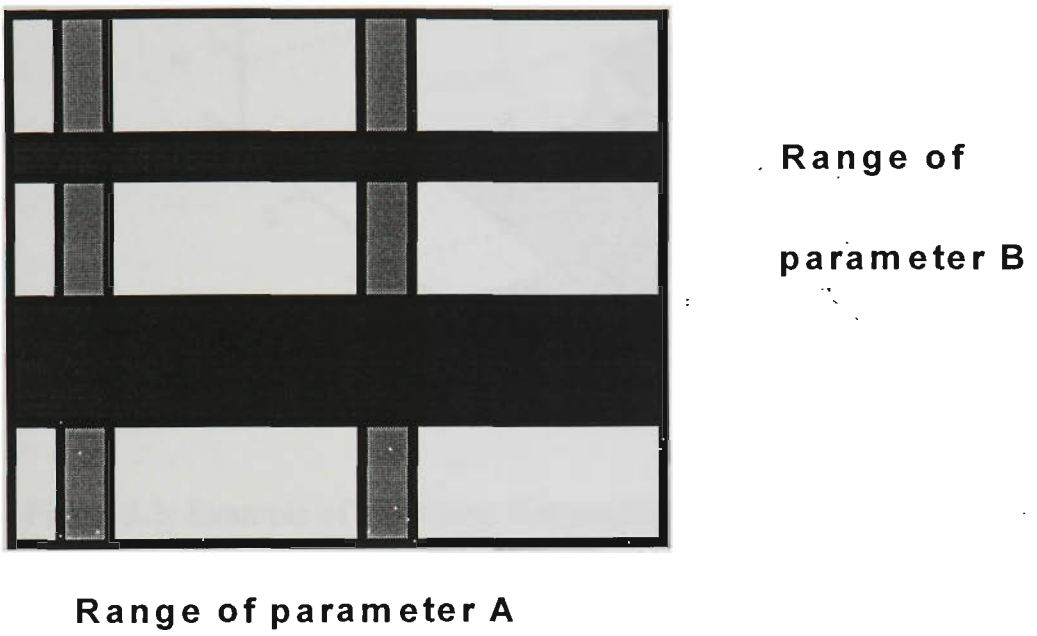


Figure 3.2: Simple crossover with a virtual alphabet. After the first few generations, the parameter values become restricted to the grey areas. Crossover can then only explore the intersection of these areas.

Goldberg then goes on to point out that for some functions, the initial, individual parameter fitness averages may not hold the global solution. Figure 3.3 is an example of a function that is designed to confuse such a real-coded GA. The central maximum is too small to have much effect on the global averages, so in the initial few generations the population settles into the two broad humps. Thereafter, no amount of crossover will reach the central peak. Goldberg describes such functions as being *blocked,* and argues that this is a potentially serious shortcoming of real-coded GAs.

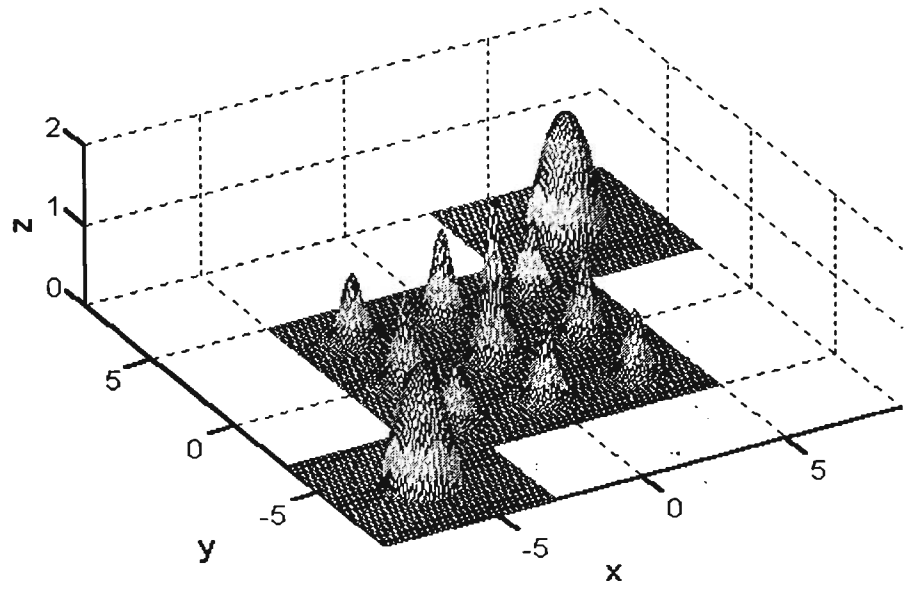


Figure 3.3: Example of a function that might, by Goldberg's analysis, cause problems for a real-coded GA.

However, this blocking presupposes the traditional form of crossover, that cuts strings between genes. Just as for mutation, we may ask what is an appropriate form of recombination for real-coded genes. Again, an obvious possibility works quite differently, cross the gene values to produce a new value somewhere in between those of the parents. Such a recombination works extremely well on the function shown in Figure 3.3, because the global maximum is conveniently situated in the centre of the range, between the two broad local maxima. Radcliffe [1990] reports that this recombination, that he calls *flat crossover,* along with a mutation operator that introduces new alleles at the end of a gene's range, works better than binary coding on the first four of DeJong's functions. Again this is not surprising, given their conveniently situated maxima. However, it should be clear that the form of recombination operator can dramatically affect the behaviour of the GA and has the potential to overcome Goldberg's concern about blocking.

3.5.5 Adaptive coding

A problem with coding real variables in a fixed number of bits is the limit on result's precision. When coding a problem for a GA, a sensible designer will restrict each parameter to a reasonable range, but there will still be some compromise between covering this range, and sufficient precision in whatever turns out to be the important part of it. Schraudolph and Belew [1990] have suggested a solution, which they call Dynamic Parameter Encoding (DPE). Genes initially code for the whole of the parameter range. However, when a gene converges sufficiently on one part of the range, the coding automatically "zooms in" on this area. The allocated number of bits is thus brought to bear on a reduced parameter range, increasing the available precision. The process may be iterated as the GA residences in on the best area of each parameter. This method means that each gene can use fewer bits, resulting in faster operation and convergence of the GA. Schraudolph and Belew report encouraging results, again using DeJong's test set, but there are attendant risks, since it is possible to narrow the search too quickly and miss something important. Thus, their performance on the multi-modal F5 was worse than without DPE, because there was insufficient resolution to find the correct hill to climb.

A more complex adaptive coding strategy has been suggested by Shaefer [1987]. His system, known as ARGOT, dynamically adjusts the parameter range coded by each gene. If the population clusters in a small part of the range, the boundaries are drawn in, much as in DPE. However, they may also move out if the population is widely distributed. If the population approaches one end of the range, the boundaries are shifted to re-centre it. The boundaries may also be "dithered", moved randomly by small amounts to effect a general mutation. Finally, the number of bits used may be changed, depending on the degree of convergence. Shaefer's results, for a number of function optimisations, indicate that the adaptive strategy compares well with a simple GA approach. However, it is obvious that there are many parameters associated with

decisions about changing the coding, and these are not specified. It seems likely that different adaptive strategies would be necessary for different problems. Referring to the range expansion, Schraudolph and Belew [1990] comment that they "believe it would be impossible to establish a well-founded, general trigger criterion for this operator". Nevertheless, such adaptive methods clearly have potential.

3.5.6 Conclusion

In this section we have looked at some possible codings and reproduction operators for numeric parameters. Many optimisation problems can be expressed purely in numerical terms, but there are also many that cannot, particularly order-based tasks such as the travelling salesman problem, while the coding for all examples used in this thesis is largely boolean. It is necessary that the reproduction operators can process similarities in the task in such a way as to combine useful building blocks. The real art of applying a GA to a task is therefore:

1. To identify the potential building blocks in a problem.

2. To design operators, principally recombination, that can process these building blocks.

Davidor [1991] has counted 56 different recombination operators in the literature. It may seem unfortunate that so much design effort is needed for each new application of a GA. It may also be difficult to identify what the suitable building blocks are.

3.6 Tuning GAs

Optimisation tasks of the sort considered in this thesis consist essentially of two parts: a search of the parameter space and hill climbing. There are numerous techniques for the second job (see Schwefel [1981] for a comparative review), one approach to the first is to restart repeatedly from different positions. The two phases are traditionally called

exploration and *exploitation.* GAs have the ability to do both. Some researchers seek to advance on the Holy Grail of a universal optimisation algorithm, that will cope with any fitness surface. DeJong's work aimed to provide a set of parameters for a GA that are reasonably robust, but such general algorithms will inevitably be beaten on any one problem by an algorithm that is tuned to the task.

The effect on the balance of exploration and exploitation of a number of GA parameters may be summarised:

Population size: A small population will tend to converge more rapidly.

Generation size: Changing only a fraction of the population each generation increases inertia, preventing convergence.

Mutation rate: Depends on the size of mutation. Big mutations encourage exploration, small mutations can be a means of hill climbing.

Recombination rate: Higher recombination rate encourages exploration while the population is diverse, but reduces it when the population has converged.

Selection pressure: If selection pressure is increased, either by scaling fitness values or by a high value for the scaling factor in rank-selection, hill-climbing will be encouraged.

Crowding: Maintains diversity, thus promoting exploration.

Elitist strategy: If the best individual from the previous population always survives, hill climbing is encouraged.

One possible method of matching these parameters to a given problem is to use a meta-level GA to tune them. The meta-GA specifies a population of GAs that act on the target problem. These are evaluated and the information used to improve the match of parameters to the task. CPU demand rather rules out this approach for any real-world

task, but Grefenstette [1986] was able to provide an improvement on DeJong's parameter set for his 5 functions.

Other workers have attempted introducing controls within the GA, which monitor convergence of the population and adjust control parameters accordingly [Shaefer, 1987; Whitley, 1990]. Goldberg argues against such "central authority" in his "Zen and the art of Genetic Algorithms" [Goldberg, 1989a], based on that it is not easy to establish robust criteria for making any adjustments. However, Ackley [1987] reports empirical success with an ingenious system he calls "Stochastic iterated genetic hillclimbing". This implements a kind of voting system, such that the algorithm climbs a hill until it effectively becomes bored with it, whereupon it goes off to find another hill to climb.

Tanese [1987] has suggested a multiple population GA, intended for running on separate processors, where the different populations have different parameter settings, the hope being that one will be near the ideal for the problem in hand.

An alternative approach to the tuning problem, mentioned in Section 3.5.6 is to adapt operator probabilities while the GA is running. One approach to this is to code, say, the mutation rate on the genetic string, where it will be selected along with the target parameters. Another, suggested by Davis [1989] takes a rather more interventionist approach of keeping a record of the improvement in fitness caused by each operator, and using this score periodically to adapt the probability of applying each operator. Montana and Davis [1989] use this method to good effect in evaluating potential operators for use in training neural network weights. It has the advantage that different operators may be of value at different stages of the search, the adaptive procedure allows those that are contributing most at any point to be selected. It also allows different operators to be compared.

3.7 Evaluating GAs

Traditionally the performance of GAs, and other optimisation techniques, has been reported in terms of *online* and *offline* performance. The latter is the average of the best individuals in each generation, the former refers to the average performance of all the strings since the start of run. This is of particular relevance when the system being optimised is a real-time one, like running a plant, and where getting it wrong costs something. Another measure is *best-yet,* simply the best performance so far seen. An alternative is the number of evaluations to achieve a given performance. That means evaluations, not generations, is important, since for any complex problem evaluations are expensive. Herdy [1991] reports results for a system with a variety of population sizes, from 1 to 40. The size 40 system requires 134 generations to completion, which is claimed to be better than the single string system, which takes 3072. The single string system requires fewer evaluations than any of the others, the result of a very simple hill-climbing task.

A complication arises because of the inherent noisiness of GAs. As has been noted, it is important to average over a number of runs. Even then, comparison is complicated by the typically non-normal distribution of results, especially with multi-modal functions. It seems that an algorithm that consistently finds the global maximum will be better than one that does so on average more rapidly, but sometimes fails altogether. In the end, it is important to specify the test conditions fully.

Chapter 4

The GAP model

This chapter describes the Genetic Algorithm Processor (GAP) model and its behaviour. Firstly the motivation for developing the GAP model, then a description of the basic GAP design and finally, a description of the pipelining in the design.

4.1 Justification for the GAP model

Genetic algorithms have been applied to many problems and have been recognised as a robust general-purpose optimisation technique. However in many optimisation problems in engineering software implementations are too slow to be useful.

Simple empirical analysis of many basic GAs indicates that a small number of simple operations and the fitness function occupy 80-90% of the total execution time. If m is the population size and g is the number of generations, a typical GA executes each of its operations mg times. For complex problems, large values of m and g are required, so it

is essential to make the operations as efficient as possible. Pipelining aids the efficient use of all of the hardware resources with maximum speed but it has its limitations and we cannot expect a very large speed improvement.

The structure and simplicity of a GA's computations provide a good basis for hardware implementation. Pipelining of a GA's operations is straightforward. For example, the selection, crossover, mutation and fitness operators can be easily chained together to form a coarse-grained pipeline. Section 4.4 provides a more detailed description of the pipelining available to the GAP.

Although the basic GA's operators are simple to implement in hardware, each problem needs some adjustments to the design. To use a general-purpose GA engine would require some special changes in the GAP hardware characteristics for a new application. The inflexibility of conventional hardware inhibits implementation and use of a general-purpose hardware-based GA. This is a major reason why hardware-based GAs have not been widely implemented to date.

The reprogrammability and low cost of FPGAs circumvent the problem of hardware inflexibility, while still maintaining a great speedup over software. The GAP design is implemented on FPGA technology and the external unit for evaluating the performance of each member is separate from the GAP. It will allow the user to apply the GAP model to a variety of different applications.

4.2 Basic Genetic Algorithm Processor design

Primarily there should be three connections between the GAP and the rest of the model (Figure 4.1).

1 - A Setup Unit (SU) which is implemented in software on a host computer. This unit is responsible for generating an intial population which, in the absence of precise information about the problem, is made up of random strings. It also assigns a low

fitness value to each of these strings and downloads this data to the GAP along with the starting parameters.

2 - A Memory Unit (MU) that stores genetic strings and fitness values. The GAP provides all necessary control signals to control the Memory Unit. It must be considered that the bit length of memory will vary with the application. For the experiments in Chapter seven, 8 bit, 16 bit, 24 bit, 32 bit and 48 bit length for memory are considered. The GAP was recompiled for each different bit length.

3 - A Fitness Unit (FU) which is external to the GAP and must be designed specially for the problem at hand. It accepts a string from the GAP as a controlling input then applies this to the problem, measures the performance (e.g. from the error signal) and returns a fitness value to the GAP. The rate of fitness evaluation is also problem dependent. High speed signal processing technology is normally required to evaluate and return each fitness value at a rate to match the presentation of strings from the GAP.

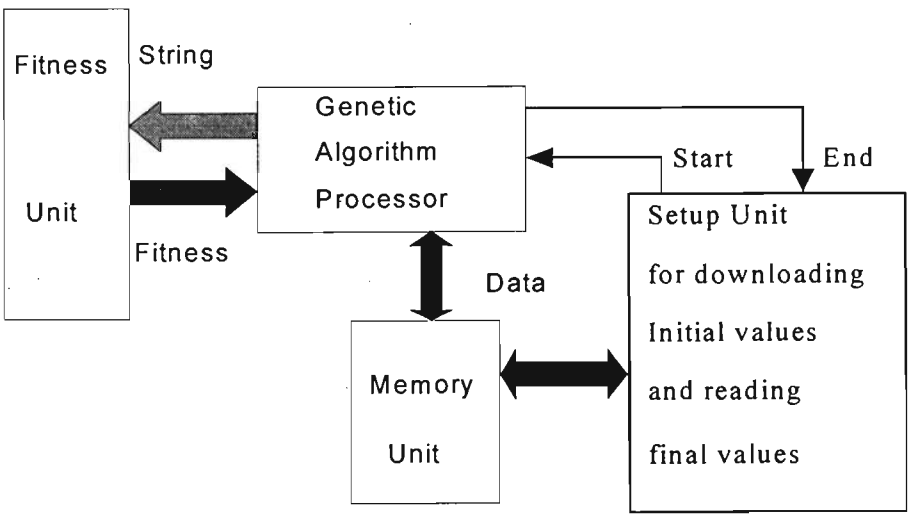


Figure 4.1: External connections to the GAP.

The process starts by filling the Memory Unit with the random population and other GA parameters from the Setup Unit. Then the Setup Unit sends a "Start" signal to the GAP. The GAP detects this signal and runs the GA using the parameters already in memory. After the set number of generations, the GAP sends an "End" signal to the Setup Unit which then reads the final population from the shared memory.

4.2.1 Development environment of the GAP

The GAP, like other hardware design projects, could have been developed at four different general levels: the behavioural level, the Register Transfer Level (RTL) the structural (gate or transistor) level, and the physical (mask) level [Weste and Eshraghian, 1993]. The GAP hardware was designed at the behavioural level using a Hardware Description Language (HDL), a high-level language used to specify hardware designs. The reasons for this choice are as follows.

• A HDL allows the designer to specify the behaviour of a complex system in terms of the actions performed by different modules and the connections between these modules. Contrast this with specifying the gate-level structure of the modules.

• A HDL allows for general (parameter-independent) designs to be created. The specific designs implemented from the general designs depend upon designer-specified parameters provided at implementation time. For example, many aspects of the GAP such as I/O bus size, storage facility size and pseudorandom number generator size depend on the size of each population member (m). Thus, some of the general design aspects can be declared as a function of m, which is provided at implementation time. This allows for quick reimplementation of the system if m changes.

The HDL chosen for this thesis is VHDL. VHDL was selected because of its widespread use and standardisation. The following additional tools were used in the development of the GAP.

• Design Architect from Mentor Graphics was used to define the behaviour of the GAP in VHDL code and compile the code.

• QuickSim II and QuickVHDL from Mentor Graphics was used to simulate the compiled VHDL code. The simulations were used to verify the design's correctness (Section 5.1) and analyse its performance (Section 5.2).

• AutoLogic from Mentor Graphics was used to synthesise the VHDL code to gate-level schematics composed of Xilinx components.

• FPGA Foundry from NeoCAD was used to map the Xilinx schematics to a file suitable for programming the Xilinx FPGAs.

4.2.2 A look at the overall design

A VHDL model of a general Genetic Algorithm was created. The model allows the GAP's user to choose several parameters which are a subset of the general GA parameters described in Appendix A. These user-controlled parameters are:

• the initial population size and its members,

• the number of generations in the GAP run,

• the initial seed for the pseudorandom number generator,

• the mutation and crossover probabilities.

Values for these parameters would be selected by the user in Setup Unit which would then pass them to the Memory Unit to initialise and start the GAP.

Other GA parameters, such as the length of the member strings and the coding scheme, would be indirectly specified according to the fitness function. These parameters are determined by the way the fitness function decodes and evaluates the population

members. Note that other stopping criteria, other than fixed number of generations, can be easily implemented. After a certain number of generations, which will be specified by a GAP parameter, the GAP looks at the final population and determines if the stopping criteria are met. If so, the GAP halts and reports the final population to the user. Otherwise the GAP continues from where it stopped. Thus, nearly all the general GA parameters listed in Appendix A can be directly or indirectly specified in the GAP.

4.2.3 The modules and their functions

The modules in Figure 4.2 are based on the GA operators defined in Goldberg's simple genetic algorithm (sGA) [Goldberg, 1989b]. Many other GA models exist, but the GAP was based on the sGA because the sGA is simple to understand and implement. The sGA is also a well-known GA implementation. The basic structure of the sGA is similar to description in Section 3.1. More detail concerning the functionality of the GAP modules and their sGA counterparts is given below.

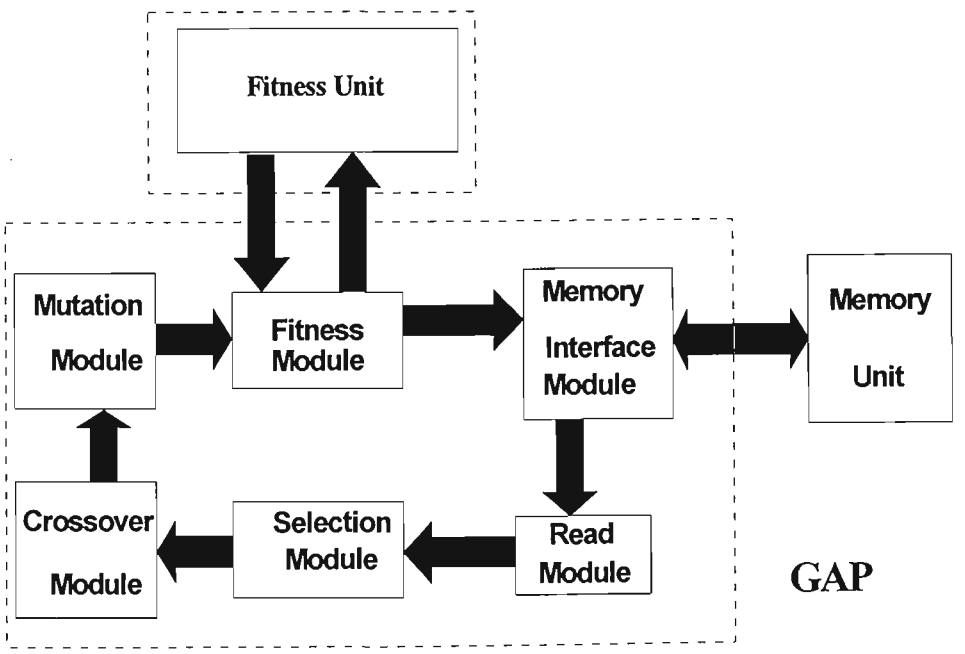


Figure 4.2: Module-level of the overall GAP model.

The GAP modules operate concurrently and together form a coarse-grained pipeline. All modules are written in VHDL and are independent of the operating environment and implementation technology (e.g. Xilinx FPGAs or fabricated chips) except for the Memory Interface Module. The functionality of this module varies according to the physical memory attached to it and the desired interface between the GAP and the user.

The basic functionality of the GAP design in Figure 4.2 is as follows.

1. After all the parameters have been loaded into the shared memory, the Memory Interface Module (MIM) receives a "Start" signal from the Setup Unit. The MIM acts as the main control unit of the GAP and is the GAP's interface to the memory.

2. The MIM informs the Fitness Module (FM), Mutation Module (MM), Crossover Module (CM), Selection Module (SM), Read Module (RM), and the Pseudorandom Number Generator (PNG) that the GAP is to begin execution. Each of these modules requests necessary parameters from the MIM, which fetches them from the appropriate places of the shared memory.

3. The Read Module starts the pipeline by requesting population members and their fitness values from the MIM and passing them along to the Selection Module.

4. Whenever the Selection Module receives a new candidate A from the RM, it determines if A is to be selected (the selection process is described in more detail below). If not, it waits for a new member to be sent by the RM. Otherwise, it stores A and proceeds to select a second member B in the same manner. After B has been selected, the pair (A and B) are sent to the Crossover Module for further processing. Once the pair is sent, the SM resets itself and restarts the process to select another pair.

5. When the Crossover Module receives the pair of members A and B, it decides, using a random value from the PNG, whether to perform crossover. When completed, the new members A' and B' formed by crossover of A and B, are sent to the next module.

6. When the Mutation Module receives A' and B', it decides, using a random value, whether to perform mutation. When completed, the new mutated members A'' and B'' are sent to the Fitness Module for evaluation.

7. The Fitness Module accepts A'' and B'' and evaluates them in an external Fitness Unit. Ideally the FU completes its evaluation in only one clock cycle. After evaluation, the FM writes the new members to memory through the MIM. The FM also maintains some records concerning the current state of the GAP such as the sum of fitness values in the current population and the number of generations. These records are used by the SM to select new members and by the FM to determine when the GAP run is completed.

8. At the end of the GAP run the FM informs the MIM of completion which in turn stops the GAP modules and sends the "End" signal to the Setup Unit.

Each of the GAP modules is described in more detail below.

4.2.3.1 Pseudorandom Number Generator (PNG)

The Pseudorandom Number Generator generates a sequence of pseudorandom bit strings based on the theory of linear Cellular Automata (CA). CA was shown by Serra [1990] to generate better random sequences than Linear Feedback Shift Registers (LFSRs) which are commonly used as pseudorandom number generators. The CA used in the PNG consists of 16 alternating cells that change their states according to rules 90 and 150 as described in [Wolfram, 1984]:

Rule 90: $s(i)+ = s(i-1) \oplus s(i+1)$

Rule 150: $s(i)+ = s(i-1) \oplus s(i) \oplus s(i+1)$.

Here s(i) is the current state of site (cell) i in the linear array, s(i)+ is the next state for s(i), and $\oplus$ is the exclusive OR operator. Thus in Rule 90 a cell is updated according to the inputs from its neighbours while in Rule 150 each cell also considers its state when

updating. It has been shown that a 16-cell CA whose cells are updated by the rule sequence 150-150-90-150...90-150 produces a maximum-length cycle. It cycles through all possible 2^{16} bit patterns except the all 0s pattern, and has more randomness than an LFSR of corresponding length. This scheme is implemented in the PNG.

The PNG is a key component of the GAP model and its output is used by three GAP modules. The PNG supplies pseudorandom bit strings to the Selection Module for scaling down the sum of fitness values. This scaled sum is used when selecting pairs of members from the population. The PNG also supplies pseudorandom bit strings to the Crossover Module and Mutation Module for determining whether to perform crossover and/or mutation and for choosing the crossover and mutation points.

4.2.3.2 Memory Unit (MU)

The memory is not truly part of the GAP model, but it is presented here for completeness. It is assumed that some memory is available to the GAP model and that its specifications are known to the Memory Interface Module. The memory is shared by the MIM and Setup Unit (Figure 4.1) and acts as the communication medium between them. Before the GAP run, the Setup Unit writes the GA parameters specified in Section 4.2.2 into the memory and signals the MIM. After receiving the signal, the MIM distributes the parameters to the appropriate modules. During the GAP run, the population members are read from and written to the memory by the MIM. When the GAP run is finished, the memory holds the final population which is then read by the Setup Unit.

The important thing about memory is the word size (bit length) of memory and the capacity. For simple operation of the GAP the memory capacity is not very important. Most GA algorithms are operate with a population of less than 200 and the recommended population is about 100 members. On the other hand the bit length of memory is very important and affects the overall performance of the model. Memory bit

length will be discussed further in Chapter 7 and 8. For the GAP model, experiments were conducted with 8, 16, 24, 32 and 48 bit memories.

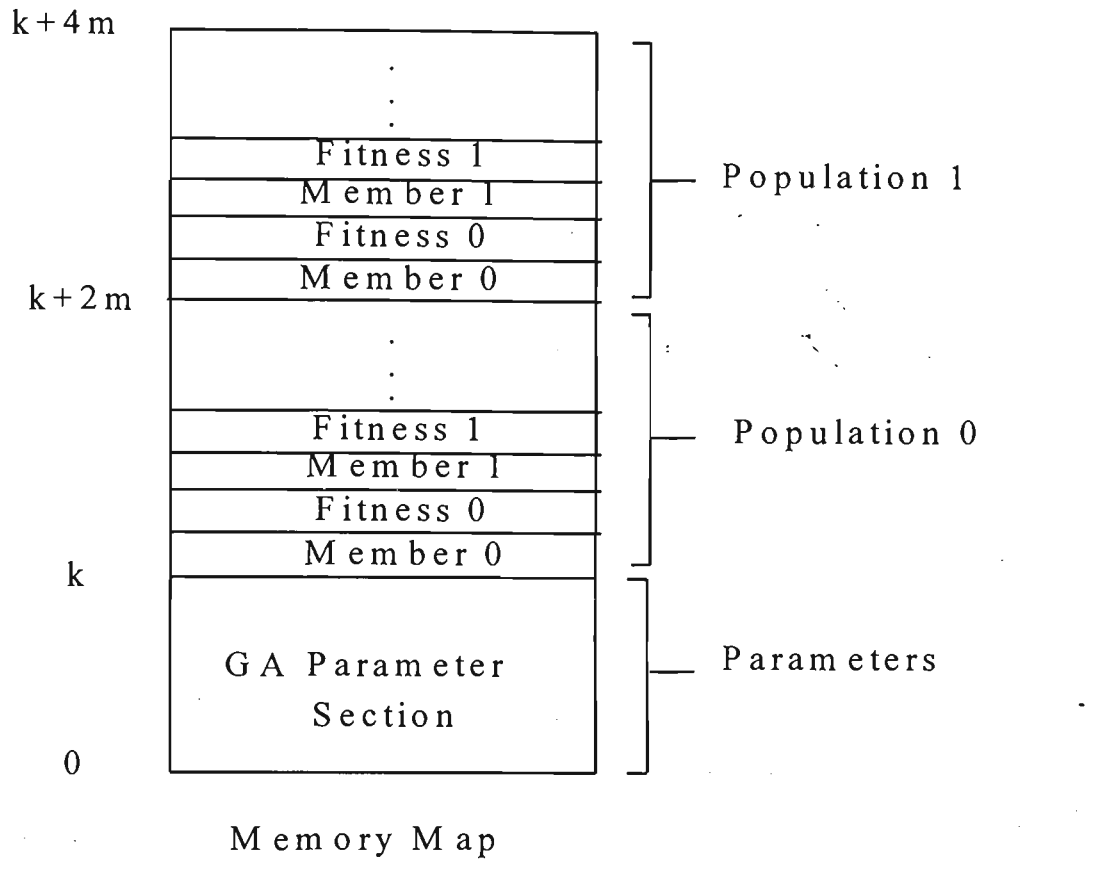


Figure 4.3: The Memory Unit map.

Figure 4.3 shows the contents of the Memory Unit during GAP processing. In the memory there are *k* GA parameters and two populations each with *m* members. The GA parameters are located in the bottom of the memory. They are initialised by the Setup Unit in the beginning. The next section is the first population which will be initialised with random member strings by the Setup Unit. The top section, the second population, will be filled during operation of the GAP. Each member has two data items: the member bit string and the fitness value of the member. For example, for a population of 32 members a total of, k+2*(32+32)= k+128 locations are required.

4.2.3.3 Memory Interface Module (MIM)

The Memory Interface Module is the only module in the GAP model which has knowledge of the GAP's environment. It provides a transparent interface to the memory for the rest of the model. At startup, the MIM acts as a control module and instructs the other modules to initialise. During initialisation, the other modules send requests to the MIM for user-specified GAP parameters. These requests involve a simple handshaking protocol initiated by the requesting modules providing a coded address to the MIM. The MIM then converts this coded address to a physical memory address. The parameter received from memory is then passed on to the requesting module. When the GA run is complete, (as signalled by the Fitness Module), the MIM stops the system and informs the Setup Unit of completion.

As described in Section 4.2.3.2, two copies of the population are maintained. One copy P_t represents the population at time t and the other copy P_{t+1} represents the population at time $t + 1$. During a run the RM reads from P_t and the FM writes to P_{t+1}. In memory, P_t and P_{t+1} are stored in separate memory blocks, labelled P0 and P1 (for population 0 and population 1 in Figure 4.3). For ease of implementation, the mapping h: $\{ P_t, P_{t+1} \} \rightarrow$ {P0, P1 } alternates between generations by the MIM, e.g. if h(P_t) = P0 in the current generation, then h(P_t) = P1 in the next generation. At any given time the FM knows (by MIM) which of P_t and P_{t+1} is located in P0 and P1.

To make the GAP independent of its operating environment, the RM and FM only know the indexes of the population members they want to read or write rather than the actual addresses. Not requiring knowledge of the actual addresses simplifies porting the GAP to other operating environments. Therefore, it is up to the MIM to translate the indexes from the RM and FM to the correct addresses by adding the appropriate base address (i.e. P0's base address or Pl's base address) to the indexes. A signal line from the FM specifies which population is being read from and which is being written to. This signal value is toggled by the FM after every generation.

The MIM was designed to be the only GAP module required to be aware of the system's execution environment (e.g. population index). Thus design and technology changes are easier to accommodate and only the MIM need be modified to meet changed specifications.

4.2.3.4 Read Module (RM)

The Read Module constantly cycles through the current population and passes the members on to the Selection Module. The roulette wheel selection process used by the sGA [Goldberg, 1989b] (Section 4.2.3.5) is independent of the order that the population members are searched. So constant cycling through the population works as well as the sGA implementation. The Read Module sends the index of a population member to the MIM, reads in the member, then increments the index and reads in the fitness value. The member and fitness value are then passed to the Selection Module. The index is then incremented modulo the population size so the next population member will be requested from the MIM. This process continues until the GA run is complete when the MIM stops all the modules.

4.2.3.5 Selection Module (SM)

The GAP's selection method is similar to the implementation of roulette wheel selection found in the sGA. Each time a new population member is required, the following process is executed by the Selection Module.

1. Using a uniform real random number $rand \in [0,1]$, scale down the sum of the fitness values of the current population to get a fitness threshold:

$$Fit\_Threshold = rand * \sum (\text{all fitness values}) \qquad (4.1)$$

2. Starting at population member 0, examine the members in the order they appear in the population.

3. Each time a new member is examined, accumulate its sum in a running sum of fitness values *Fit_Sum*. As soon as *Fit_Sum* ≥ *Fit_Threshold* the member under examination is selected. Otherwise the next member is examined (Step 2).

The Selection Module executes the roulette wheel selection process similar to the sGA, but it selects a pair of population members *A* and *B* simultaneously rather than one member at a time. It receives the sum of the fitness values of the current population members from the Fitness Module and scales down this sum by two random values provided by the Pseudorandom Number Generator. These two scaled sums, *SumA* and *SumB*, are stored for future use. Upon receipt of a population member *M* and its fitness from the Read Module, *M*'s fitness is accumulated in a running sum *SumR*. If *SumR* surpasses *SumA* at this time, then *M* is latched as the selected member *A*. Selected member *B* is chosen in the same fashion. Since the values *SumA* and *SumB* are determined by independent random numbers, selection of *A* and *B* are independent, concurrent processes.

Once *A* and *B* are selected, they are sent to the Crossover Module for further processing. After sending *A* and *B*, the Selection Module resets *SumR* and scales down the sum of fitness values by two new random values to generate new values for *SumA* and *SumB*. When an entire generation has been selected, the FM resets the Selection Module so that it can use the new sum of fitness values in its calculations. This process repeats until the Selection Module is halted by the MIM at the end of the GAP run.

4.2.3.6 Crossover Module (CM)

The Crossover Module waits for a new pair of members *A* and *B* from the Selection Module. It then reads a pseudorandom unsigned binary integer *rand1* from the PNG and compares it to the crossover probability *Pc* (also interpreted as an unsigned binary integer) that it received from the MIM as a user-specified parameter. If *rand1* < *Pc* then crossover is performed between *A* and *B* forming two new members *A'* and *B'*. A new

pseudorandom bit string *rand2* is used as an index of A and B which indicates the crossover point. If $rand1 \geq Pc$ then crossover is not performed and $A' = A$ and $B' = B$.

Because of this implementation, the crossover step takes only one clock cycle and arrays of multiplexers and inverters are essentially all that are required for doing the job. In contrast, the sGA has to cycle through each bit of the new member and copy the result to a new location that forces it to spend more time in the crossover operation.

4.2.3.7 Mutation Module (MM)

The Mutation Module waits for two new members A' and B' from the Crossover Module. It then reads a pseudorandom unsigned binary integer *rand3* from the PNG and compares it to the mutation probability Pm that it received from the MIM as a user-specified parameter. If $rand3 < Pm$ then mutation is performed on a single bit in A'. The mutated bit is selected by another pseudorandom bit string *rand4* which acts as a pointer into the bit string A'. The final new member is A''. The same steps are repeated for the B' member to produce B'' and the new members are sent to the Fitness Module. Here the GAP differs from the sGA in that the sGA makes a decision about mutating each bit in A', effectively increasing the mutation probability. The GAP makes only one mutation decision for each new member and chooses the mutation point at random. This implementation decision was based on simplifying the design and speeding up the operation.

The mutation step takes only one clock cycle in this implementation. In the sGA, mutation is slower because it is carried out bit by bit. On the other hand, in the GAP mutation will be decided for a new member and finally only one bit per member will be changed.

4.2.3.8 Fitness Module (FM)

The purpose of the Fitness Module is to evaluate the population members mated by the Crossover Module and Mutation Module and insert them into the new population. Although there are control signals for asynchronous evaluation of the fitness value, ideal performance is only achieved if the fitness value can be computed in one clock cycle per member.

When the FM receives a pair of members from the Mutation Module, it evaluates their fitness using the Fitness Unit and then writes the new members and their fitness values to the appropriate memory location with the help of the Memory Interface Module. The FM then waits for the MM to send two more members.

The FM also maintains a running sum of the fitness values for reporting to the Selection Module after each generation. The Selection Module will then scale down that sum with a random value and use the scaled sum in the selection process. Additionally, the Fitness Module maintains a record of how many generations remain to be tested and how many members still need to be chosen in the current generation. When the last generation is complete, the FM notifies the Memory Interface Module of GAP completion.

4.2.3.9 Fitness Unit (FU)

The Fitness Unit accepts a genetic string, calculates the fitness value and returns the result to the Fitness Module. The actual process of evaluation of the fitness for each string will depend on the problem. One possible configuration which would suit the application of a GAP to some engineering problem such as instrument tuning is shown in Figure 4.4. The Transducer (e.g. a Digital to Analog Converter (DAC) or a state machine) converts the genetic string into suitable signals for the Unit Under Test (UUT) which is the system to be optimised. A Fitness Measurement (e.g. an Analog to Digital

Converter (ADC) or signal processor) converts and evaluates the responses of the UUT and delivers the result as a digital fitness value.

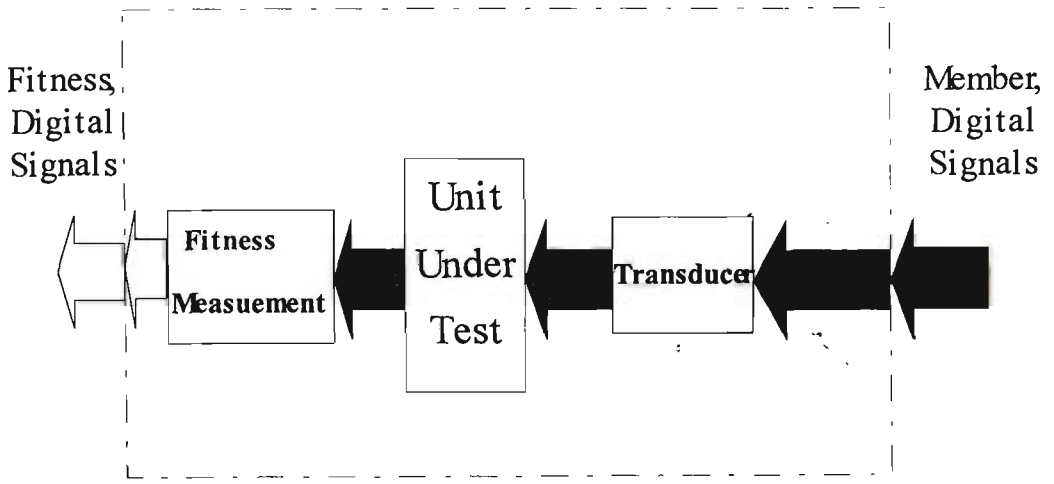


Figure 4.4: Typical Fitness Unit.

Ideally the Fitness Unit should complete the fitness evaluation within one clock cycle after setting up the input value. This becomes quite difficult to achieve in most practical situations (see Chapter 10). It is possible to extend this time, but it will decrease the overall performance of the GAP. This means the GAP is best suited to fast response environments. There are some control signals between the Fitness Unit and FM to synchronise the GAP with a slower Fitness Unit.

4.3 Design parameters

Since the modules of the GAP model were written entirely in VHDL, specific aspects of the design such as I/O bus size or storage facility size can be specified in terms of parameters which can be easily changed when the need arises (Section 4.2.1). The interesting parameters of the GAP are n: the bit length of the population members, f: the bit length of the fitness values, m: the size of the population and *ngen*: the maximum number of generations. Other parameters affect the GAP, but they address particular

implementation details and are not as interesting as those listed above. These parameters are specified at VHDL compile time and should not be confused with the GA run time parameters. They include *addressm*: width of address bus for the Memory Unit, *valuem*: width of data bus for Memory Unit and *randomsize*: size of the maximum random number in the GAP.

All of these parameters are defined in a single package file called a *geneparam* in the VHDL code. When the need arises for changing the design for a special purpose (e.g. new limitations are applied by the Fitness Unit), all that is necessary is to change the appropriate values in the *geneparam* package and recompile the VHDL code. Although some facilities exist for parameterising and scaling of designs at lower levels of design entry (e.g. at the gate level), they are not as easy to use or as intuitive as using the VHDL packages. It should be noted, however, those design entry methods are only feasible if HDL synthesis tools are available. If one wants to go beyond design simulation to implementation without using HDL synthesis tools are available, then a lower level of design entry is necessary.

4.4 Pipelining

As mentioned in Section 4.1, GA operations can be easily pipelined and parallelled. This offers the GAP a great advantage over sequential software GA implementations.

The design in Figure 4.1 is a coarse-grained pipeline. The Read Module gets a new population member from the Memory Interface Module and passes it to the Selection Module. When the SM has received a pair of members it passes the pair to the CM through a handshaking protocol and immediately restarts the selection process. After crossover and mutation are complete (in CM and MM), the MM passes the new members to the Fitness Module by a hand-shaking protocol and CM looks to the Selection Module for the next pair. Finally, when the FM receives the new members, it

evaluates them and sends the new members and their fitness values to the MIM for writing to memory.

Thus GA operations are executed in a pipelined fashion and a significant speedup over software implementations is expected.

Chapter 5

Design verification and analysis

This chapter starts with tests to verify the functionality of the model. The GAP is applied to find an optimum point on each of two different mathematical function surfaces. This is followed by a mathematical analysis of the design using techniques described by Kenyon et al. [1993]. This includes an analysis of the pipelines to identify the bottlenecks which concludes with a series of tables and graphs comparing the number of clock cycles consumed in GAP simulations with that predicted from the analysis.

5.1 Verification of correct functionality

Two levels of functional verification were used. First each module was tested to confirm correct operation under all conceivable conditions. For each module a set of test input vectors is selected and the output vector is examined carefully to confirm correct

functionality. The second level of functional verification involved simulating the GAP on different fitness functions to see how well the functions were optimised. The modules were connected in the configurations of Figures 4.1 and simulated on the two fitness functions. During these simulations each module was closely examined to confirm correct functionality. Completion of these simulations verified the correctness of the operation of the modules and their intercommunication. In both examples, the population size was 32, the size of each member was 16 bits and the maximum width of the fitness values was 8 bits.

In the first example, the GAP was tested with the following mathematical fitness function [Davis, 1990].

$$F(X,Y) = 0.5 - \frac{\left(\mathrm{Sin}\sqrt{(X^2+Y^2)}\right)^2 - 0.5}{\left(1.0+0.001*(X^2+Y^2)\right)^2} \qquad (5.1)$$

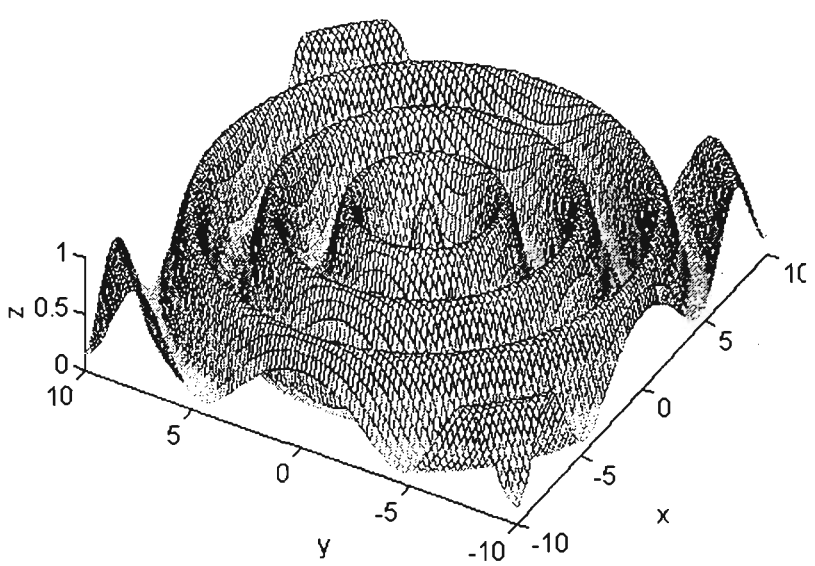


Figure 5.1: Problem surface for (5.1).

This is typical of functions used in testing genetic algorithms and other hill climbing strategies. This function is represented by the surface shown in Figure 5.1 (i.e. the

fitness landscape). This function has a global maximum at the point (0,0) and a series of maxima which are located on the ridges and peaks of the surface near:

$$\mathbf{R} = \sqrt{\left(\mathbf{X}^2 + \mathbf{Y}^2\right)} = \pi, 2\pi, \ldots \tag{5.2}$$

These kinds of problems are very difficult for conventional gradient ascent methods such as the Newton-Raphson method. The search for the global maximum usually ends in becoming trapped in one of the local maxima. Because genetic algorithms use a population of attempted solutions which are randomly generated, a (nearly) correct solution can be obtained in most cases. The results of simulation are shown in Figure 5.2. The fitness functions were optimised over the discrete domain

$$D = \{x \mid -10 \leq x \leq 10\}\ ,\ \{y \mid -10 \leq y \leq 10\}. \tag{5.3}$$

The fitness value varies between 0 and 1. These results are averaged over 10 runs for each test.

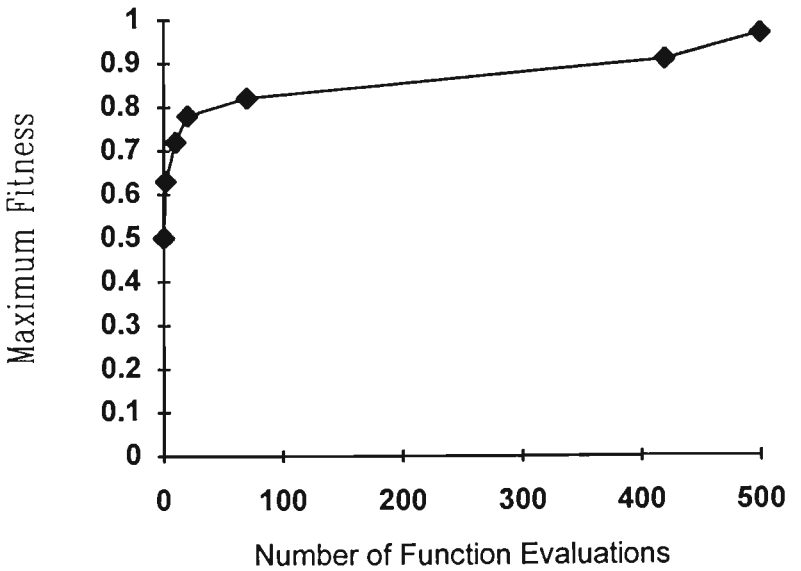


Figure 5.2: The results of example 1.

In the second example, another mathematical fitness function (5.4) was used. This is also a typical function used as benchmark for quasi-discrete problem surfaces for comparing genetic algorithms.

$$F(X,Y) = e^{(-0.2*\sqrt{(X^2+Y^2)}+3*(Cos(2*x)+Cos(2*y)))} \tag{5.4}$$

The surface of this function is shown in Figure 5.3.

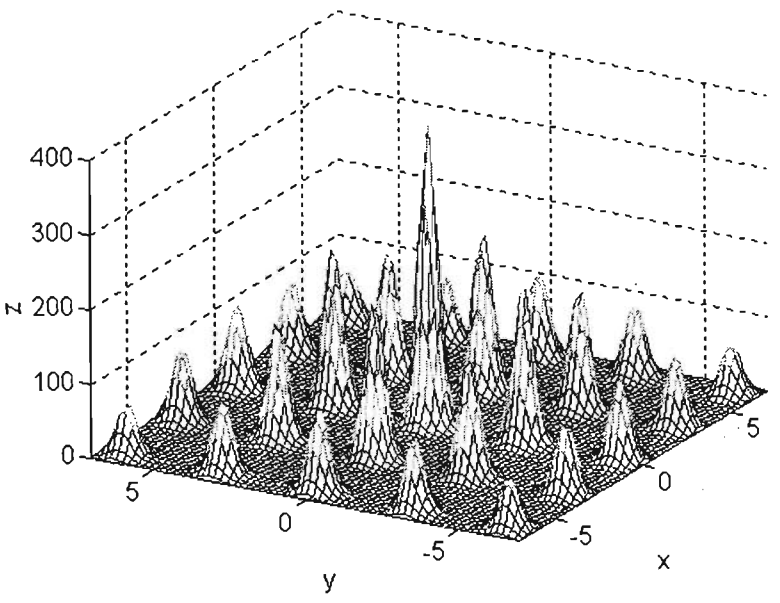


Figure 5.3: Problem surface for (5.4).

This function has a global maximum at the point (0,0) with the value about 400 and a series of maxima which are located near

$$\mathbf{X} = \pi, 2\pi, \ldots \qquad \mathbf{Y} = \pi, 2\pi, \ldots \tag{5.5}$$

This function has peaks at (π,0), (0,π), (0,-π), (-π,0) and GAs may become trapped on them. The results of simulation are shown in Figure 5.4. These results are average over 10 runs and the fitness function was optimised over the discrete domain

$$D = \{x \mid -10 \leq x \leq 10\}\ ,\ \{y \mid -10 \leq y \leq 10\}. \tag{5.6}$$

These two examples show that the GAP is capable of optimising some difficult functions.

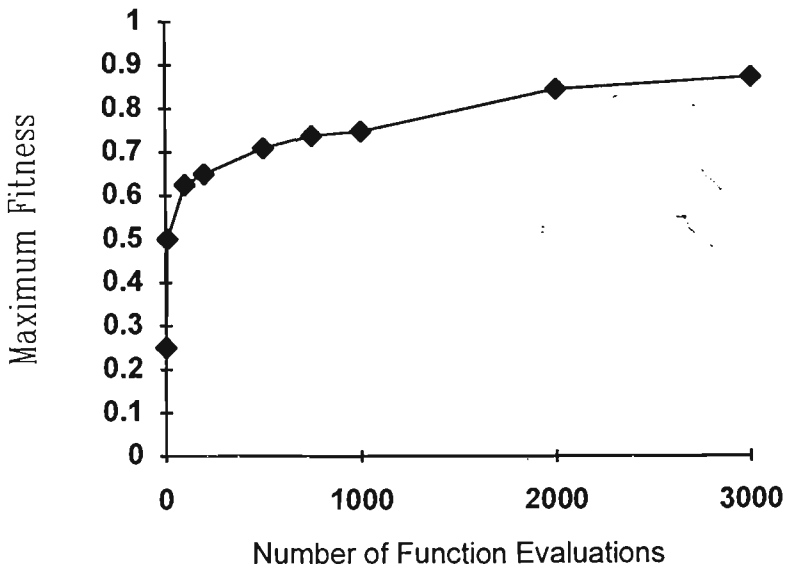


Figure 5.4: The normalised results of example 2.

5.2 Mathematical analysis

The modules in Figures 4.1 were analysed to determine the parameters which impact on asynchronous pipeline performance. These parameters are defined in [Kenyon et al., 1993] as follows.

1. *i*: The stage number. Each stage corresponds to the operation of one of the GAP modules (Table 5.1).

Stage number (*i*)	0	1	2	3	4
Module	Read	Selection	Crossover	Mutation	Fitness

Table 5.1: Each stage number is matched with one module of the GAP.

2. s_i: The actual service time of pipeline stage (module) i is the amount of time stage i takes to receive a message at its inputs, process it and send the output to the next stage.

3. F_i: The flow rate of stage i is the number of messages arriving at stage i during the entire run.

4. S_{norm_i}: The normalised service time of stage i is defined as $S_{norm_i} = \frac{s_i F_i}{F_{out}}$ where F_{out} acts as a normalising factor.

5. F_{out}: The flow rate out of the pipeline.

The analysis is designed to calculate the flow rate (F_i), defining service time (s_i) and the normalised service time (S_{norm_i}) for each module and hence finds the bottleneck of the system. The other parameters in the analysis are:

6. g: The number of generations in the GAP run.

7. m: The population size.

8. T: The total number of clock cycles in the entire GAP run.

9. r: The total number of cycles to read from the memory.

10. w: The total number of cycles to write to the memory.

11. d: The number of delay cycles in the Fitness Unit.

To retain technological independence, all formulae and simulation results are given in terms of clock cycles. In this analysis, $F_{out} = mg/2$ because mg members are selected in the GA run a pair at a time. Once S_{norm_i} was determined for each stage, all the S_{norm_i}'s were compared. The stage with the highest S_{norm_i} will be the pipeline bottleneck.

Although not explicitly mentioned, the analysis of the Read Module (Section 5.2.1) and the Fitness Module (Section 5.2.4) involve the Memory Interface Module. This is because the Read Module and the Fitness Module's service times partially depend on communication overhead with the MIM and the MIM's time to read from and write to the memory. Thus, the MIM can be thought of as partially merged with the Read Module and partially merged with the Fitness Module.

After analysing each module, the GAP will be simulated to determine the values of s_i and F_i for each module in practice. The following sections present the results of the analyses and simulations of the GAP modules.

5.2.1 Read Module analysis

The simple request-acknowledge handshaking protocol between the Read Module and the Memory Interface Module requires 6 clock cycles to communicate a request for a new population member and to receive that member. It also requires r cycles for the Memory Interface Module to read the member from memory. Thus, it typically takes $(6+r)$ cycles to fetch a member from memory. For each member the GAP needs to read the next location for fitness value as well, therefore the total number of cycles is $(12+2r)$. However, if the Fitness Module requests access to the MIM to write a new population member to the memory, it will receive priority.

To keep the GAP simple no pre-emption is supported, so if the FM has a lock on the MIM, the RM is blocked. The FM's access to the MIM could block the RM between 1 and $(4+w)$ clock cycles, all equally probable. To find a weighted average of these delays, first note that the FM will make exactly $2mg$ write requests during the entire GAP run. In each generation the GAP writes every pair (member+fitness) back to the memory. Then the probability of an FM write request at a given clock cycle is $2mg/T$. This means that at the time of a RM read request, there is a $2mg/T$ probability of an

additional delay of *(4+w)* cycles, a *2mg/T* probability of an additional delay of *(3+w)* cycles, etc. The weighted average of these possible delays is:

$$\frac{\sum_{i=1}^{4+w} 2img}{T} = \frac{(5+w)(4+w)mg}{T} \tag{5.7}$$

The delay should be multiplied by two because the Read Module needs to read two values from memory. Therefore the average service time of the RM is:

$$s_0 = 12 + 2r + 2\frac{(w^2 + 9w + 20)mg}{T} \tag{5.8}$$

The flow rate of the RM is the number of messages it generates. This can be approximated by multiplying the number of pairs of members that must be selected *(mg/2)* by the average number of members that the Selection Modules must receive to select one pair *(m/2)*:

$$F_0 = \frac{m^2 g}{4} \tag{5.9}$$

The normalised service time for the RM (S_{norm_0}) is the product of $s_0 F_0$ to the normalising factor F_{out}, the total flow out of the pipeline, which is equal *(mg/2)*. So

$$S_{norm_0} = \frac{s_0 F_0}{F_{out}} = s_0 \frac{m}{2} = \frac{m}{2}\left(12 + 2r + 2\frac{(w^2 + 9w + 20)mg}{T}\right) \tag{5.10}$$

5.2.2 Selection Module analysis

Since the input flow differs from the output flow, the service time s_1 for the Selection Module is a weighted average of the service time per input ($S_{1\,in}$) and the service time per output ($S_{1\,out}$). Accumulation of fitness values and the necessary comparisons to check for selection are easily done in one clock step, so $S_{1\,in} = 1$. However, $S_{1\,out}$ is more

complex. In the best case, the handshaking and transmission between the Selection Module and the Crossover Module is 7 cycles.

If the GAP is operating at maximum efficiency, the FM would be always busy. But if the Fitness Module is blocked by the Read Module, then the FM blocks the CM, which in turn blocks the Selection Module which is ready to send output. The Fitness Module has a probability of

$$1 - s_0 F_0 = 1 - \frac{m^2 g s_0}{4T} \tag{5.11}$$

of being busy. If the Selection Module is ready, it will take an average of $\frac{(s_4 - 4 - 2d)}{4}$ cycles (Section 5.2.5) to be served by the Fitness Module (s_4 is the service time for FM). So the total additional delay is:

$$\frac{(s_4 - 4 - 2d)}{4} \times \left(1 - \frac{m^2 g s_0}{4T}\right) \tag{5.12}$$

which leads to the following formula for the service time per output:

$$S_{1\,out} = 7 + \frac{(s_4 - 4 - 2d)}{4} \times \left(1 - \frac{m^2 g s_0}{4T}\right) \tag{5.13}$$

Since one output message is generated for approximately every m input messages, averaging $S_{1\,in}$ and $S_{1\,out}$ yields the actual service time for the Selection Module as:

$$s_1 = \frac{m S_{1\,in} + 1 \times S_{1\,out}}{m} = S_{1\,in} + \frac{S_{1\,out}}{m} \tag{5.14}$$

The Selection Module's flow rate is:

$$F_1 = \frac{m^2 g}{4} \tag{5.15}$$

which is the total number of messages expected to activate the Selection Module during the GAP run. Finally, the Selection Module's normalised service time is given by

$$S_{norm_1} = \frac{s_1 F_1}{F_{out}} = s_1 \frac{m}{2} \tag{5.16}$$

Applying (5.14), it produces:

$$S_{norm_1} = \frac{m}{2} S_{1in} + S_{1out} = \frac{m}{2} + 3.5 + \frac{(s_4 - 4 - 2d)}{8} \times \left(1 - \frac{m^2 g s_0}{4T}\right) \tag{5.17}$$

5.2.3 Crossover Module analysis

The Crossover Module normally takes 7 cycles to process input and transmits the result to the Mutation Module. If however, the MM is blocked while waiting to send to the FM (see Section 5.2.2 for more detail), then an added delay is incurred because the GAP has no buffering in its modules. Due to the nature of the GAP, the added delay is:

$$\frac{(s_4 - 4 - 2d)}{4} \left(1 - \frac{m^2 g s_0}{4T}\right) \tag{5.18}$$

This is because over the entire run (T cycles long), the Fitness Module will delay all previous modules with a delay of $\frac{(s_4 - 4 - 2d)}{4}$ cycles with the probability of $\left(1 - \frac{m^2 g s_0}{4T}\right)$.

Using the above probability, the service time for the CM is:

$$s_2 = 7 + \frac{(s_4 - 4 - 2d)}{4}\left(1 - \frac{m^2 g s_0}{4T}\right) \tag{5.19}$$

The CM's flow rate is:

$$F_2 = \frac{mg}{2} \tag{5.20}$$

and the CM's normalised service time is:

$$S_{norm2} = \frac{s_2 F_2}{F_{out}} = s_2 \frac{gm}{2} \frac{2}{gm} = s_2 \tag{5.21}$$

5.2.4 Mutation Module analysis

The Mutation Module normally takes 7 cycles to process input and transmits it to the Fitness Module. If the FM is blocked while waiting to send to the MIM, then an added delay is incurred. The delay for this module is exactly equal to that of the Crossover Module:

$$s_3 = 7 + \frac{(s_4 - 4 - 2d)}{4}\left(1 - \frac{m^2 g s_0}{4T}\right) \tag{5.22}$$

and the normalised time is:

$$S_{norm3} = \frac{s_3 F_3}{F_{out}} = s_3 \frac{mg}{2} \frac{2}{mg} = s_3 \tag{5.23}$$

5.2.5 Fitness Module analysis

When the Fitness Module receives input in the form of strings A'' and B'', the following events occur during processing.

1. Evaluate A'' and B'', accumulate their fitness values and request access to the Memory Interface Module. The delay time for this step is $(4+2d)$ cycles where d is the delay cycle in the Fitness Unit.

2. Wait for the MIM to acknowledge the request. For this step the delay time is between 1 and $(6+r)$. The potential additional delay is due to the lock that the RM may have on

the MIM. In the worst case, the FM requests MIM access immediately after the RM locked the MIM. This worst case would cause a delay of *(6+r)*. The average delay is:

$$\frac{\sum_{i=1}^{r+6} i}{r+6} = \frac{r+7}{2} \tag{5.24}$$

3. Receive the MIM's acknowledgment, send *A''*, wait for the MIM to write *A''*, notify the FM and issue the next request: *(4+w)* cycles.

4. Steps 2 and 3 will be repeated 4 times. Two times for writing the actual member and two more times for writing their fitness values. This makes the service time for the Fitness Module equal to:

$$s_4 = 4 + 2d + 4(4 + w + (\frac{r+7}{2})) = 34 + 2d + 4w + 2r \tag{5.25}$$

The flow rate for the Fitness Module is the total number of pairs of members that must be written to the memory, which is:

$$F_4 = \frac{mg}{2} \tag{5.26}$$

and the normalised service time for the Fitness Module is:

$$S_{norm4} = \frac{s_4 F_4}{F_{out}} = s_4 = 34 + 2d + 4w + 2r \tag{5.27}$$

5.2.6 GAP analysis

To determine the bottleneck of the GAP model in Figure 4.1, substitute the appropriate parameters into the equations for S_{norm_i} $\{0 \leq i \leq 4\}$ and find the maximum. The fixed parameters substituted were $w = r = 0$ and d=0. The value of T can be determined by

estimating how many messages the Read Module will have to generate, multiplying it by its service time s_0, and then adding the service times for the remaining pipeline modules to process the RM's final message (earlier messages would have been processed concurrently while the RM generated more messages):

$$T = \frac{m^2 g s_0}{4} + s_1 + s_2 + s_3 + s_4 \tag{5.28}$$

Because of the $m^2 g$ component in (5.28), the first term easily dominates the remaining four terms. Equation (5.28) can be simplified by dropping $s_1+s_2+s_3+s_4$ and leaving an approximation of T as:

$$T \approx \frac{\left(16 + \frac{84mg}{T}\right) m^2 g}{4} \tag{5.29}$$

Since the right hand side of (5.29) is in terms of T, convert it to a quadratic equation and find its roots:

$$T^2 - 4m^2 g T - 21 m^3 g^2 = 0 \tag{5.30}$$

which gives

$$T = 2m^2 g \pm \sqrt{4m^4 g^2 + 21 g^2 m^3} \tag{5.31}$$

and produces

$$T = m^2 g \left[2 \pm \sqrt{4 + 21/m}\right] \tag{5.32}$$

The positive root found from evaluating (5.32) was used to evaluate (5.7-5.32). The results of these evaluations are given in Table 5.2 which shows the service time and the normalised service time for each stage in clock cycles.

Since the Read Module has the maximum value of S_{norm}, it was identified as the bottleneck of the GAP. To evaluate the performance of the GAP, the information in Table 5.2 was analysed to determine the number of clock cycles per generation per population member. Here g is the number of generations, m is the population size, T is the estimated time for completion, and R is the rate in clock cycles at which that test executed per generation per population member. Values for R were calculated by:

$$R = \frac{T}{mg} \tag{5.33}$$

Table 5.3 presents the results which indicate roughly 4m clock cycles per generation per member.

5.2.7 Comparison between simulation and analysis

During the GAP simulations, statistics of number of clock cycles and the total service times were generated for all modules (Table 5.4). These values approximate the evaluations of the analysis functions given in Section 5.2.5 (Table 5.2). The Read Module is obviously the bottleneck of the GAP model.

Along with the above analysis, simulation statistics were generated to determine the actual number of clock cycles per generation per population member. Here T_a is the actual time for completion, and R_a is the actual rate in clock cycles at which that test executed per generation per population member. R_a was calculated by:

$$R_a = \frac{T_a}{mg} \tag{5.34}$$

Table 5.5 presents the results of the simulations which indicate that R_a is roughly $4m$ clock cycles per generation per member.

No.	m	g	s_0	s_{n0}	s_1	s_{n1}	s_2, s_3, s_{n2}, s_{n3}	s_4, s_{n4}	T
1	16	16	17.216	137.73	2.7490	21.992	7.1327	46	17685
2	16	32	17.219	137.74	2.7500	22.002	7.0663	46	35318
3	16	64	17.219	137.75	2.7509	22.007	7.0332	46	70583
4	16	128	17.219	137.75	2.7512	22.009	7.0166	46	141115
5	32	16	16.6309	266.09	1.6642	26.627	7.0337	46	68175
6	32	32	16.631	266.1	1.6643	26.629	7.0168	46	136297
7	32	64	16.631	266.1	1.6643	26.629	7.0084	46	272540
8	32	128	16.633	266.1	1.6643	26.63	7.004	46	545028
9	64	16	16.321	522.29	1.2765	40.849	7.0085	46	267467
10	64	32	16.322	522.29	1.2765	40.849	7.0043	46	534881
11	64	64	16.322	522.29	1.2765	40.85	7.0021	46	1069709
12	64	128	16.322	522.29	1.2765	40.85	7.0011	46	2139365
13	128	16	16.162	1034.4	1.1239	71.934	7.0022	46	1059273
14	128	32	16.162	1034.4	1.124	71.934	7.0011	46	2118493
15	128	64	16.162	1034.4	1.124	71.934	7.0005	46	4236933
16	128	128	16.162	1034.4	1.124	71.934	7.0003	46	8473813

Table 5.2: Analysis of the service times in clock cycles for the GAP model (s_n=s_{norm}).

No.	m	g	T	R
1	16	16	17685	69.0821
2	16	32	35318	68.98
3	16	64	70583	68.929
4	16	128	141115	68.9036
5	32	16	68175	133.154
6	32	32	136297	133.102
7	32	64	272540	133.076
8	32	128	545028	133.064
9	64	16	267467	261.198
10	64	32	534881	261.172
11	64	64	1069709	261.16
12	64	128	2139365	261.153
13	128	16	1059273	517.223
14	128	32	2118493	517.21
15	128	64	4236933	517.204
16	128	128	8473813	517.2

Table 5.3: Performance estimation based on the GAP analysis in Table 5.2.

Figure 5.5 compares the total number of cycles to complete a task in the case of both simulation and mathematical analysis. The analysis line and simulation line are very close which lends confidence to the mathematical analysis. There is a mismatch for both small and large m values. This mismatch is a result of simplification in the mathematical analysis.

Figure 5.6 compares the results of simulation and mathematic analysis for actual time completion. It shows that these two results are very similar and the difference is produced by the simplification in the mathematical analysis.

5.3 Design improvements

The above analysis and simulations of the GAP suggest that design improvements could be made in the following ways.

• Increase parallelisation of the Selection Modules which is the bottleneck of the GAP model. Parallelise the Selection-Crossover-Mutation-Fitness pipelines.

• Parallelise the internal structure of each module. For example, rather than selecting one pair at a time, the Selection Module can select two independent pairs. Then the SM can pass both pairs to the Crossover Module. The CM can then perform two independent crossovers in parallel and send them to MM, where two pairs can be mutated in parallel and sent to the Fitness Module for evaluation. This structure reduces inter-module communication overhead, requires increased communication bandwidth, and reduces the modularity of the design.

• Modify the Fitness Module to evaluate both new members in parallel. The current design evaluates them sequentially. Concurrent evaluation could save at least two clock cycles per pair of members. This would be most useful if the delay in the Fitness Module is large and affects the overall performance.

No.	m	g	s_0	s_{n0}	s_1	s_{n1}	s_2, s_3, s_{n2}, s_{n3}	s_4, s_{n4}	T_a
1	16	16	19.3975	155.18	2.459	19.6719	7.9543	46.5938	19807
2	16	32	19.7261	157.81	2.4844	19.875	7.4727	46.4219	40344
3	16	64	19.4114	155.29	2.457	19.6562	7.5059	46.3242	79453
4	16	128	19.4523	155.62	2.4629	19.7031	7.377	46.2832	159300
5	32	16	16.2439	259.90	1.7603	28.1641	7.3125	46.2969	66485
6	32	32	16.4684	263.49	1.7794	28.4707	7.2891	46.207	134850
7	32	64	16.5604	264.97	1.7878	28.6045	7.3643	46.1582	271270
8	32	128	16.9103	270.56	1.8168	29.0688	7.4102	46.1343	554060
9	64	16	15.9955	511.86	1.5336	49.0742	7.2422	46.1504	262015
10	64	32	16.1872	517.99	1.5497	49.5889	7.2539	46.1025	530370
11	64	64	16.0987	515.16	1.5423	49.3545	7.3096	46.0771	1054990
12	64	128	16.1079	515.45	1.5432	49.3838	7.2827	46.0664	2111235
13	128	16	15.4788	990.64	1.3895	88.9297	7.1543	46.0742	1014360
14	128	32	15.3877	984.82	1.382	88.4458	7.2266	46.0483	2016845
15	128	64	15.4796	990.69	1.3896	88.9365	7.2387	46.0361	4057825
16	128	128	15.5151	992.96	1.3926	89.1285	7.219	46.0306	8134315

Table 5.4: Simulation results of GAP tests.

No.	m	g	T_a	R_a
1	16	16	19807	77.3711
2	16	32	40344	78.7969
3	16	64	79453	77.5908
4	16	128	159300	77.7832
5	32	16	66485	129.8535
6	32	32	134850	131.6895
7	32	64	271270	132.4561
8	32	128	554060	135.2686
9	64	16	262015	255.874
10	64	32	530370	258.9697
11	64	64	1054990	257.5659
12	64	128	2111235	257.7191
13	128	16	1014360	495.293
14	128	32	2016845	492.3938
15	128	64	4057825	495.34
16	128	128	8134315	496.4792

Table 5.5: Performance simulations of the GAP tests.

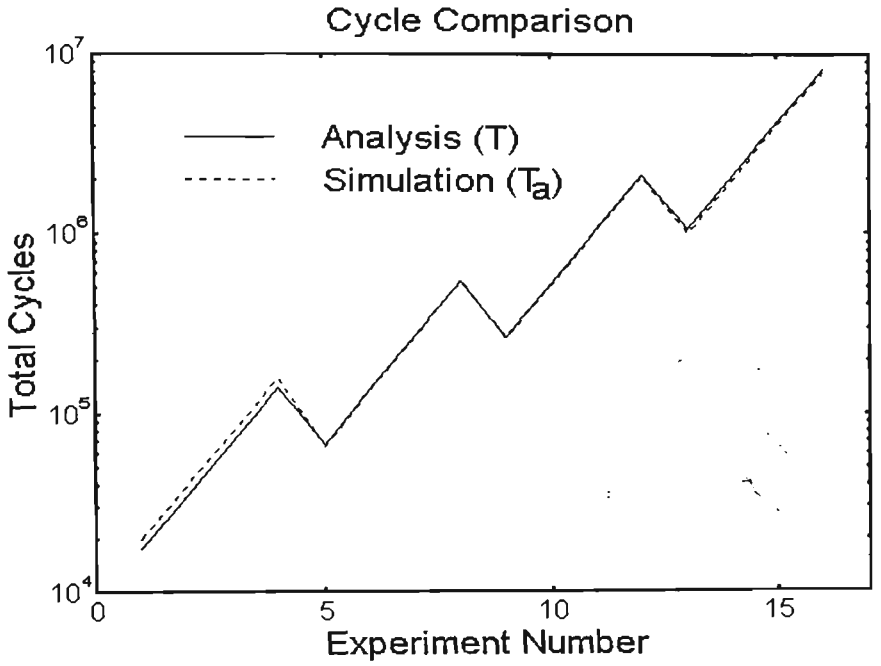


Figure 5.5: The result of comparison between mathematical analysis and hardware simulation of total number of cycles (T) needed to complete a task with different *m* and *g* according to Tables 5.3 and 5.5.

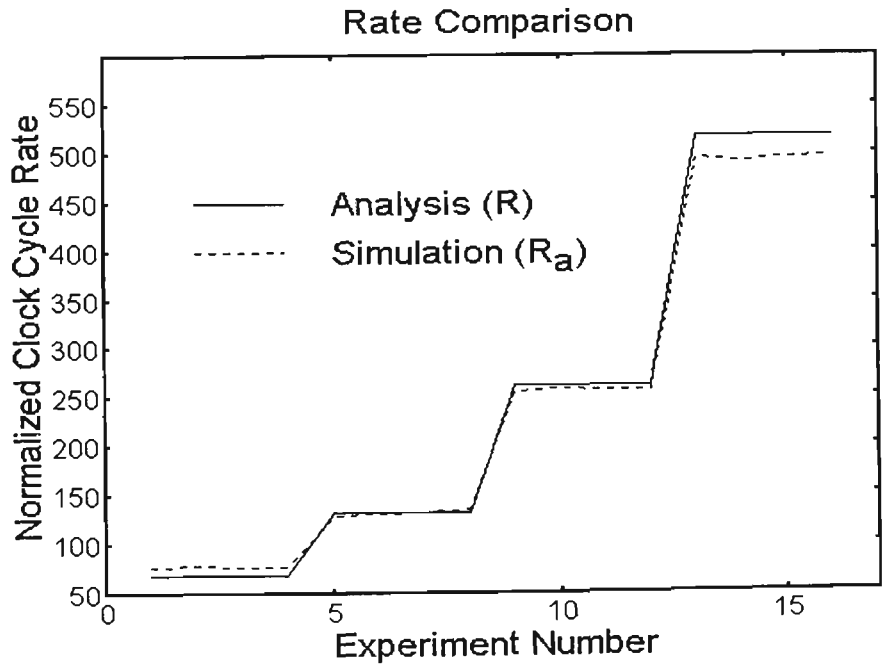


Figure 5.6: The result of comparison between mathematical analysis and hardware simulation of clock cycle rate (R) with different *m* and *g* according to Tables 5.3 and 5.5.

• Modify the Memory Interface Module's interaction with the Fitness Module. In the current GAP design the MIM services one request and then immediately looks for a new request. So after the FM sends one member, it has to wait for the RM to receive service before the FM has a chance to send the second member. Modifying the MIM to always service the FM twice in a row would reduce some blocking delays for the FM. However, this could also increase some blocking delays for the RM.

• Make the inter-module communication protocol more efficient. The current handshaking protocol requires six clock cycles per data transfer. If these delays were reduced, the entire GAP would run much faster.

• Buffer the outputs of all the modules. This should reduce the delays associated with some modules waiting for service and blocking others that are waiting upstream in the pipeline.

• Use a memory configuration which supports reads and writes of population members in parallel. Coupled with effective buffering, this could significantly reduce delays due to modules blocking each other. Additionally, an ability to read from one population while concurrently writing to another would eliminate the blocking that occurs between the Read Module and the Fitness Module.

• When implementing the aforementioned improvements, design the GAP from scratch. The original VHDL implementation of the GAP could be redesigned to be much more space and time efficient. Throwing away the old design while retaining the lessons learned from it is the best way to attain this goal. Some modules could even be designed at the gate level if even more speedup were desired. The improved space and time efficiency would allow faster clocking.

Chapter 6

Implementation of the GAP on FPGAs

This section starts with an explanation of programmable technologies particularly FPGAs. This is followed by a description of the GAP implementation on FPGAs and preliminary hardware tests. Finally the GAP is compared with a software model to find how much speed advantage can be achieved.

6.1 ASIC design

Integrated circuits consist of connected transistors fabricated on a single semiconductor chip. The locations and connectivity of the transistors are defined by several masks. A mask corresponds to one of the silicon compound layers that form transistors and interconnect layers.

Digital integrated circuit implementations may be grouped into two main categories, fully custom and semicustom designs, as illustrated in the hierarchy shown in Figure

6.1. These approaches have facilitated the design and manufacturing of Application-Specific-Integrated Circuits (ASICs) for digital applications.

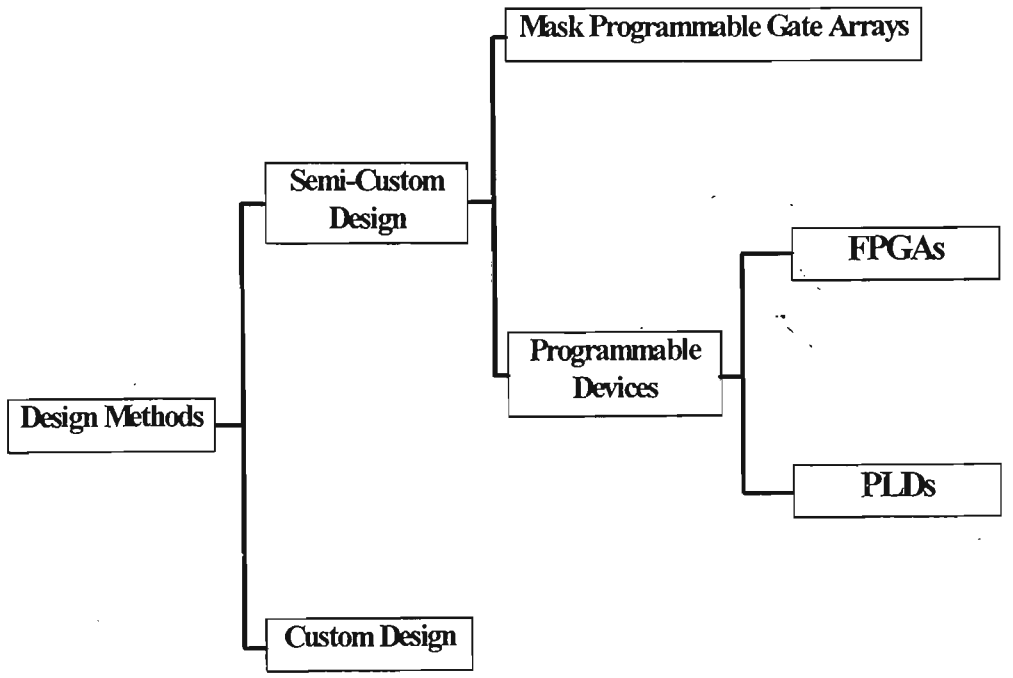


Figure 6.1: Circuit design methods and target technologies.

1. **Custom design:** Custom ICs are created using unique masks for each layer during the manufacturing process. Since the designer controls all stages of the chip layout, maximum design flexibility and high performance are possible. Consequently, only highly skilled designers are engaged in this design methodology. Also, the development time is long, and its costs are extremely high. For high volume applications, the high cost of design and testing custom ICs can be ignored.

2. **Mask-Programmable Gate Arrays (MPGAs):** Gate array implementations use generic masks for all but the metallisation layers, which are customised to the user's specifications [Hollis, 1987]. The generic masks create an array of modular functional blocks. Modules of transistors are arranged in rows that are separated by fixed-width channels. User logic is implemented by patterning these transistors into logic functions and connecting the different modules. The design is usually facilitated by a cell library,

making the designer's expertise less critical than in the case of the full custom methodology. For the same reasons, MPGAs offer shorter development time and lower development costs than do custom ICs.

6.1.1 Field programmable technology

Field-programmable devices are prefabricated in a variety of architectures based on an array of logic cells. The logic is implemented by personalising the basic cells and electrically programming the interconnects. This is usually done in the user's laboratory rather than the factory.

Implementing the design in programmable logic devices has the advantage of fast turnaround but it limits the design flexibility. Development time and costs are significantly lower than for any other IC implementation methodology but the cost per gate is high for volume production. According to their architectures, two main categories of user programmable logic devices can be distinguished: Programmable Logic Devices (PLDs) and Field-Programmable Gate Arrays (FPGAs).

1. *PLDs* consist of programmable AND arrays (product terms) and fixed fan-in programmable *OR* gates that are followed by flip-flops, as shown in Figure 6.2. The outputs of the flip-flops can be fed back as input lines in the product terms. The product line can be connected to any combination of inputs. The connections are indicated by an "o" and are programmed by users to implement their designs. The connecting device may be a fuse as in the case of bipolar chips, or a transistor. The transistor can be chosen to act as an open connection or to function normally as a switch [Monolithic Memories, 1986]. PLDs are at the low-density end of field programmable logic devices. Their densities range from 1,000 up to 10,000 gates. Their utilisation varies with applications, but it is typically very low because of the rigid AND/OR architecture. Initially, PLDs used to be fabricated in bipolar technology but Complementary Metal-Oxide Semiconductors (CMOS) devices are now more popular.

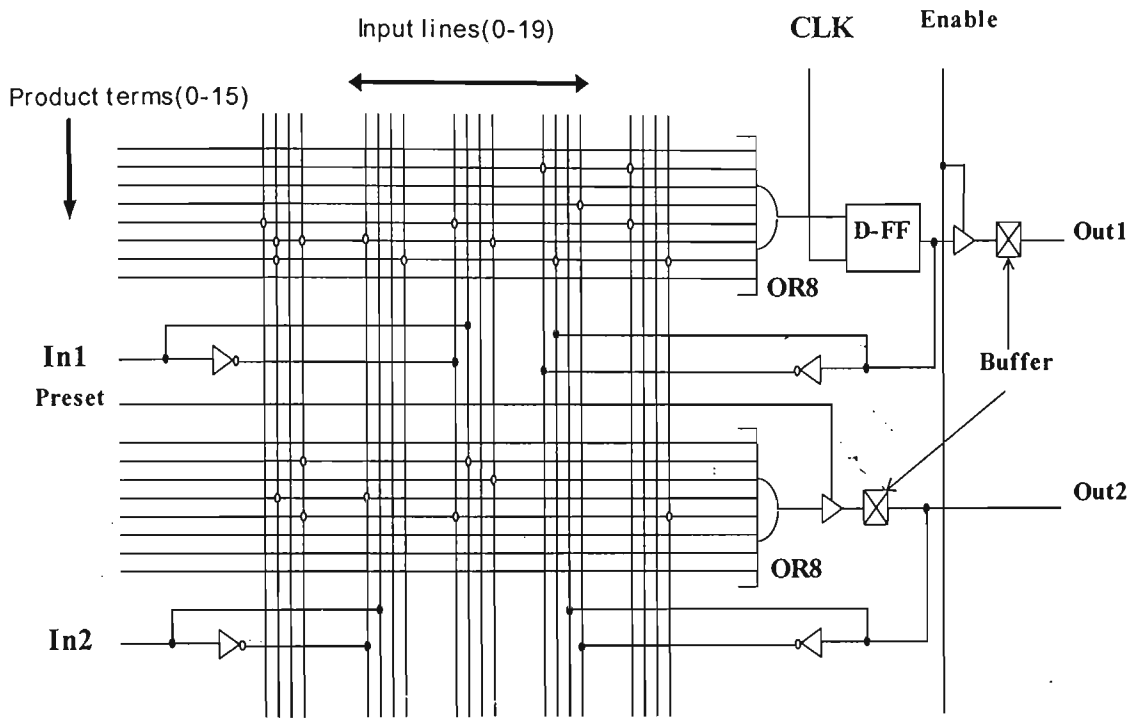


Figure 6.2: General structure of a PLD.

2. *FPGAs* combine the architecture of gate arrays with the programmability of PLDs. Some of the FPGA real estate is occupied by vendor logic to implement the field programmability feature of the FPGA, and a large portion of the die area is allocated for programmable routing. The number of gates typically available to the user in current (1995) designs varies from 3,000 to 40,000. An FPGA normally consists of an array of uncommitted logic blocks in which the design is to be encoded (Figure 6.3). A logic block consists of universal gates that can be programmed to represent any function: multiplexers (MUXs), Random-Access Memories (RAMs), NAND gates, transistors, etc. The connectivity between blocks is programmed by different types of devices, Static Random-Access Memory (SRAM), Electrically Erasable Programmable Read-Only Memory (EEPROM), or antifuses. Further description of FPGA architectures is given in Appendix D.

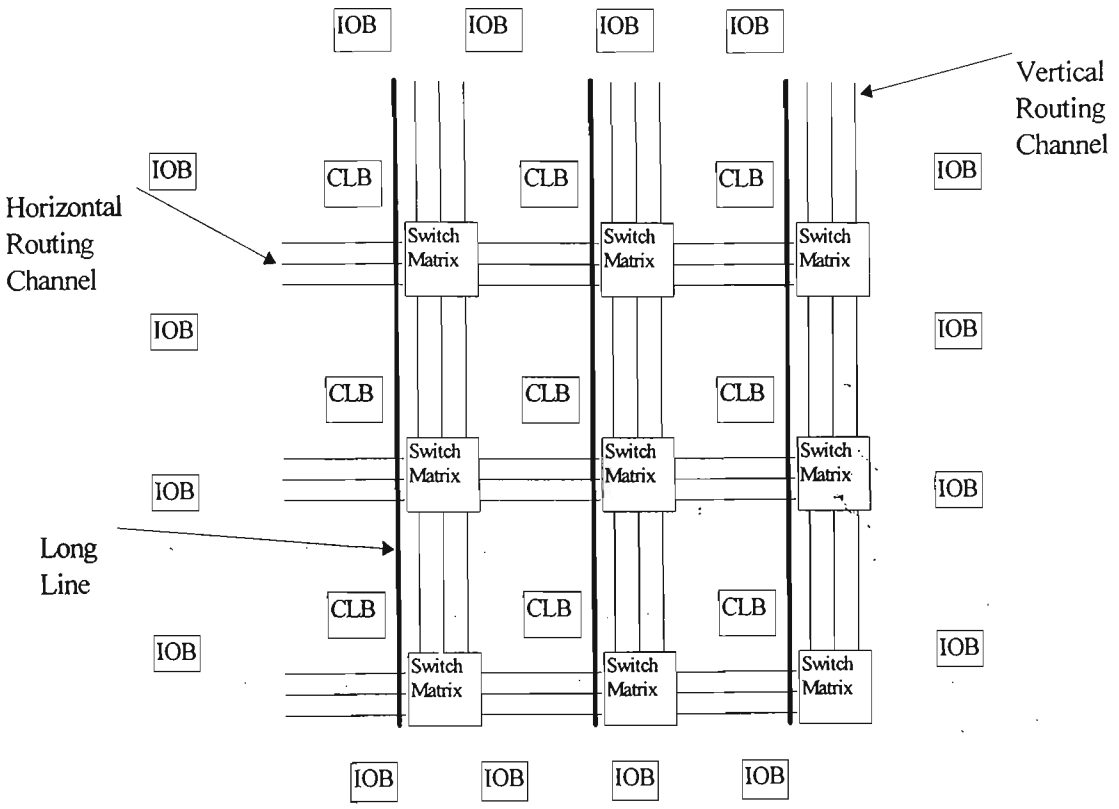


Figure 6.3: General structure of a FPGA.

6.1.2 The design cycle

The design process for using FPGAs generally requires six steps [Chan and Mourad, 1994]:

1. Entering the design in the form of schematics, logic expressions and hardware description language statements.

2. Simulating the design for functional verification.

3. Mapping the design into the selected FPGA architecture.

4. Placing and routing the FPGA design.

5. Extracting delay parameters and maximum frequency.

6. Configuring or programming the device.

Most FPGA vendors allow the entry of the design in a schematic form though, it is also possible to enter the design in boolean expressions. The most flexible design entry is via a hardware description language. The most popular languages are Verilog HDL and VHDL. Such languages require logic synthesis tools before mapping into FPGAs.

Skipping the simulation steps enables the designer to obtain the end product faster, but can weaken the product quality. Usually, the design goes through several iterations of simulation.

For every FPGA, the vendor provides design implementation tools to perform steps 3 to 5. The front-end design entry (schematic capture or other methods) and simulators may also be part of the tools. Most vendors configure their package with different front-end tools to allow the user more choice and flexibility.

Steps 3 and 4 involve several processes: logic minimisation, technology mapping, placement, and routing. Technology mapping binds the technology independent description of the circuits to the basic entities of the target technology. Placement allocates these entities to a specific physical block on the device and routing establishes the connections between the different blocks, and is usually done in two stages: *global routing* and *detailed routing*. All FPGA vendors have an automatic placement and routing tool. The placement and routing algorithms have a strong effect on the performance of the design.

6.2 Design implementation cycle

The GAP was implemented on FPGA technology using the VHDL language [Coelho, 1989] and Mentor Graphics tools. These tools facilitate modelling the behaviour and design of the architecture. QuickVHDL was used to compile the design and the NeoCAD FPGA Foundry software converted the GAP modules to Xilinx files to implement the design into field programmable gate arrays.

6.2.1 Entering the design

There are many advantages in using VHDL. Firstly, it is accepted internationally for hardware implementation and there are many vendors that provide simulators for VHDL. Secondly, it is a high level language and the programmer does not need to specify all details of the design. For instance the add and multiply operators, and in some cases division operators, are automatically expressed in hardware and the designer can use them in any design. Third, VHDL is the generic language for hardware implementation. If in the next decade new technology arrives and replaces existing hardware technologies, then by using a mapping tool we will be able to map our design to the new technology.

The design is written in a hierarchical configuration. The main core of the GAP consists of the Memory Interface Module, Read Module, Selection Module, Crossover Module, Mutation Module, and Fitness Module (Figure 4.1). Each module is written in a separate block and one main block (GAP Module) handles connectivity between the modules. This block is used for synthesis and hardware implementation. For the simulation purposes, two more modules (Memory Unit and Fitness Unit) are included with the GAP Module to form the main design (Main Module).

After designing the GAP modules in VHDL, they are compiled to generate a schematic diagram. Figure 6.4 shows the schematic diagram of the GAP module, created automatically using the Autologic tools in Mentor Graphics Software. The six blocks inside the schematic are the same six blocks in the Figure 4.1. The input and output signals are necessary to connect the GAP to the Memory Unit and Fitness Unit.

6.2.2 Simulating the design

Computer-aided design tools have greatly facilitated the design implementation process. These tools have replaced many of the heavy design tasks such as design entry,

verification and synthesis (logic minimisation, technology mapping, state reduction, and state assignment). These tasks are time consuming and often error-prone.

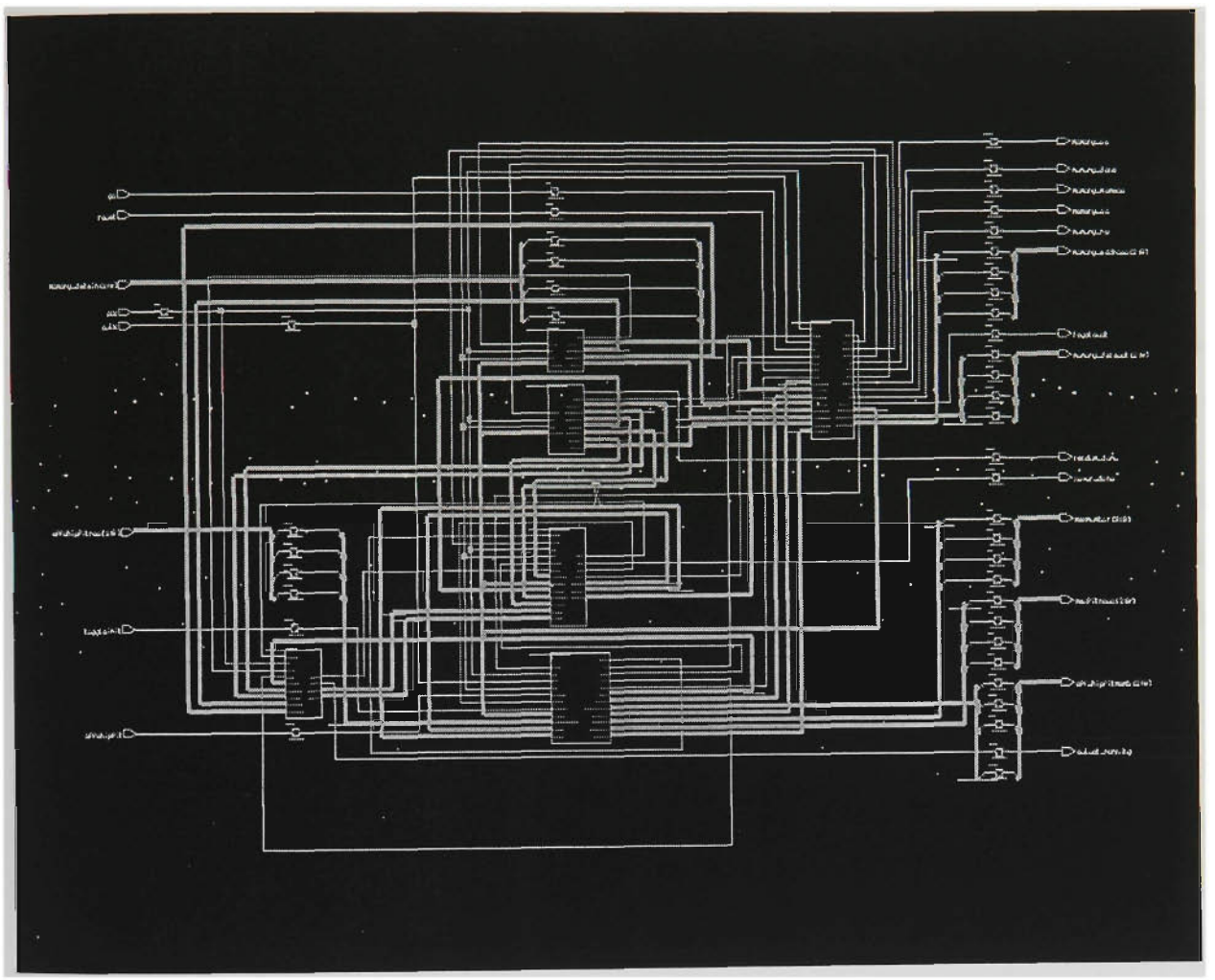


Figure 6.4: The schematic diagram of the GAP.

Simulation is a process that imitates the functionality or behaviour of the digital design on a computer. It is used to identify design errors or timing problems in a circuit. Simulation was originally used for prototyping and employs models that represent system attributes. These may include behavioural or timing models. Now it is often used to debug a prototype. FPGAs have also been used to prototype designs before their actual fabrication in other technologies.

There are several aspects of a digital circuit that need to be verified before implementation: functionality, timing, effect of certain parameters, etc. There are also a variety of simulation types that are dictated by circuit-level representations. For example, the functionality of a circuit may be simulated at the behavioural level, the gate or logic level, or the circuit level. In the case of FPGAs, the functional simulation can be performed at the behavioural level or the gate level. This simulation may be run with zero (no) delays, or one-unit delays. In the case of timing simulation, several approaches can be taken. Timing can be checked with nominal delays for the technology or with worst case scenarios. More importantly, timing can be verified using the actual layout of the design on the FPGA. The actual delays of the placed and routed design can be extracted and used in a timing simulation.

Initially the behaviour of the model was simulated without regard to synthesis in hardware. It is possible to mix and test a combination of synthesisable and unsynthesisable modules. For the test examples the simulation studies were conducted with the memory and fitness units in an unsynthesised form.

The Memory Unit in the design is based on the MCM6164 static 8K × 8bit RAM chip. Two or more memory chips operate in parallel to give 16 to 32 bit member strings in the population. The GAP module provides the necessary read, write and chip-enable signals for memory operation.

QuickVHDL version 8.2 was used for compiling and simulating the GAP Module and the whole design. The simulation of modules for correct functionality was shown in Section 5.1.

6.2.3 Mapping the design into FPGAs

There is a wide variety of Field-Programmable Gate Arrays as shown in Table 6.1. Some of these devices are actually Programmable Logic Devices with specific

enhancements that make them larger and more flexible than traditional PLDs. The products can be compared by focussing on four factors that can influence the design and the selection of the device for a given application:

1 - architecture,

2 - gate density or capacity,

3 - routing resources or basic cells,

4 - programming method.

For each vendor listed in the first column in Table 6.1, the range of capacities for different devices is given. The effective capacities are usually lower since high utilisation of the logic modules would generally decrease routability. The architecture is identified in the third column as gate array (or row-based FPGAs), matrix form, sea-of-gates, or Programmable Logic Array (PLA). The next column is the logic unit in which the user logic is implemented. The last column lists the programming method.

The Xilinx logic cell array family was introduced in 1983. Since then the product has passed through three generations: series XC2000, XC3000, and more recently the XC4000.

Table 6.2 summarises the main features of the three generations of Xilinx devices. The number of equivalent gates capacity (two-input NANDs) serves as a guide for a designer to select the appropriate part type.

User logic is implemented by configuring the logic components. The Xilinx chip incorporates SRAM technology and is reprogrammable. The number of Configurable Logic Blocks (CLBs) in an Logic Cell Array (LCA) ranges from 64 in the XC2064, the low end of the 2000 series, to 576 in the XC4013, the largest device presently available of the 4000 series (1994).

A Xilinx 4000 series component (XC4013) was selected to implement the design because of its high capacity and large number of Input/Output pins.

The following steps are followed in transferring the VHDL models into FPGA technology. First the model (GAP Module) is synthesised using Autologic version 8.2 from the Mentor Graphics Software. Then the synthesized model is transferred to a netlist file format using Mentor Graphics Software.

Manufacturer	Capacity (Number of gates)	Architecture	Basic Cell	Programming Method
Actel	2K-8K	Gate Array	MUX	Antifuse
Altera	1K-5K	Extended PLA	PLA	EPROM
Algotronics	5K	Sea-of-gates	Functional	SRAM
Concurrent	3K-5K	Matrix	XOR, AND	SRAM
Crosspoint	5K	Gate Array	Transistors	Antifuse
Plessey	2K-40K	Sea-of-gates	NAND	SRAM
QuickLogic	1.2K-1.8K	Matrix	MUX	Antifuse
Xilinx	2K-10K	Matrix	RAM block	SRAM

Table 6.1: Examples of FPGAs.

Feature	XC2000	XC3000	XC4000
Number of chips in family	2	6	11
Equivalent gates	1K-1.5K	2K-9K	2K-13K
MAX I/Os	58-74	64-176	64-192
Flip-Flops	122-174	256-1320	256-1536
MAX CLBs	64-100	64-484	64-576
Number of package PINs	34-74	34-176	61-193

Table 6.2: Features of the Xilinx devices (up to year 1994).

Then the NeoCAD FPGA Foundry tools are used to map the netlist file to an FPGA device. The result is a mapped model of the design for a specific family (XC4000). The mapped model is then placed and routed on the selected device type (i.e. XC4013). Finally the model is transferred to bitmap format for downloading to the chip. If the design is small enough to fit in one chip then the place and route phase and downloading is straightforward. On the other hand if the design is large (which occurs in most cases of the GAP implementation) then it must be partitioned across multiple chips. There are special tools in the NeoCAD software for partitioning. Figure 6.5 shows an FPGA XC4003 chip containing the Read Module. The black boxes inside the chip are the CLBs with defined logic.

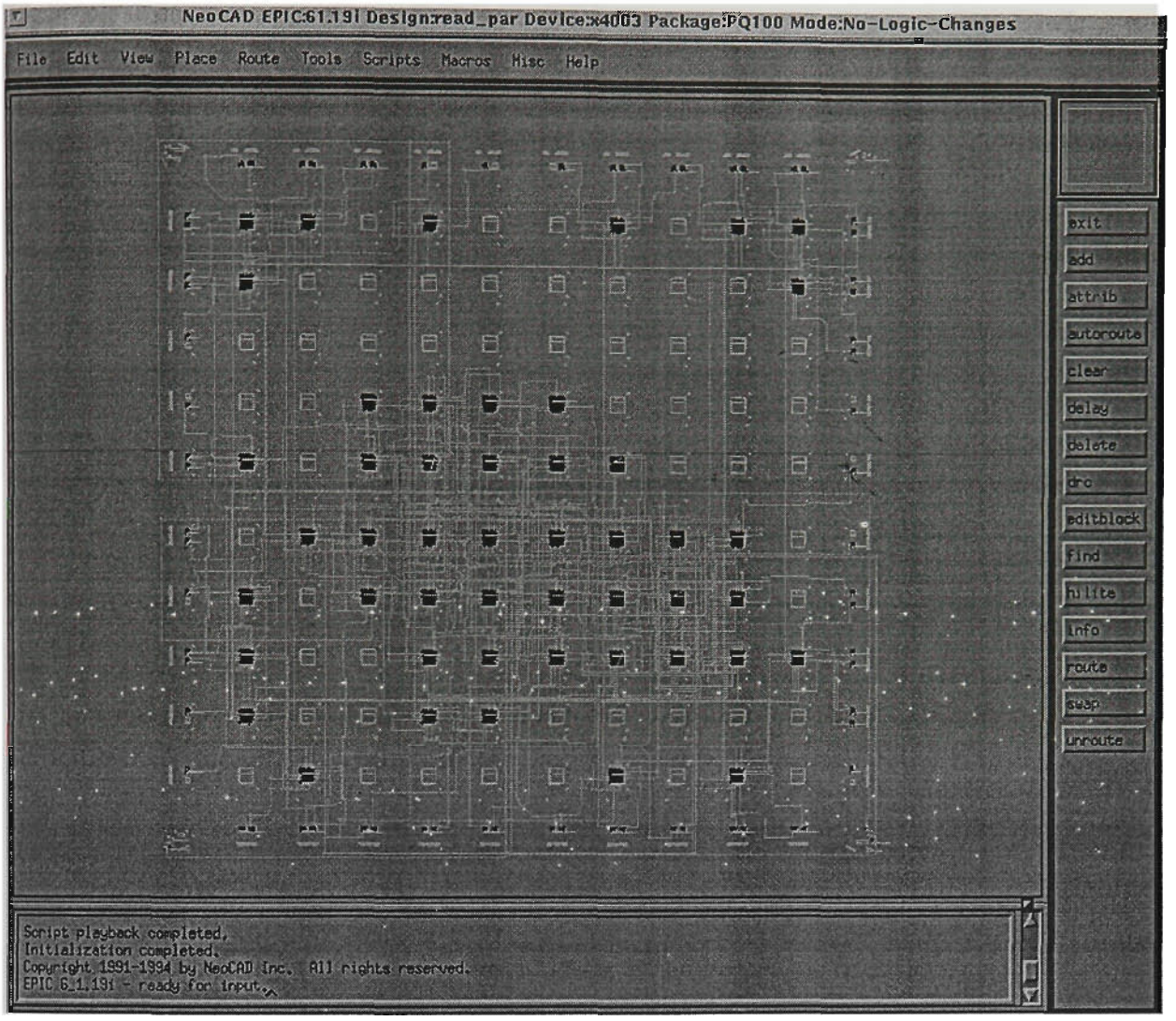


Figure 6.5: The Read Module for the 24 bit configuration on XC4003.

Converting the VHDL code to a bitmap file normally takes hours and depends on the memory bit length in the GAP Module. The steps in the process were timed for GAPs with 4, 8, 16, 24 and 32 bit members. The time for each step in the conversion process is summarised in the Table 6.3. All timing is based on running the process steps on a SUN Sparc 10 workstation.

Table 6.3 shows that most of the processing time is spent in the synthesise, partitioning and place and route phases.

Configuration	4 BIT	8 BIT	16 BIT	24 BIT	32 BIT
Number of Chips	1	2	2	2	3
Compile (Minutes)	10	10	10	10	10
Synthesise (Minutes)	60	72	120	150	180
Netlist Format (Minutes)	10	14	18	20	30
Partitioning (Minutes)	----	60	90	150	180
Mapping (Minutes)	30	40	50	60	70
Place and Route (Minutes)	40	54	115	150	210
Bit Generation (Minutes)	3	5	7	15	20
Total Time (Minutes)	153	255	410	555	700

Table 6.3: The processing time for mapping VHDL source code of GAP to an FPGA bitmap file on a SUN Sparc 10 workstation.

6.2.4 Programming an FPGA device

The bitmap file contains information which should be downloaded into an FPGA device. A commercially available demonstration board (Figure 6.6) can be used to download the design in the simplest form into a single FPGA. Because of limitations of the board, it is not possible to download a model of the GAP with more than 4 bit memory length.

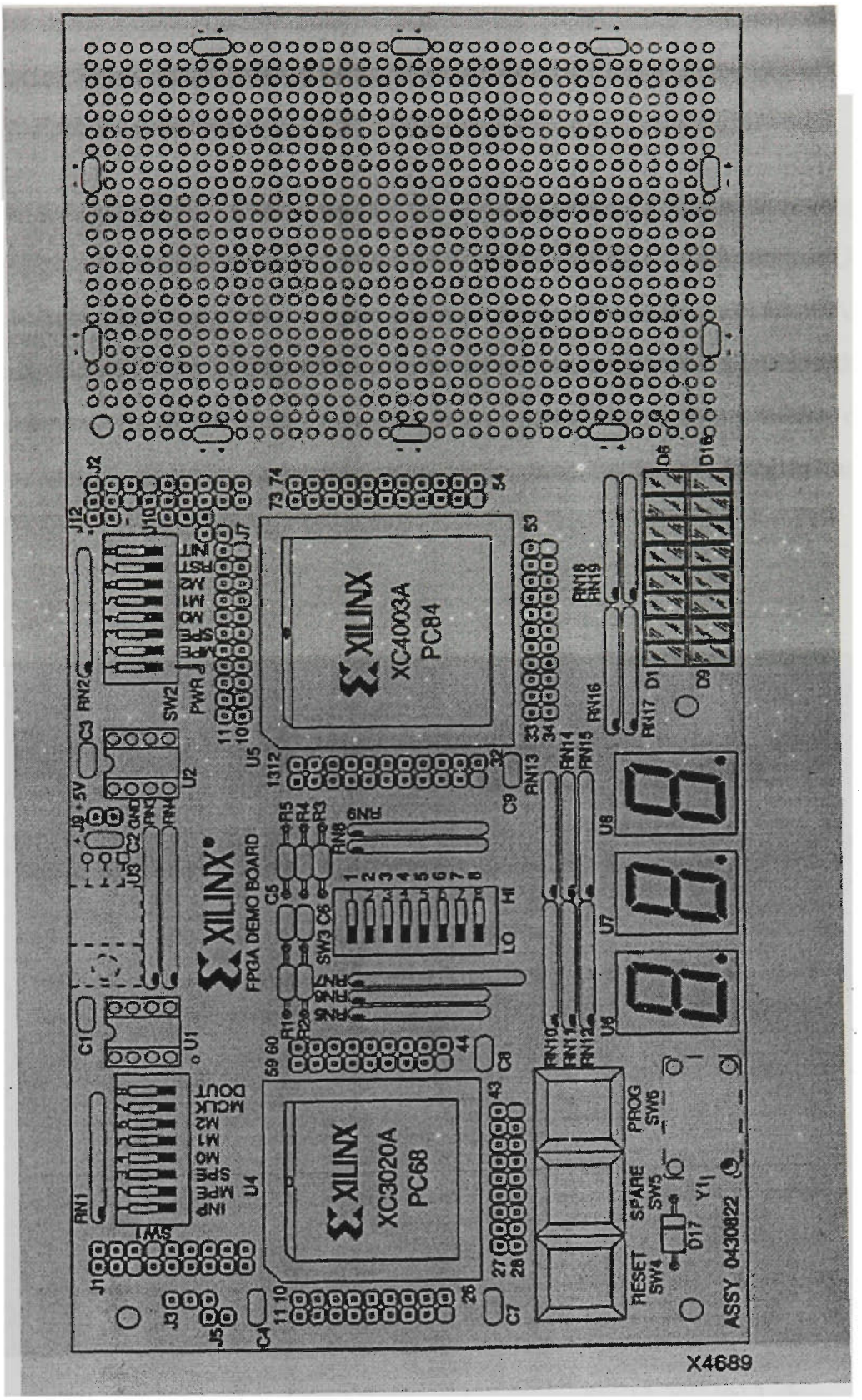


Figure 6.6: FPGA demo board component layout.

The FPGA demo board is a stand-alone board for experimenting and prototyping with FPGAs using Xilinx FPGA device families. The board comes with an XC3020-68 PIN and an XC4003-84 PIN part.

The board connects to the output port of the SUN workstation. There is a prototyping area on the board for adding the memory chips and connection to the Fitness Unit. The program for downloading a bitmap is called BITGEN and is part of the NeoCAD tools. The bitmap file can be downloaded into an on-board serial EPROM and then the board can use the contents of the EPROM at start up for downloading to the FPGA chip. An alternative way of programming the FPGA is by downloading directly to the FPGA using BITGEN and then testing the model.

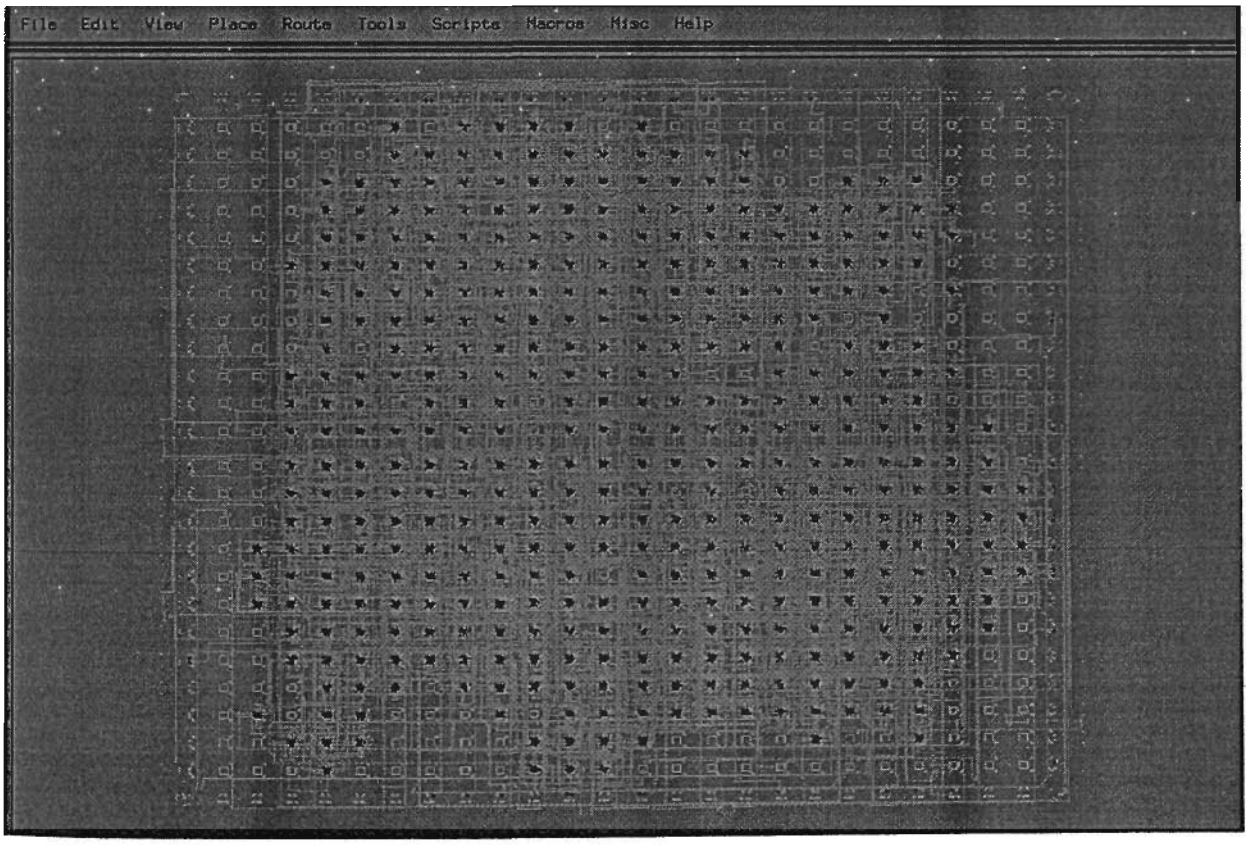


Figure 6.7: A 4-bit GAP implemented on the FPGA XC4013 chip.

The simplest configuration for the GAP (i.e. 4 bit for memory data and 4 bit for memory address) can be implemented on a smaller device of the 4000 series such as 4010. Figure 6.7 shows the layout of the 4-bit GAP design on an XC4013 chip.

6.2.5 GAP parameters and timing considerations

There are many potential applications for Genetic Algorithms and the GAP in engineering. For best performance, each application requires a different set of parameters including memory bit length, memory size, population size and probability of crossover and mutation. Some of these parameters (population size and probability of crossover and mutation) will be stored in the memory before operation of the GAP. The size and bit length of memory are critical structural features of the GAP and must be specified in the VHDL code.

There are several timing issues which must be resolved before the GAP can be used in a real-time engineering application. These include the maximum clock frequency of the GAP, the number of clock cycles required to complete a task and the response time of the fitness calculation.

Table 6.4 shows the maximum clock frequency of the GAP varying the member's bit length. For these results, 8 bits is chosen for the fitness value and the Fitness Unit is assumed to return the fitness value within one clock cycle. The table shows that if more XC4013 chips are used in the implementation, then the maximum attainable frequency will be increased. In the first row, the maximum frequency for 4 bits is similar to the 8 bit configuration because the fitness length is also 8 bits. If 4 bits is selected for the fitness value then it is possible to implement the GAP on a single FPGA chip and the maximum frequency falls to 5.55 MHz.

Member Bit Length	Number of XC4013	Maximum Clock Frequency (MHz)
4 bit	**2**	**6.26**
8 bit	**2**	**6.53**
16 bit	**3**	**8.20**
32 bit	**4**	**9.67**
32 bit	**5**	**11.2**

Table 6.4: Maximum attainable frequency of the GAP for different member bit lengths.

Table 6.5 shows the number of clock cycles to process a given number of generations for different configurations. The table shows that the number of clock cycles required to complete a task increases dramatically with the population size (m) and the number of generations (g) as expected from the analysis in Chapter 5. The fourth row of the table shows the real time for processing the task assuming a 10 MHz clock frequency. Note that the Fitness Unit is assumed to return the fitness value within one clock cycle which in this case is 100 nS. The real time for processing the task is between 4 and 400 milliseconds.

Population size	16	16	32	32	64	64	128	128
Number of generations	32	64	32	64	32	64	32	64
Number of clock cycles	40344	79453	134850	271270	530370	1054990	2016845	4057825
Total GAP time (mS)	4.03	7.9	13.5	27.1	53.03	105.5	201.7	405.8

Table 6.5: Number of clock cycles needed by the GAP for processing a task and the corresponding real time (based on working clock frequency of 10 MHz).

The next issue is the relationship between the response time of an application and the processing capability of the GAP. Figure 6.8 shows the results of simulation based on a 10 MHz clock frequency for the GAP. The figure shows the total GAP time to process a given number of generations versus the Fitness Unit delay time when the population size is 16. There are four curves, one for each number of generations (G). If the delay time is small then the GAP needs a fixed amount of time to complete the task which depends on the number of generations. On the other hand if delay time is high then the GAP is waiting for responses from the fitness system. Note that the delay break time (which marks the transition from full processing to mostly waiting by the GAP) is similar for all numbers of generations (i.e. about 5μS).

Figure 6.9 shows the same information when the number of generations is fixed at 16 and the population size (P) is varied. The graph shows the delay break time is proportional to the size of population.

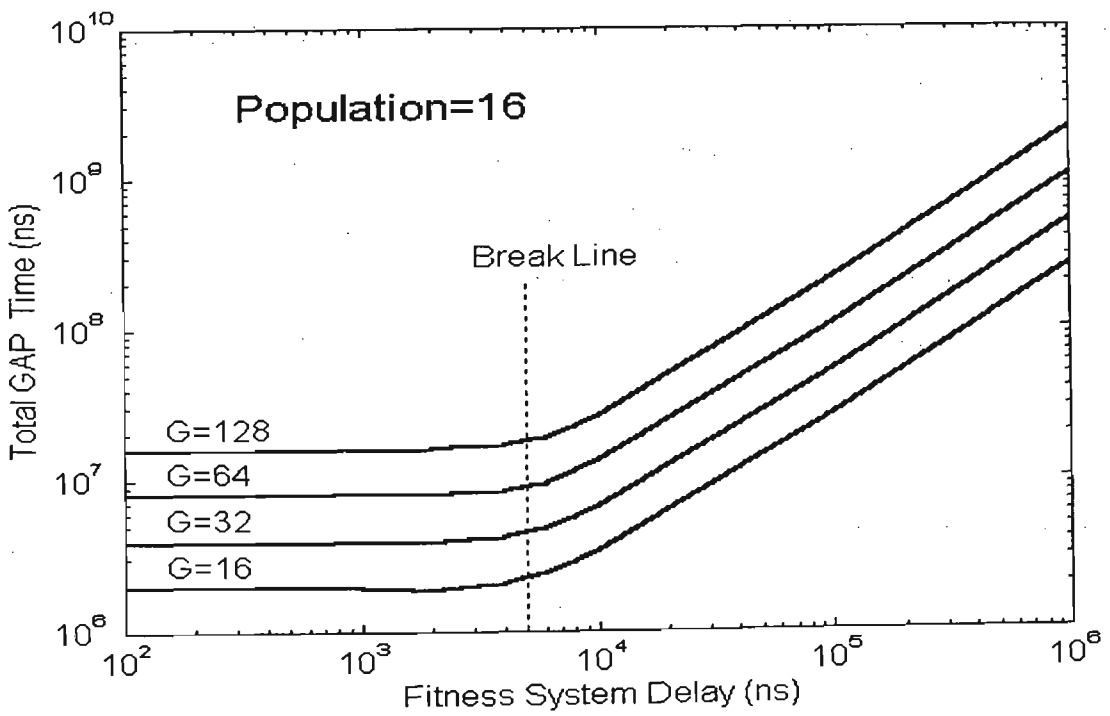


Figure 6.8: Total GAP run time versus the fitness delay time when the number of generations varies for population size equal 16.

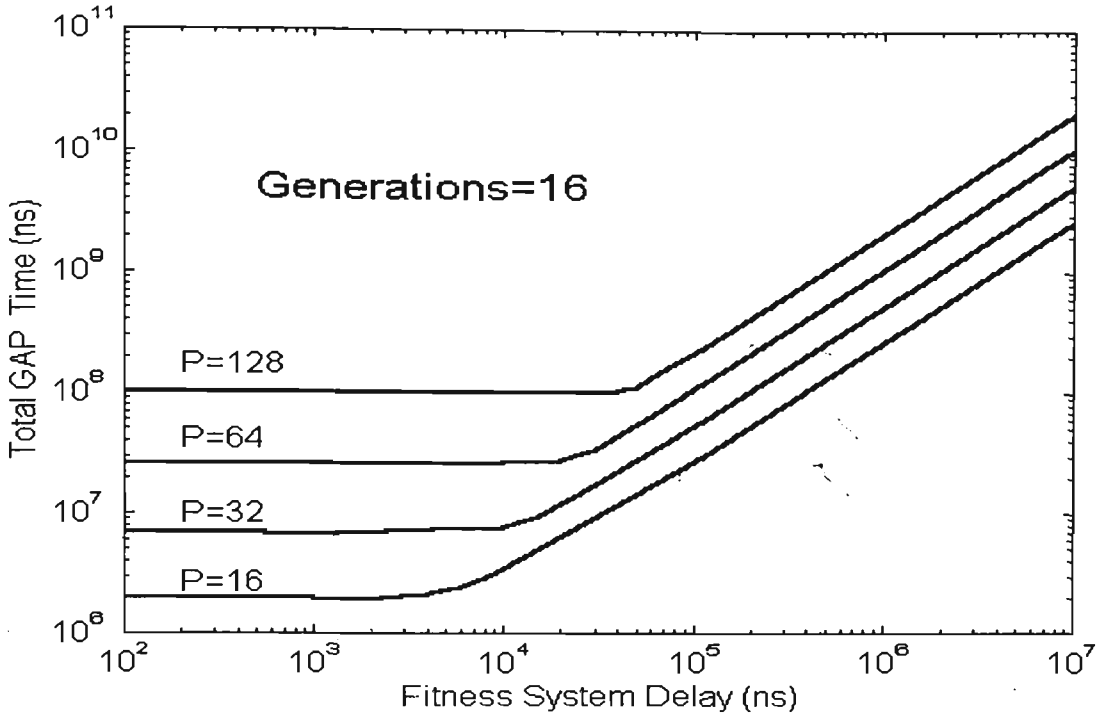


Figure 6.9: Total GAP run time versus the fitness delay time when population size varies for number of generations equal 16.

This break point is very important for the GAP model and fitness system or application. Indeed before the break point, GAP processing is responsible for 100% of the time to complete a task, but after that point the percentage falls towards zero. So to take full advantage of processing speed of the GAP then the delay time of the Fitness Unit (i.e. the application) must be less than the break point. The break point depends only on the clock frequency of the GAP and the size of the population.

6.3 Comparison with a software GA

The whole model was tested for performance and comparison with standard genetic algorithm models. Firstly the model was tested with four standard static problems and the result of optimisation was compared with software GAs. Later the VHDL models were compared with the software approach to compare the speed performance in real time processing.

6.3.1 Testing the optimisation capability

The GAP model was tested for functionality using a set of four standard fitness functions from the DeJong test suites [DeJong, 1975]. These tests include searches for an optimum point on flat, curving or noisy surfaces and they are considered to be difficult for conventional search algorithms. In each case the equation for the surface provides the fitness function for the test and each member string represents a point on that surface. The test functions are defined as follows.

1 Sphere: A unimodal function that is three dimensional and has one minimum at (0,0,0):

$$F_1(\vec{X}) = \sum_{i=1}^{3} x_i^2 \tag{6.1}$$

2 Rosenbrock's saddle: A two-dimensional function that has a curving valley that fools many optimisation algorithms into halting prematurely and returning a point that is not the global optimum:

$$F_2(\vec{X}) = (100*(x_2 - x_1)^2 + (x_1 - 1)^2) \tag{6.2}$$

3 Step: A five-dimensional function that has long, flat surfaces surrounded by discontinuities. Simple hill-climbing algorithms often become stuck on these flat surfaces:

$$F_3(\vec{X}) = \sum_{i=1}^{5} \text{int}\,eger(x_i) \tag{6.3}$$

4 Quartic: A 30-dimensional function in which evaluation is modified by Gaussian noise. Thus, successive evaluations of the same point return different values:

$$F_4(\vec{X}) = \sum_{i=1}^{30} ix_i^4 + Gauss(0,1) \tag{6.4}$$

The results of these hardware simulations were compared with those from a software implementation using the package SUGAL V2.0 [Hunter, 1995]. A population of 64 individuals and a mutation rate of 2% and crossover rate of 90% were used for all tests.

The software GA employs floating point arithmetic and advanced methods of selection, normalisation, crossover, mutation and replacement. These methods are generally difficult to implement in a hardware algorithm and therefore the GAP was not expected to perform to the same precision.

The results of comparisons are given in Figure 6.10 to 6.11, which shows the evaluation of the error value of the best point averaged over 10 runs as a function of the number of evaluations. One cannot expect the hardware approach to outperform the software algorithm in any way. However these tests indicate that the hardware is capable of reaching a similar optimum point in the same number of generations.

6.3.2 Comparing the speed of hardware and software

The GAP was compared with the Software-based GA (SGA) running on a 66 MHz Pentium. The SGA is functionally similar to the GAP. To make the software GA even more similar to the GAP, the software GA was changed in the following ways.

• The SGA's population maintenance was changed from copying population P_{t+1} into population P_t at the end of each generation to using the mapping $h:\{P_t, P_{t+1}\}\rightarrow\{P_0, P_1\}$ as described in Chapter 4. If $h(P_t)=P_0$ in the current generation, then $h(P_t)=P_1$ in the next generation. This mapping was controlled by a value that was toggled after every generation, just like the GAP's implementation. Thus, all the software GA's copying overhead was removed.

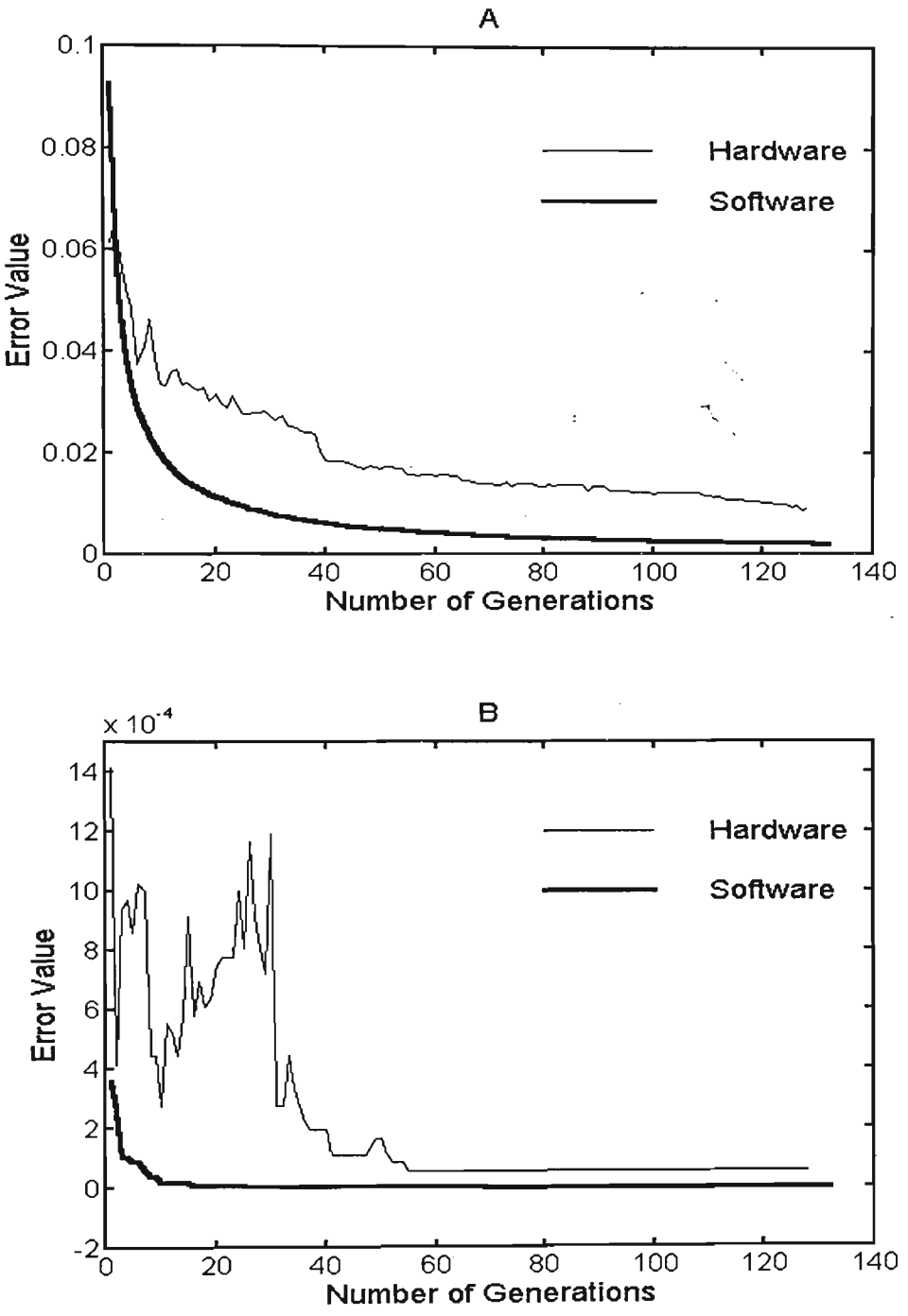


Figure 6.10: The error value of the best individual versus number of generations (A =Sphere, B =Rosenbrock's saddle).

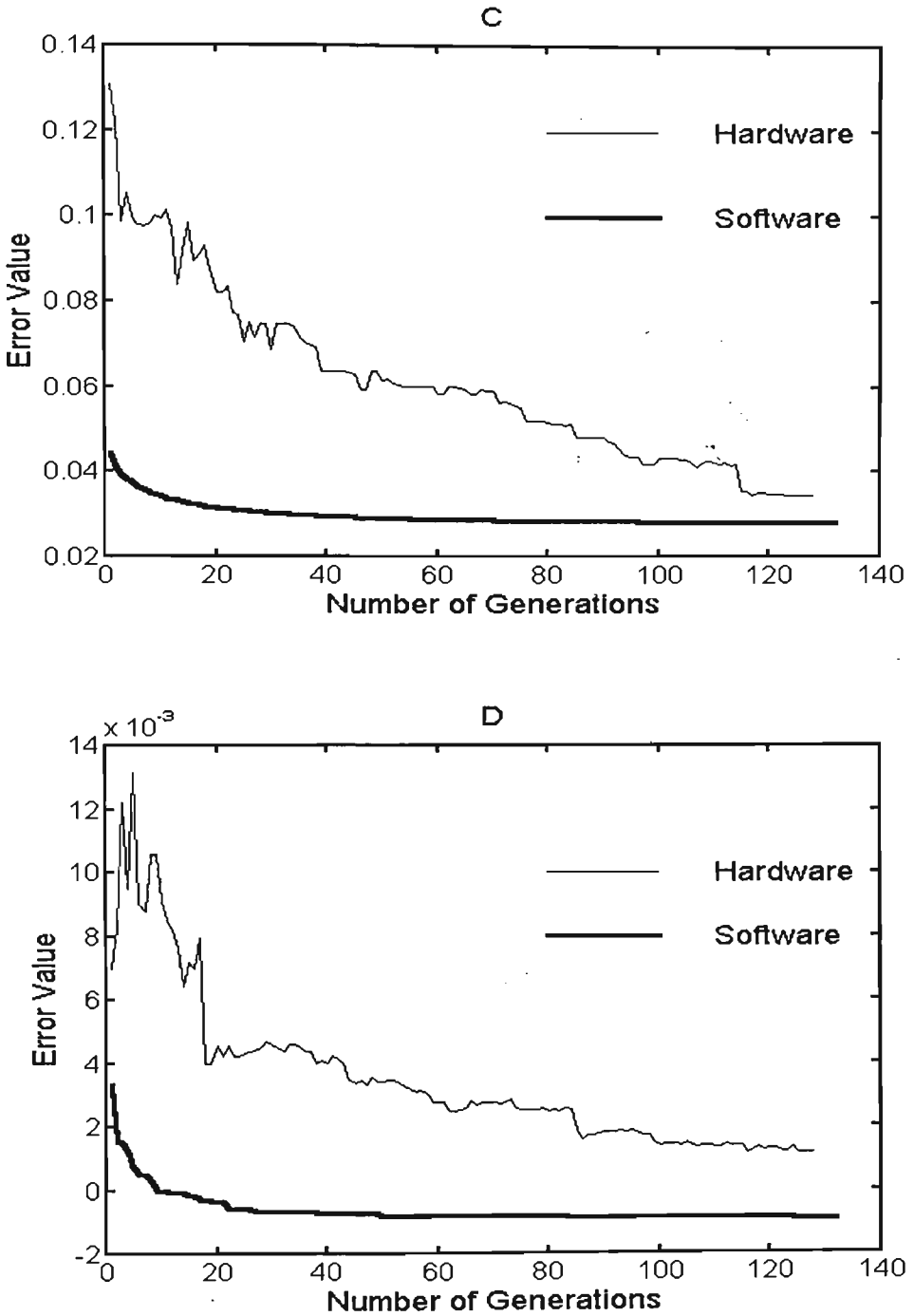


Figure 6.11: The error value of the best individual versus number of generations (C =Step, D =Quartic).

• The software GA's selection procedure was altered to choose two new members per function call instead of one member per call. Thus the software GA's selection procedure more closely resembles the GAP's selection module described in Section 4.2.3.6.

• The software GA's mutation procedure was changed to more closely resembles mutation in the GAP. The GAP's mutation probability of P_m means that there is a P_m probability that between the two crossed members, one bit will be flipped. In the original software GA this probability applies to each bit of the crossed members. The software GA's mutation procedure was altered to reflect the GAP's mutation operation.

The GAP was compared with the software GA when optimising the fitness functions $f(x)=x^{10}$ over the discrete domain $D=\{x|-8\leq x\leq 7\}$. The GAP and software GA both ran with 8 bit members and 8 bit fitness values.

Both the software GA and GAP are started with the same initial population, so the only variations in the runs were in the pseudorandom number generation.

The results of the runs are shown in the Figure 6.12 and Table 6.6. The table presents the average execution times of the software GA and GAP in milliseconds. All I/O times are removed from the comparisons. The GAP prototype, clocked at 10 MHz, ran an average 5.2 times faster than the software GA. The range of speed improvements varies from 3.5 to 7.6. Considering the difference between clock frequencies (66/10=6.6), the speed improvements are now range from 23 to 50 with the average of 35 (Table 6.7).

Populations	Generations 16	Generations 32	Generations 64	Generations 128	Hardware speed improvement
16(SGA)	15.38ms	30.77ms	61.48ms	122.97ms	----
16(GAP)	2.00ms	3.99ms	8.06ms	16.11ms	7.63
32(SGA)	38.79ms	77.53ms	155.06ms	310.39ms	----
32(GAP)	7.05ms	14.10ms	28.2ms	56.4ms	5.5
64(SGA)	109.62ms	219.29ms	438.46ms	876.92ms	----
64(GAP)	26.20ms	52.40ms	104.95ms	210.00ms	4.18
128(SGA)	347.86ms	695.71ms	1390.66ms	2834.48ms	----
128(GAP)	101.44ms	202.05ms	405.07ms	809.85ms	3.5

Table 6.6: Timing results of the software GA and the GAP on different fitness functions. The GAP was clocked at 10 MHz, the software GA at 66 MHz.

Populations	16	32	64	128	Average
Hardware speed improvement	7.63	5.5	4.18	3.5	5.2
Overall speed improvement	50.4	36.3	27.6	23.1	35

Table 6.7: Overall speed improvements.

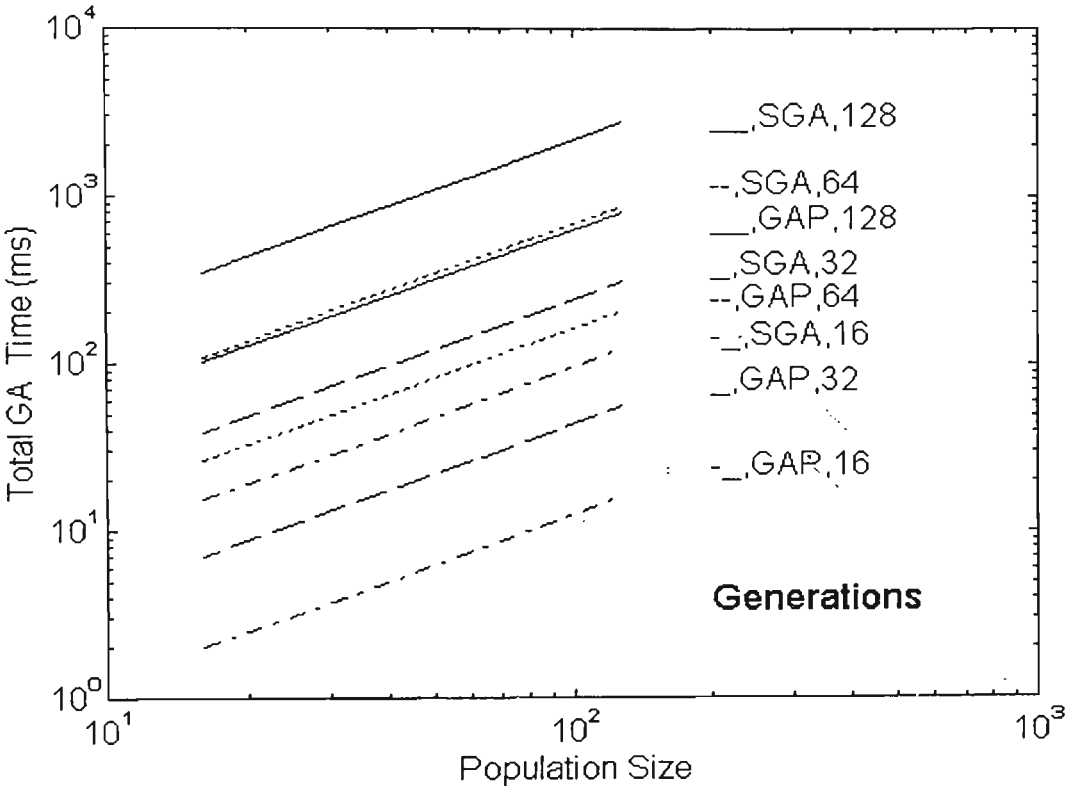


Figure 6.12: Total run time for the GAP and Software GA (SGA) for different population sizes and generations.

Chapter 7

Application of the GAP in engineering

This chapter describes investigations into the potential use of the GAP in three different applications all of which involve "parameter tuning". In each case the process being controlled is approximated by either a polynomial equation or the ratio of two polynomials. The role of the GAP is to update the control coefficients in the face of non-linearities or changes in the plant or process behaviour.

The first application is a Proportional-Integral-Differential (PID) controller system which can be optimised with the GAP. The second application is the Economic Power Dispatch problem where the GAP optimises load distribution between generators. The third is an Adaptive IIR Filter tuned by the GAP. In all applications various bit length configurations are tested to examine the effect of member length in solving problems.

7.1 Application in a PID controller

The conventional PID regulator, because of its remarkable effectiveness, simplicity of implementation and broad applicability, is the most widely used digital control strategy in use today [Ogata, 1990].

In practice, designing PID controllers is often carried out by an experienced operator using a 'trial and error' procedure and applying some practical rules. This is a time consuming and difficult activity when the dynamic process is slow, partly nonlinear, contains significant dead-time, or is subjected to random disturbances. Once designed, the control performance may later deteriorate because of nonlinear or time-varying characteristics of the process under control. Although PID controllers are common and well known, they are often poorly tuned [Dorf, 1991].

Modern adaptive control algorithms can be a good solution to such problems. They are able to self-tune the controller and to adapt it to changes in the process, provided certain conditions are fulfilled [Paraskevopoulos, 1988]. The introduction of these controllers in industry may cause some resistance and difficulties, mostly related to the lack of knowledge by the operating personnel about their internal mechanisms. An attractive alternative is to try to combine the well-known PID controller with algorithms which are able to provide on-line a set of optimal PID parameters, using input/output data from the system.

7.1.1 The PID controller system

One form of controller widely used in industrial process is called a three-term or process controller. This controller has a transfer function

$$G(S) = \frac{U(S)}{E(S)} = K_p + \frac{K_i}{S} + K_d S \tag{7.1}$$

The controller provides a proportional term K_p, an integration term K_i, and a derivative term K_d and is also called a PID controller. The equation for the output in the time domain is

$$u(t) = K_p\, e(t) + K_i \int e(t)\, dt + K_d \frac{de(t)}{dt} \tag{7.2}$$

If we set K_d=0, then we have the familiar PI controller. When K_i=0, we have

$$G(S) = K_p + K_D\, S \tag{7.3}$$

which is called a proportional plus derivative (PD) controller.

Many industrial processes are controlled using PID controllers. The popularity of PID controllers can be attributed partly to their robust performance in a wide range of operating conditions and partly to their functional simplicity, which allows engineers to operate them in a simple, straightforward manner. To implement such a controller the three parameters must be determined for a given process.

In PID control we attempt to derive a plant in accordance with a given reference signal (Figure 7.1). If a mathematical model of the plant can be derived, then it is possible to apply various design techniques for determining parameters of the controller that will meet the transient and steady state specifications of the *closed loop* system. However, if the plant is so complicated that its mathematical model cannot be easily obtained, then an analytical approach to the design of a PID controller is not possible and we must resort to experimental approaches. The process of selecting the controller parameters to meet given performance specifications is known as controller tuning.

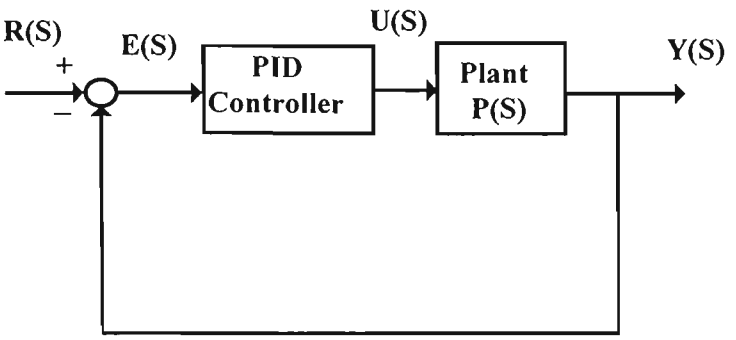


Figure 7.1: A typical PID controller system.

The three gain parameters (Ki, Kp, Kd) of the PID control law interact with the plant parameters *P(S)* in a complex fashion when the designer attempts to derive the specified roots of the closed loop equation. These roots are chosen in order to obtain the desired transient response of the closed loop, while taking the resultant zeros into account.

The controller introduces a new pole at the origin of the s-plane and shifts the original roots of the closed loop system to new positions. PID controllers increase the order of the closed loop equation by one. In addition to these effects, PID controllers introduce a pair of zeros, usually a complex conjugate pair, which will normally have a significant effect on the transient behaviour of the compensated system.

Designing PID controllers, even for low order plants such as a robot arm, can be a difficult problem. Consider the system illustrated in Figure 7.1 where the PID controller obeys the following control law:

$$U(S) = \left(K_i / S + K_p + K_d * S\right) E(S) \tag{7.4}$$

where *Y(S)* is the output of the plant system, *R(S)* is the reference signal, *E(S)* is the error signal equal to *Y(S)* - *R(S)* and *U(S)* is the output of the PID controller. Using the equality $S=(1-Z^{-1})$, (7.4) can be expressed in the *Z* domain as:

$$U(Z) = \left(K_i / \left(1 - Z^{-1}\right) + K_p + K_d * \left(1 - Z^{-1}\right)\right) E(Z). \tag{7.5}$$

The goal of PID controller design is to determine a set of gains, (K_i, K_p, K_d), of the control law such that the set of roots of the closed loop equation chosen by the designer are obtained.

The efficiency of the system can be measured by calculating the integral of the time multiplied by the error for the unit step response during [0,T]:

$$error = \int_{t=0}^{T} t \left|e(t)\right| dt \tag{7.6}$$

The problem confronting the designer, therefore, is to calculate the three gains of the PID controller while ensuring that transient response specifications (minimum error, overshoot, rising time, settling time and steady-state error) are met.

7.1.2 Applying the GAP to a PID controller

The selection of the three coefficients of PID controllers is basically a search problem in a three dimensional space. Points in the search space correspond to different selections of a PID controller's three parameters. By choosing different points of the parameter space, different step responses can be produced for a step input. A PID controller can be tuned by moving in this search space on a trial and error basis.

The main problem in the selection of the three coefficients is that they do not readily translate into the desired performance and robustness characteristics that the control system designer has in mind.

Genetic Algorithm Processor simulations have been conducted for the PID controller system in Figure 7.1. The simulations measured the response Y(t) to the reference signal R(t) which is a unit step function as shown in Figure 7.2. In order to represent a typical

plant to be controlled we use the transfer function described by Hwang and Thompson [1993] as:

$$P(S) = \frac{2}{S(S-1)(S+5)} \tag{7.7}$$

To calculate the fitness value, the transfer function must be converted from the S domain to the Z domain, and to the discrete values (K domain). Then 2000 points are selected between 0 and 10 seconds (T = 10/2000 = 0.005). The fitness value is taken as the integral of time, multiplied by the absolute error values.

$$Fitness = \sum_{k=1}^{2000} \left((kT) * |e(k)| \right) \qquad T = 0.005 \tag{7.8}$$

The following parameters have been selected for the GAP model:

population size	= 64
generations	= 128
fitness value	= 8 bits
crossover rate	= 90%
mutation rate	= 2%
member size	= 24 bits.

Each K value is allocated 8 bits in each member string (or chromosome) which makes a total length of 24 bits. So the memory must have at least 256*24 bit locations. Figure 7.3 and Figure 7.4 demonstrate the result of simulations averaged over ten individual runs. The minimum and maximum K values and the normalised error values are shown after each generation. Figure 7.5 displays the unit step response signal Y(t) when the best set of K values (from the final generation) are applied to the PID controller. The measured characteristics of the response signal are as follows:

Steady State Error	= 0.000
Overshoot	= 1.66 %
Rise Time	= 0.975 Second

Settling Time = 1.455 Second.

The results show that the GAP is able to find good K values very quickly (generally in 20 generations) and the resulting unit step response is acceptable for a PID controller.

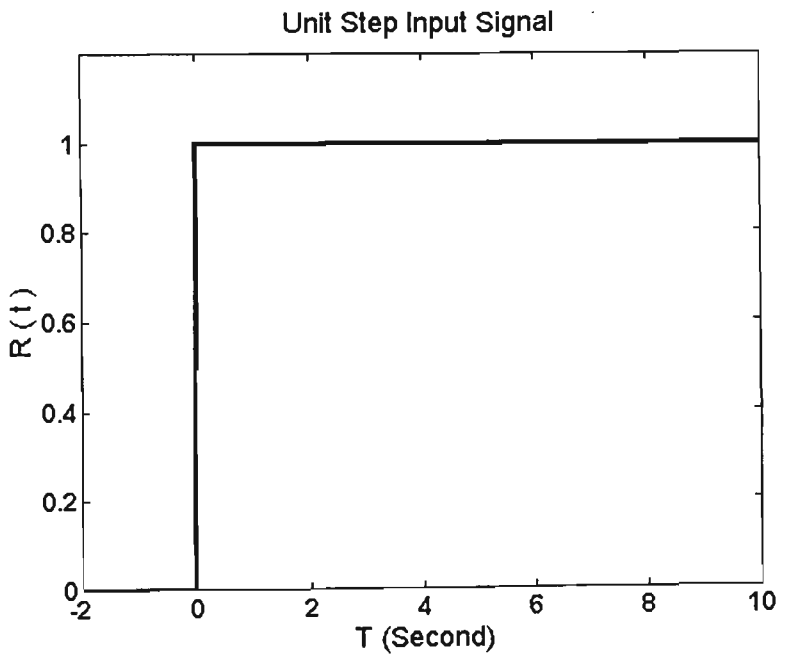


Figure 7.2: The reference signal.

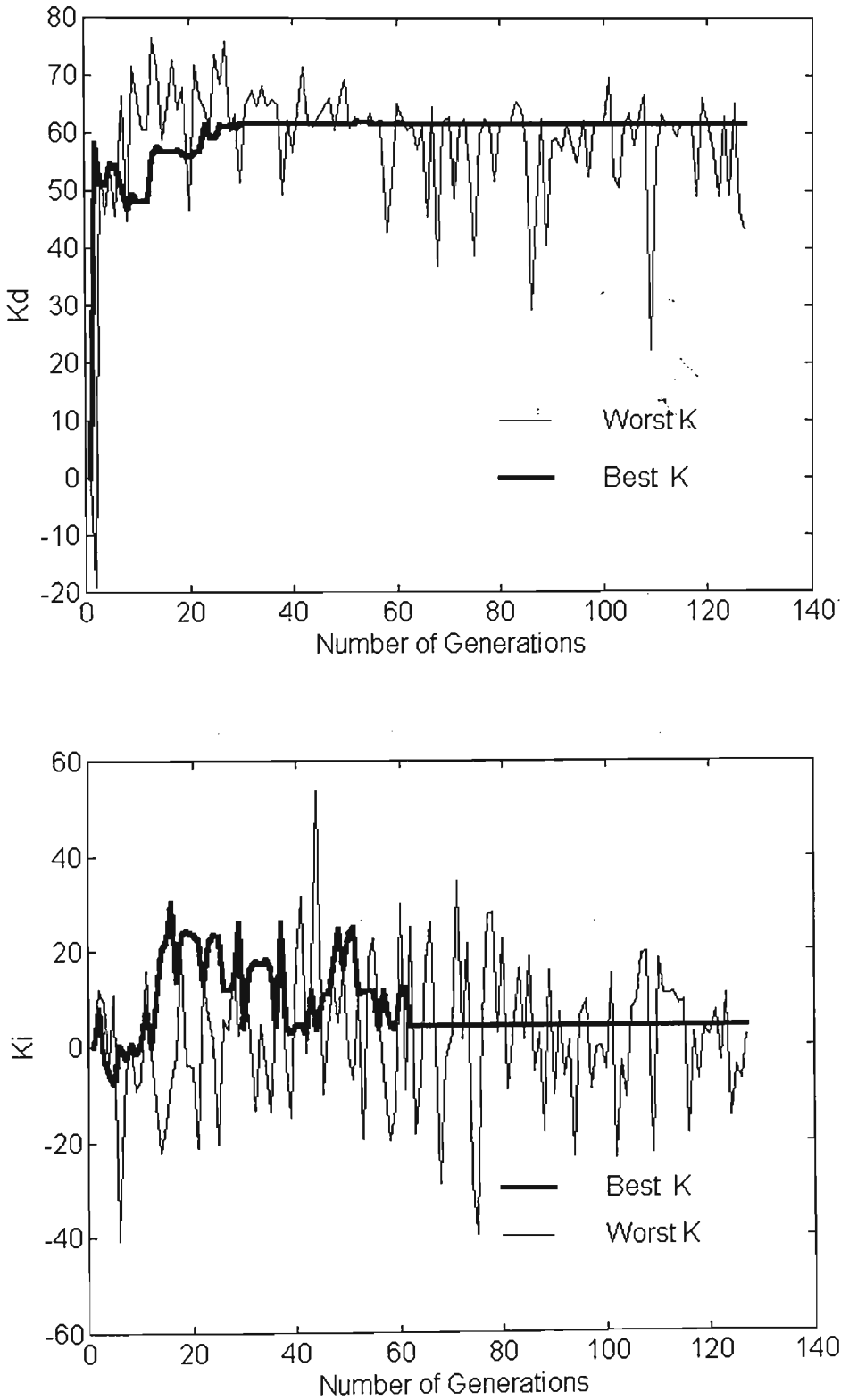


Figure 7.3: The results of GAP simulations for PID controller (Kd and Ki).

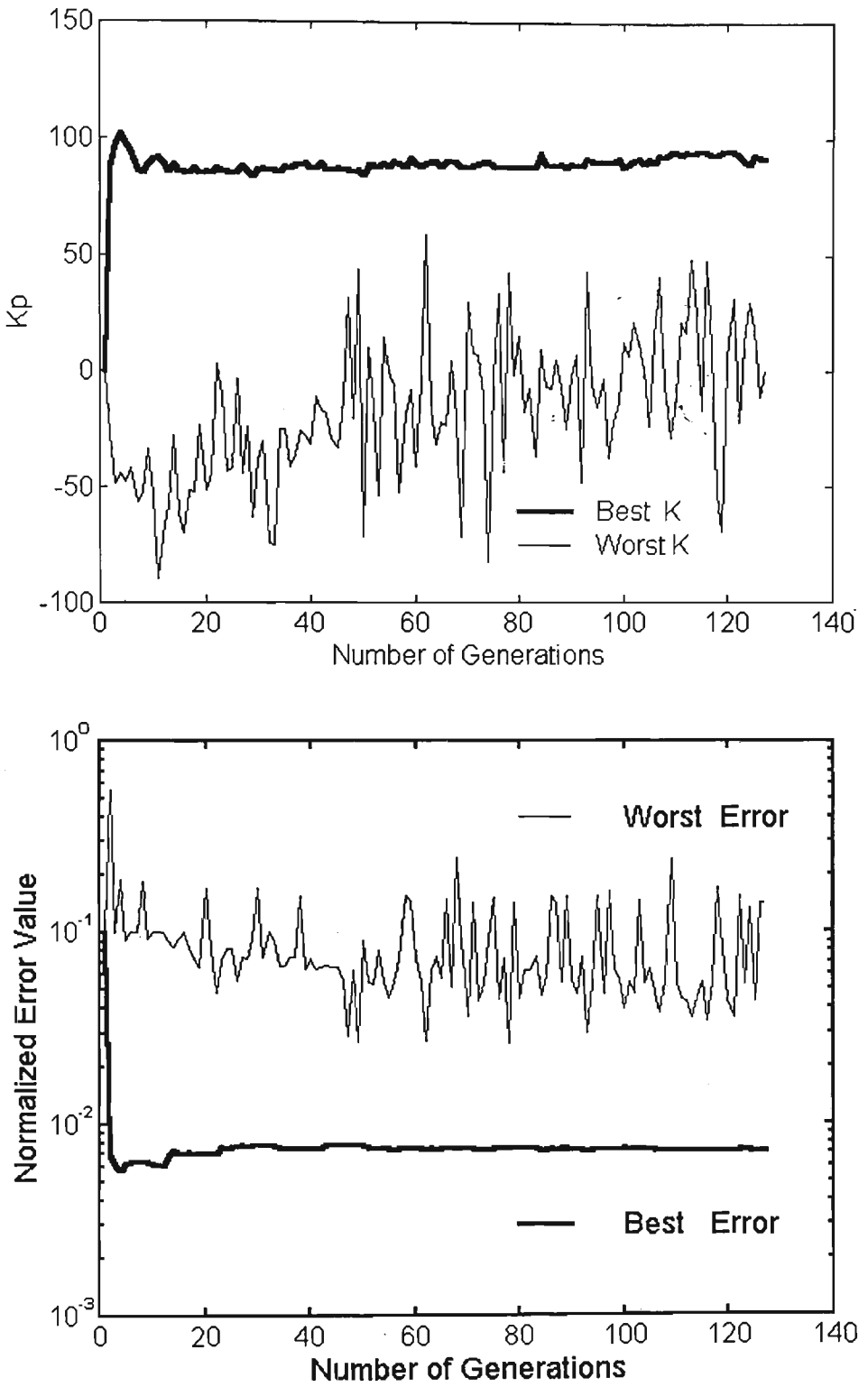


Figure 7.4: The results of GAP simulations for PID controller (Kp and normalised error value).

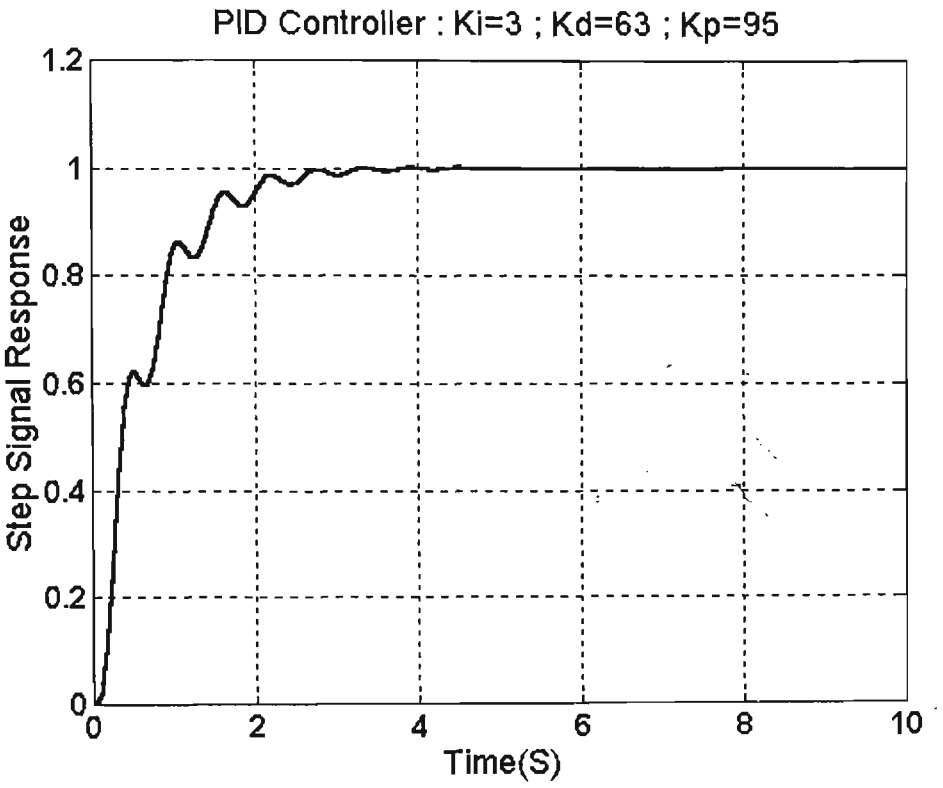


Figure 7.5: The unit step response of the best set of K values.

7.1.3 Other GAP configurations for the PID controller

This section compares the results of simulations for different GAP configurations. Three configurations were tested with member bit lengths of 12, 24 and 36 bits. The following parameters were selected for all GAP configurations:

generations	= 128
population size	= 64
fitness value	= 8 bits
crossover rate	= 90%
mutation rate	= 2%.

In all configurations, one third of the memory bit length in each member is used for each K value. Figures 7.6 to 7.9 show the results of simulations for the GAP averaged over ten individual runs. Ki, Kd, Kp and the normalised error values are shown after each generation.

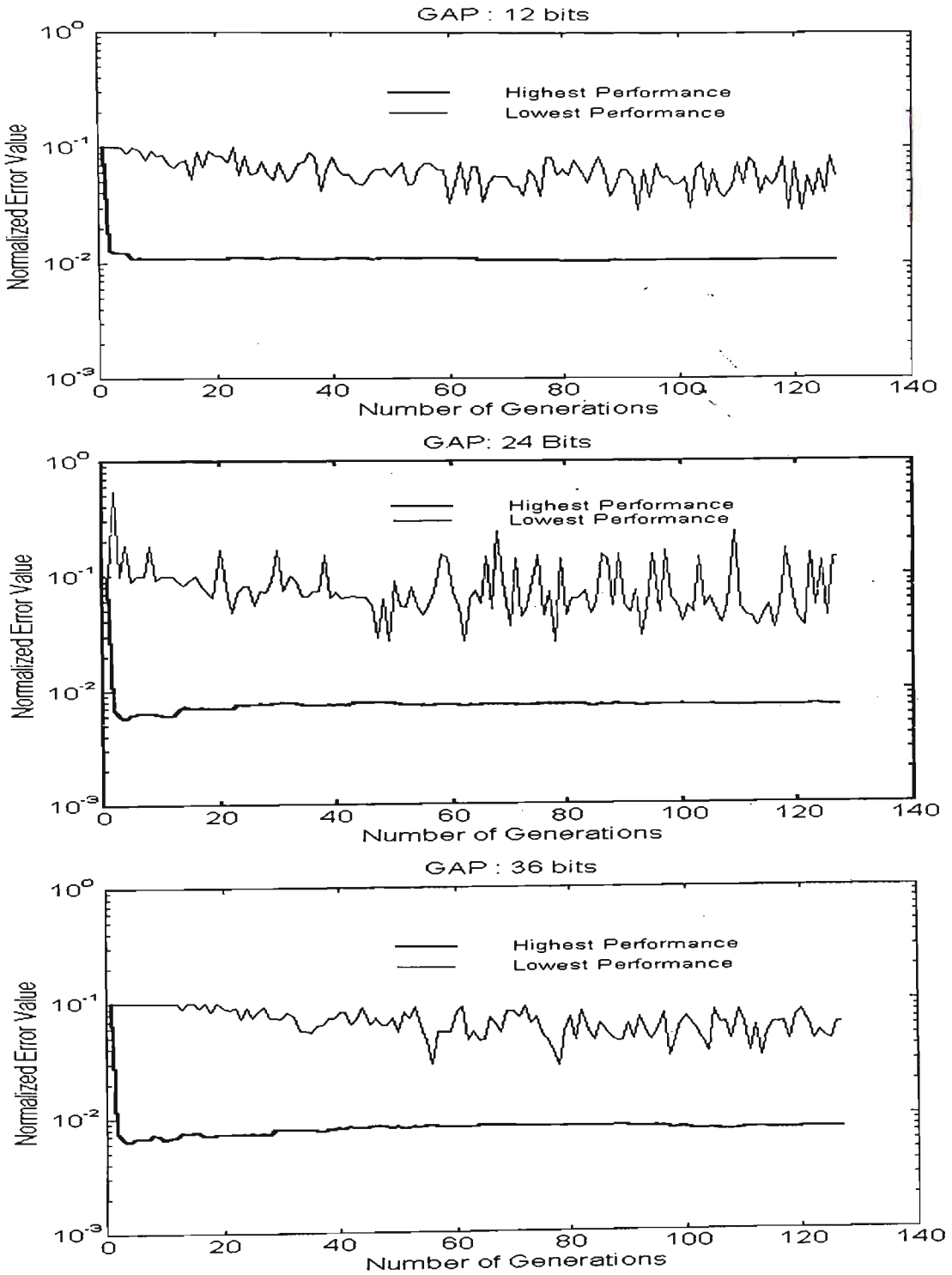


Figure 7.6: The results of the PID controller simulation with 12, 24 and 36 bit configurations (Normalised Error Value).

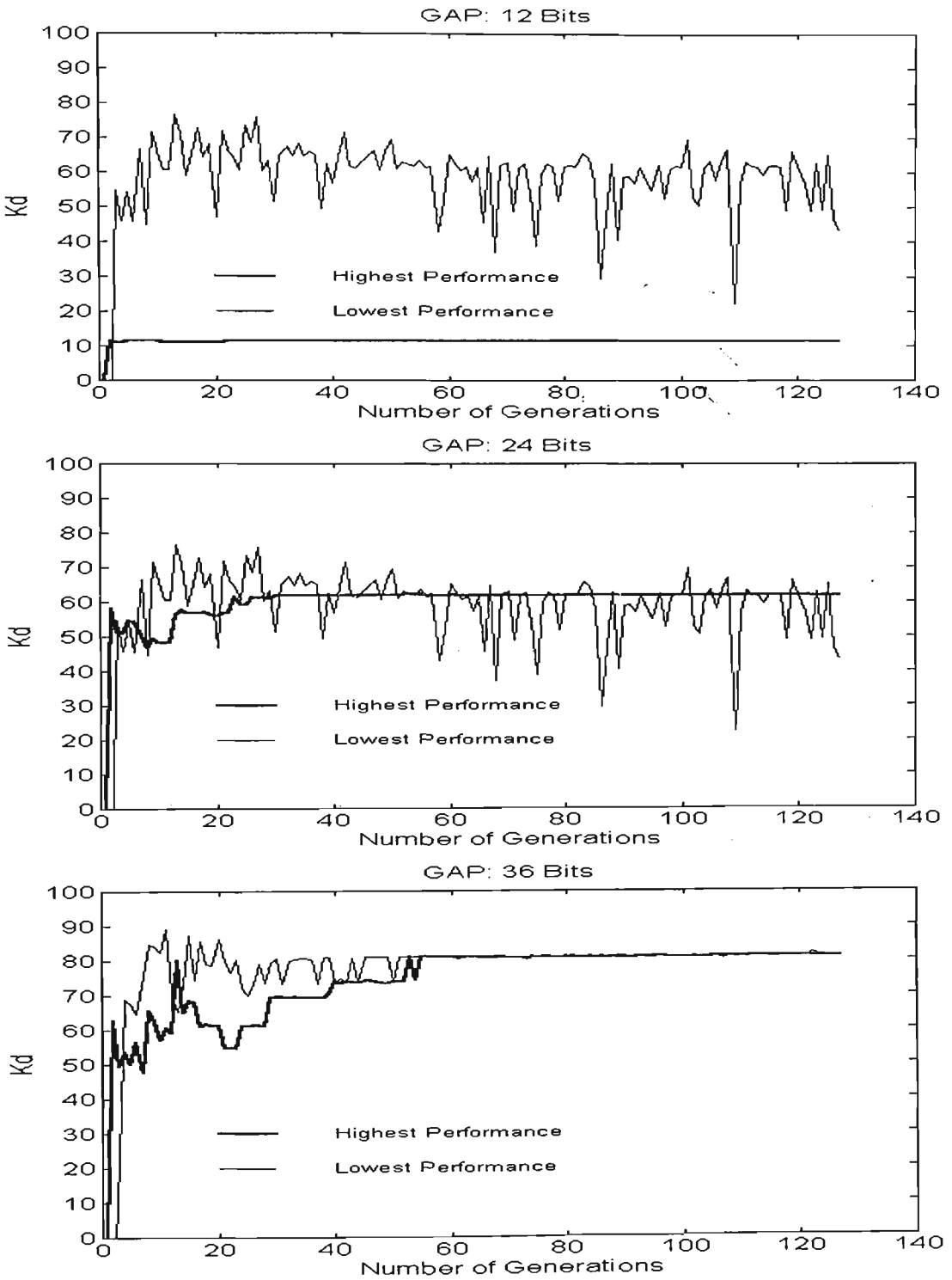


Figure 7.7: The results of the PID controller simulation with 12, 24 and 36 bit configurations (Kd).

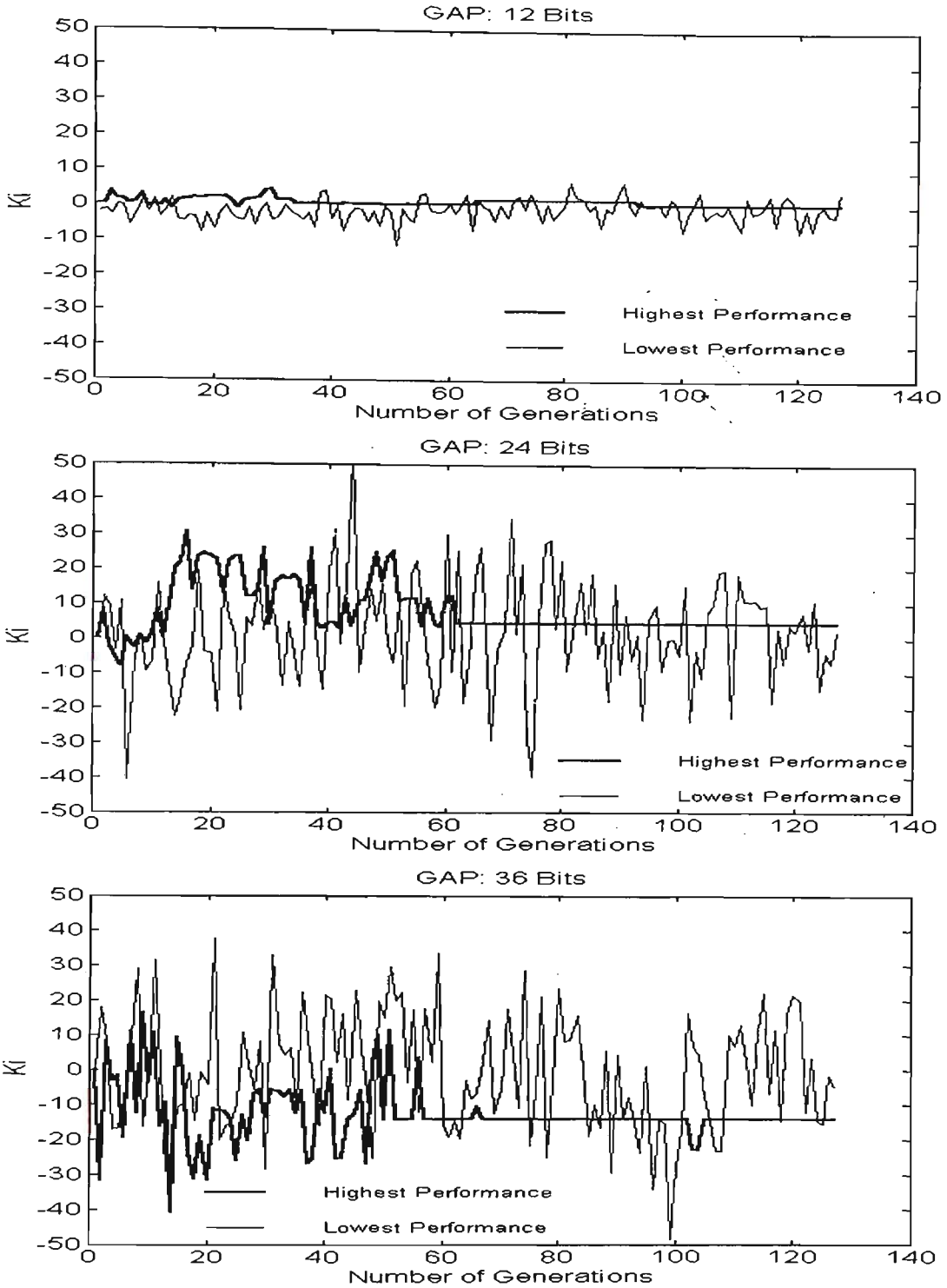


Figure 7.8: The results of the PID controller simulation with 12, 24 and 36 bit configurations (Ki).

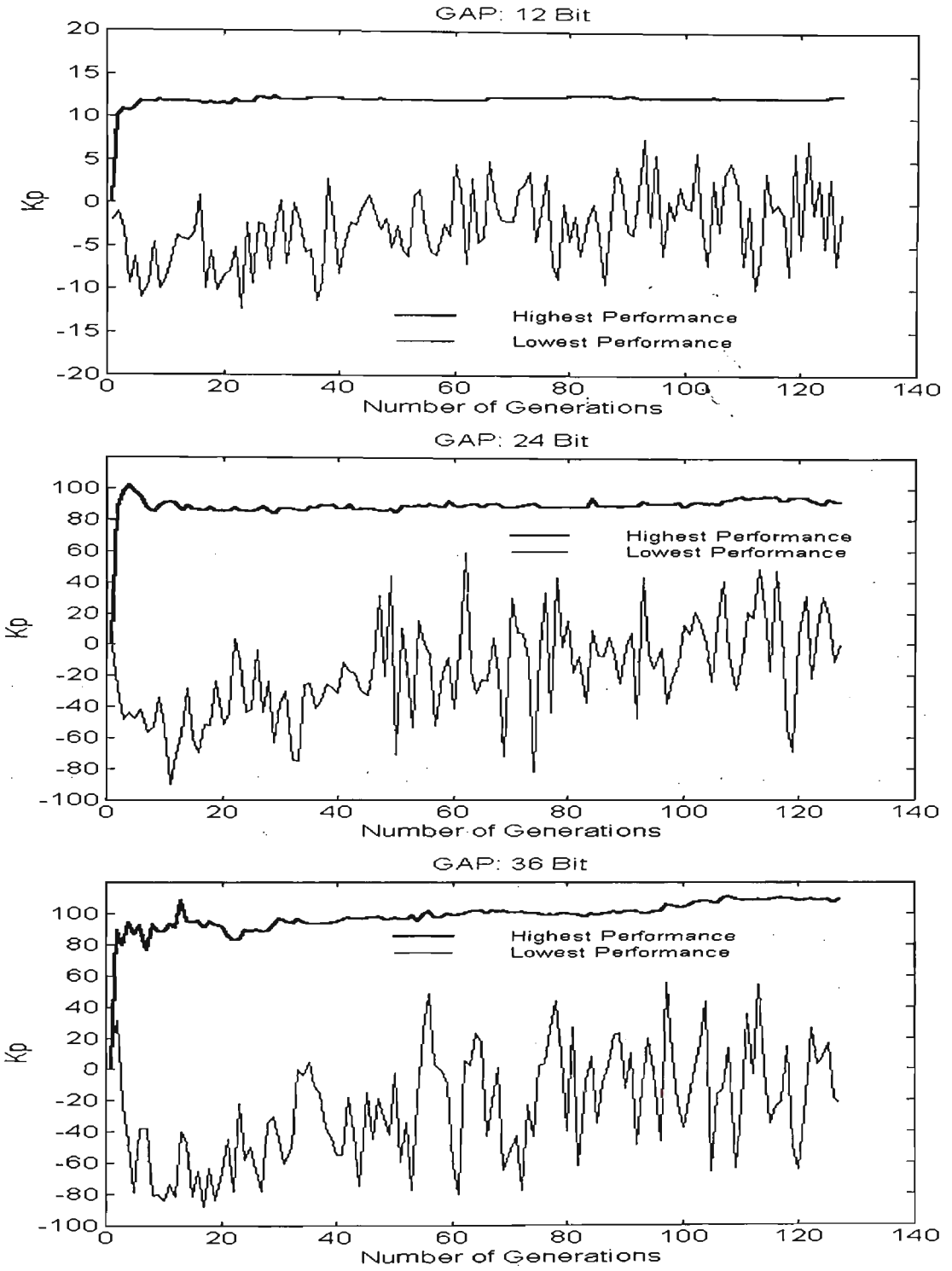


Figure 7.9: The results of the PID controller simulation with 12, 24 and 36 bit configurations (Kp).

Table 7.1 shows the results of comparison of the three different configurations.

Best values in the final generations	12 bits	24 bits	36 bits
Normalised error value	1e-2	7e-3	8e-3
Kd	12	62	81
Ki	2	5	-13
Kp	13	90	110

Table 7.1: The simulation results for different configurations.

From the Figure 7.6, it can be concluded that the 36 and 24 bit processors outperform the 12 bit configuration. In the 12 bit configuration the GAP does not have enough resolution to converge to the best result. There is not much difference between 36 bit and 24 bit configurations, but the 24 bit configuration delivers slightly better performance than the 36 bit configuration. Considering the lower cost for 24 bits, it can be concluded that there is no advantage in using a higher bit length. The speed of convergence in 24 bits is also faster than the 36 bit configuration.

7.2 Application of the GAP in Economic Power Dispatch

With the development of modern power systems the optimal Economic Power Dispatch (EPD) problem has received increasing attention. EPD aims to minimise the fuel cost while providing consumers with adequate and secure electricity. The issue is concerned with economically dividing the loads among the generators when the total capacity

exceeds load demands [Happ, 1977]. This is a typical constrained nonlinear dynamic problem made difficult by an uncertain demand environment.

7.2.1 The EPD problem

Two directions have been pursued in the study of optimal operation of power systems. One is towards an effective computational algorithm and the other is towards the development and formulation of a rigorous theory [El-Hawary and Christensen, 1979]. However, it is realised that conventional optimisation techniques become very complicated when dealing with increasingly complex dispatch problems and are further limited by their lack of robustness and efficiency in a number of practical applications. Thus developing a reliable, fast and efficient algorithm is still an active area of research.

Economic dispatch is mainly concerned with the minimisation of an objective function, usually the total fuel cost $F_T(P)$, while satisfying both the equality and inequality constraints as follows

$$P_T = P_R + P_L \qquad \text{and} \qquad P_{i_{\min}} < P_i < P_{i_{\max}} \tag{7.9}$$

where

- $F_T(P) = \sum_{i=1}^{N} F(P_i)$ is the total fuel cost of generation for N plants.
- P_i is the power generated by the unit i.
- P_T is the total power generation.
- P_R is the total load demand.
- P_L is the total transmission loss $P_L = \sum_{i=1}^{N} B_i P_i^2$.

- $P_{i_{min}}$ and $P_{i_{max}}$ are the lower and upper limits of permitted power generation for unit *i*.

- B_i is the transmission loss coefficient.

The total fuel cost for power generation is given by the equation

$$F(P_i)=a_i+b_i P_i+c_i P_i^2+\left|e_i \sin\left(f_i\left(P_{i_{min}}-P_i\right)\right)\right| \qquad (7.10)$$

where a_i, b_i, c_i, e_i, f_i are the constants of the input-output load curve for the generators.

In most of the conventional methods, the complexity of the problem and its solution procedure are dependent largely on the configuration of the power generators and the number and type of constraints involved. Conventional optimisation techniques are application-dependent and in certain situations, a combination of different methodologies has to be employed for an efficient solution [Sasson and Merrill, 1974]. There is a large and ever increasing number of specific methods in each of these categories and many of these methods have been tried on various issues of economic power dispatch and optimal power flow in different combinations. As a rule, the most powerful optimisation methods are unacceptably slow on problems of large dimensions. Conversely, the faster methods tend to be less reliable in convergence and/or require restrictive application formulations and modeling assumptions [Cohen and Sherkat, 1987].

No practical methods are guaranteed to solve real problems or find a globally optimal solution. The limitations of the optimisation techniques have been a major obstacle to the development of production quality economic dispatch and optimal power flow programs for industrial applications and practical problem formulations. The situation has recently improved with the development of several promising methods [Choudhury and Rahman, 1990], although there are still many obstacles to be overcome.

7.2.2 Applying the GAP to the EPD problem

Genetic Algorithm Processor simulations have been conducted for the optimal economic dispatch of a 3 generator power system described in [Walters and Sheble, 1993]. The constraints and coefficients are shown in Table 7.2.

Parameter	Unit 1	Unit 2	Unit 3
Maximum	600 MW	450 MW	250 MW
Minimum	100 MW	150 MW	100 MW
a	0.001562	0.00194	0.00482
b	7.92	7.85	7.97
c	561	310	78
e	300	200	150
f	0.0315	0.042	0.063

Table 7.2: Coefficients for generators in the simulations.

The loss power is ignored in all simulations and the following parameters were selected for the GAP:

generations = 128
population size = 32
fitness value = 8 bits
crossover rate = 90%
mutation rate = 2%
member size = 32 bits.

The demand power is fixed at 850 MW. Half of each member is assigned to the power generated from generator one and two (P1 and P2). The power for generator 3 (P3) can be calculated as P3=850-P1-P2. Each member is 32 bits wide thus P1 and P2 are allocated 16 bits each. Figure 7.10 plots the minimum and maximum cost after each generation averaged over ten individual runs. The actual minimum cost achievable is about 8200 MBtu/hr. The GAP needs about 20 generations to find a reasonably good

answer and after that it tries to optimise the answer. On average, in the final generation, the GAP was able to find the minimum of 8343 which is about 2% off the optimum. The best minimum cost in all generations for this configuration was 8212.30 MBtu/hr.

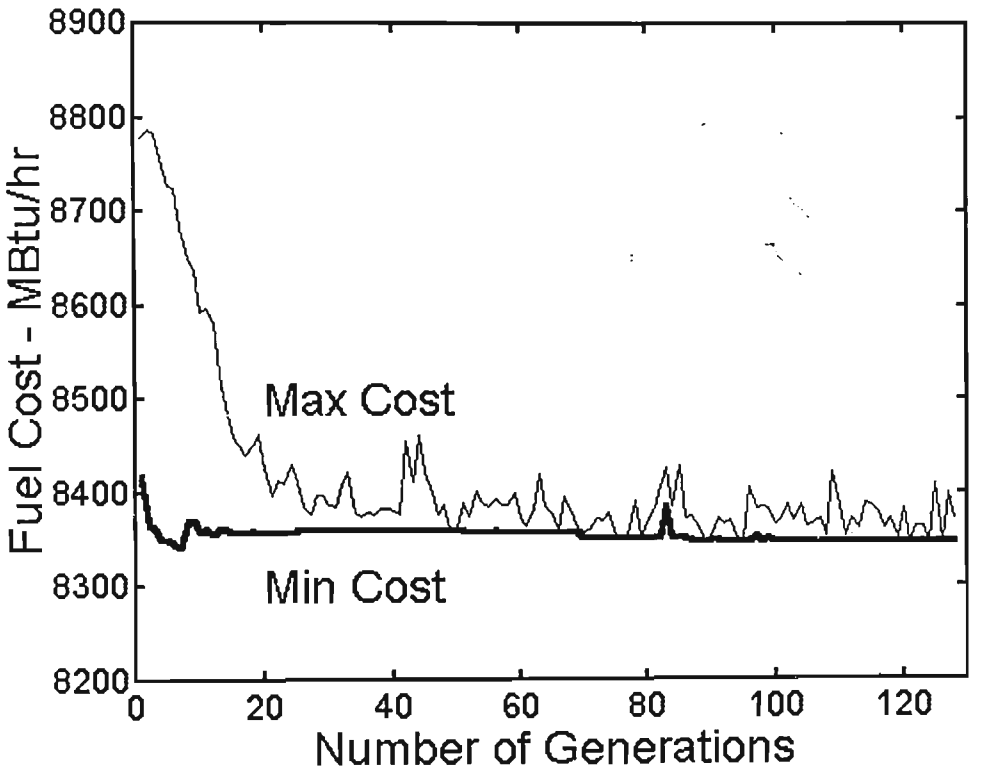


Figure 7.10: Cost versus number of generations for the best and worst individual in the population.

7.2.3 Other GAP configurations for the EPD problem

The simulation in this section was repeated for three different member sizes in the GAP. The constraints and coefficients are shown in Table 7.2. For all cases the demand power is 850 MW and the loss power is ignored. The following parameters were selected for all GAPs:

generations	= 128
population size	= 64
fitness value	= 8 bits
crossover rate	= 90%
mutation rate	= 2%.

Figures 7.11 to 7.13 show the result of simulations for configurations of 32, 16 and 8 bit members averaged over ten individual runs. In the figures the minimum and maximum costs are shown after each generation. In all configurations half of the member string represents P1 and the other half represents P2. Table 7.3 shows the best results and the average results for the three configurations after the final generation.

Costs	32 bit GAP	16 bit GAP	8 bit GAP
Best ever minimum	8212.30	8211.96	8248.23
Average minimum	8343	8271	8333

Table 7.3: The best ever minimum costs and the average minimum costs.

From Figures 7.11, 7.12 and 7.13 and Table 7.3, it can be concluded that the 16 bit processor converges to the best result. The 8 bit GAP does not have enough resolution to converge to a good result. The 32 bit configuration converges quickly but to a poor result. It is evident that the intermediate configuration of 16 bits produces better results than the 32 bit configuration.

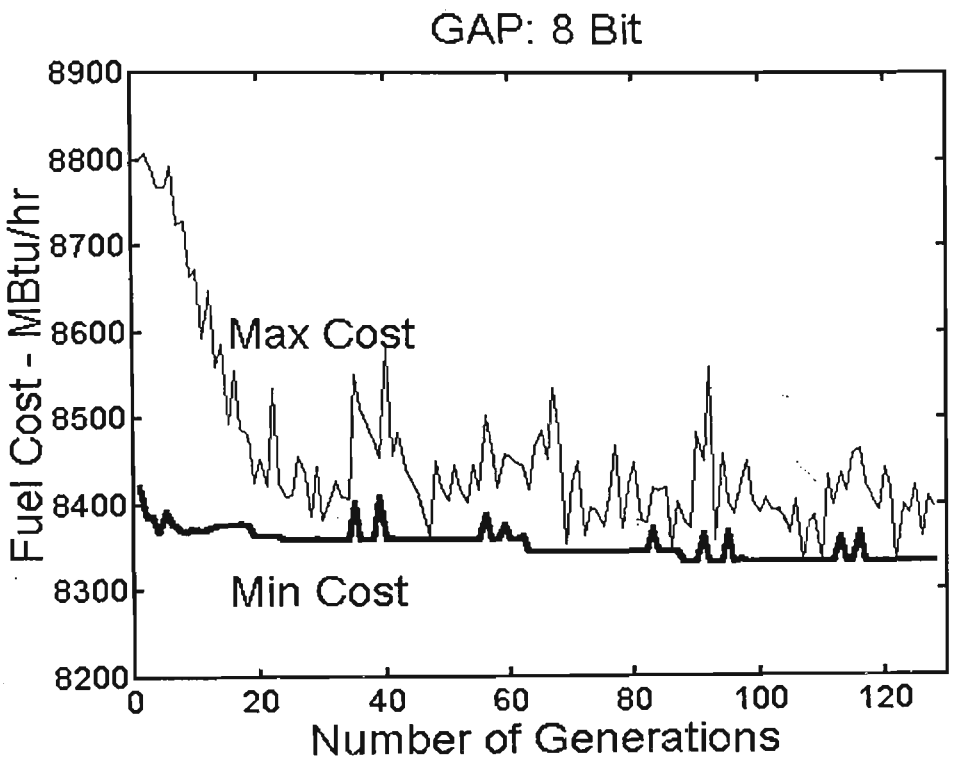


Figure 7.11: Maximum cost and minimum cost versus number of generations for 8 bit members.

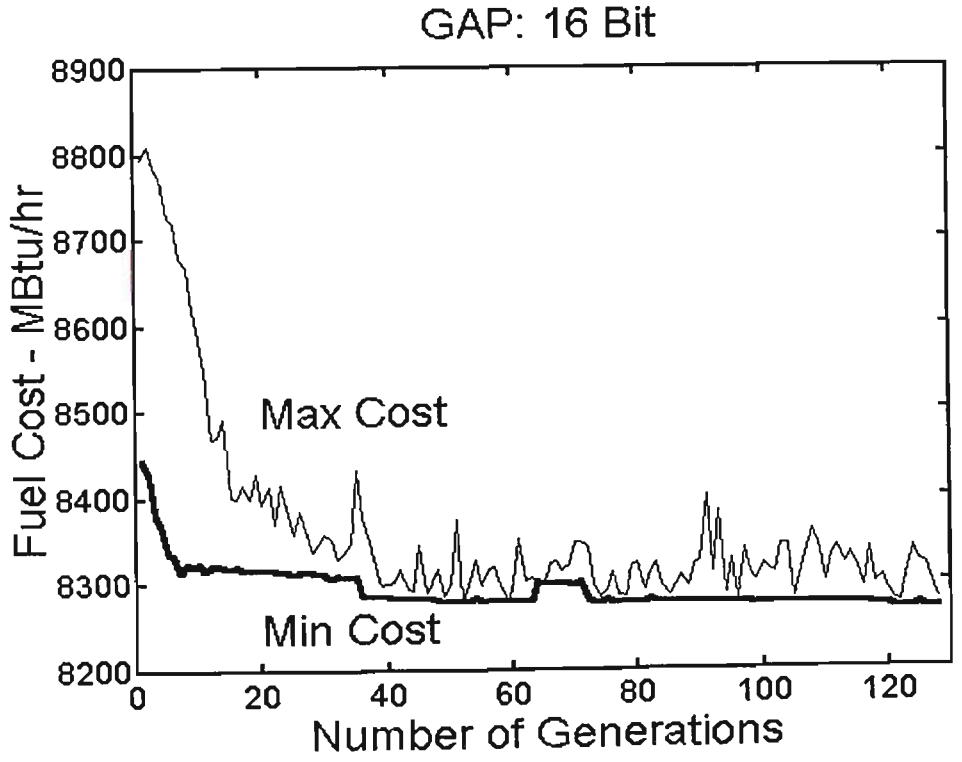


Figure 7.12: Maximum cost and minimum cost versus number of generations for 16 bit members.

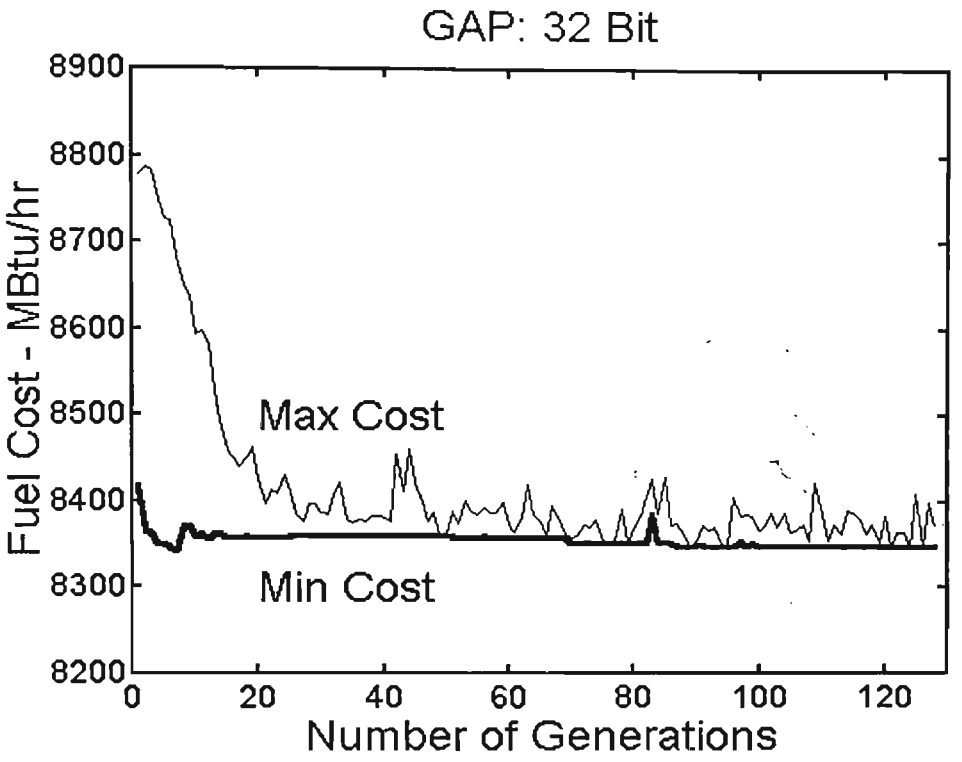


Figure 7.13: Maximum cost and minimum cost versus number of generations for 32 bit members.

7.3 Application in adaptive IIR filters

In recent years there has been a growing interest within the communications industry in using adaptive filters. Applications of adaptive filters have been reported for channel equalisation, noise cancellation and echo cancellation [Willsky, 1985]. In each application the task of adaptation is essentially the same: the adaptive filter is adjusted to match a desired system transfer function and hence may be regarded as a variant of the system identification problem. The common objective is to minimise a performance criterion, usually the Mean Square Error (MSE) between the adaptive filter output and a desired response. Thus, one of the fundamental problems in adaptive filter research is to devise suitable algorithms to alter the filter coefficients to minimise the MSE.

7.3.1 Properties of Infinite Impulse Response Filters

Digital filters with an Infinite-duration Impulse Response (IIR) have characteristics that make them useful in many applications. This section develops and discusses the properties and characteristics of these filters.

Because of its feedback architecture, the IIR filter is also called a recursive filter. In contrast to the Finite Impulse Response (FIR) filter with a polynomial transfer function, the IIR filter has a rational transfer function consisting of a ratio of two polynomials. This means it has finite poles as well as zeros, and the frequency-domain design problem becomes a rational function approximation problem. This contrasts with the polynomial approximation of an FIR filter and gives considerably more flexibility and power, but brings with it certain problems in both design and implementation [Roberts and Mullis, 1987].

The defining relationship between the input and output variables for the IIR filter is given by (Figure 7.14)

$$y(n) = -\sum_{k=1}^{N} w(k)\, y(n-k) + \sum_{k=0}^{M} v(k)\, x(n-k) \tag{7.11}$$

The first summation is a weighted sum of the previous N output values and the second summation is the average of the present plus past M values of the input $x(n)$. The calculation of each output term $y(n)$ from (7.11) requires $N+M+1$ multiplications and $N+M$ additions.

The output of an IIR filter can also be calculated by convolution.

$$y(n) = \sum_{k=0}^{\infty} h(k)\, x(n-k) \tag{7.12}$$

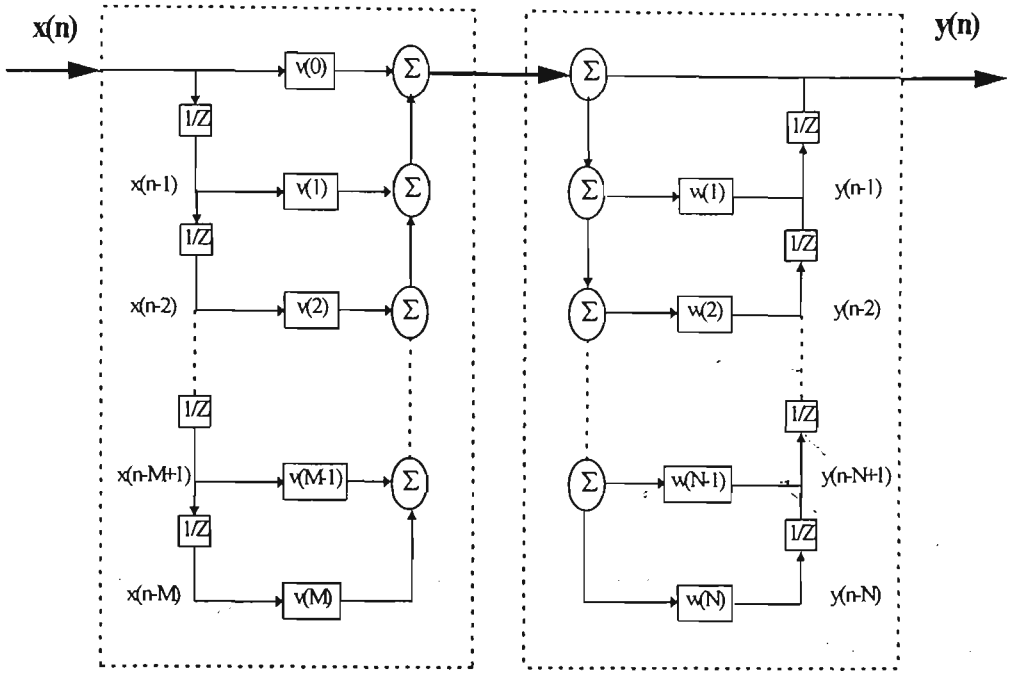


Figure 7.14: Structure of an IIR filter.

In this case the duration of the impulse response *h(n)* is infinite, and therefore, the number of terms in (7.12) is infinite. The *N+M+1* operations required in (7.11) are clearly preferable to the infinite number required by (7.12).

The transfer function of a filter is defined as the ratio *Y(z)/X(z)*, where *Y(z)* and *X(z)* are the *z* transforms of the output *y(n)* and input *x(n)*, respectively. It is also the *Z* transform of the impulse response. Using the definition of the *z* transform, the transfer function of the IIR filter defined in (7.11) can be obtained as:

$$H(z) = \sum_{n=0}^{\infty} h(n)z^{-n} \tag{7.13}$$

This transfer function is also the ratio of the *z* transforms of the *v(n)* and *w(n)* terms.

$$H(z) = \frac{\sum_{n=0}^{M} v(n)z^{-n}}{\sum_{n=0}^{N} w(n)z^{-n}} \tag{7.14}$$

The frequency response of the filter is found by setting $z = e^{j\omega}$, which gives (7.13) the form

$$H(\omega) = \sum_{n=0}^{\infty} h(n)\, e^{-j\omega n} \tag{7.15}$$

This frequency-response function is complex valued and consists of a magnitude and a phase. Even though the impulse response is a function of the discrete variable n, the frequency response is a function of the continuous frequency variable ω and is periodic with period 2π.

The FIR linear-phase filter permits removal of the phase from the design process. The resulting problem is a real-valued approximation problem requiring the solution of linear equations. The IIR filter design problem is more complicated. Linear phase is not possible, and the equations to be solved are generally nonlinear. The most common technique is to approximate the magnitude of the transfer function and let the phase take care of itself. If the phase is important, it becomes part of the approximation problem, which then is often difficult to solve.

The design of a digital filter is usually specified in terms of the characteristics of the signals to be passed through the filter. In many cases the signals are described in terms of their frequency content. For example, even though it cannot be predicted just what a person may say, it can be predicted that the speech will have frequencies between 300 and 4000 Hz. Therefore, a filter can be designed to pass speech without knowing what the speech is. This frequency-domain description is true of many types of signals and noise or interference. For these reasons, among others, specifications for filters are

generally given in terms of the frequency response of the filter. The basic IIR filter design process is:

1. Choose a desired response, usually in the frequency domain.

2. Choose an allowed class of filters—in this case, the *N*th-order IIR filters.

3. Establish a measure of distance between the desired response and the actual response of a member of the allowed class.

4. Develop a method to find the best allowed filter as measured by being closest to the desired response.

The mathematical problem inherent in the frequency-domain filter design problem is the approximation of a desired complex frequency-response function $H_D(z)$ by a rational transfer function $H_A(z)$ with an *M*th-degree numerator and an *N*th-degree denominator for values of the complex variable z along the unit circle of $z = e^{j\omega}$. This approximation is achieved by minimising an error measure between $H_D(\omega)$ and $H_A(\omega)$.

Figure 7.15 illustrates the general structure and the components of an adaptive Infinite Impulse Response filter with input $x(n)$ and output $y(n)$. The IIR filter is characterised by the adjustable coefficients $w(n)$ and $v(n)$, and a recursive algorithm that adjusts these coefficients so that $y(n)$ approximates some desired response $d(n)$, which is determined by the particular application. Figure 7.16 shows the adaptive filter in a system identification configuration, where D is the set of desired system parameters, and $d(n)$ is simply the measured output of the system, which usually includes an additive noise process $V(n)$. The objective of the algorithm is to minimise a performance criterion which is based on the prediction error $e(n)$ (sometimes called the estimation error), defined by $e(n) = d(n) - y(n)$. One commonly used criterion is the mean-square error, $\xi = E[e^2(n)]$, where E is statistical expectation. Another criterion is based on the method

of least squares, and the resulting algorithms are known as recursive Least Mean Squares (LMS) [Widrow and Stearns, 1985].

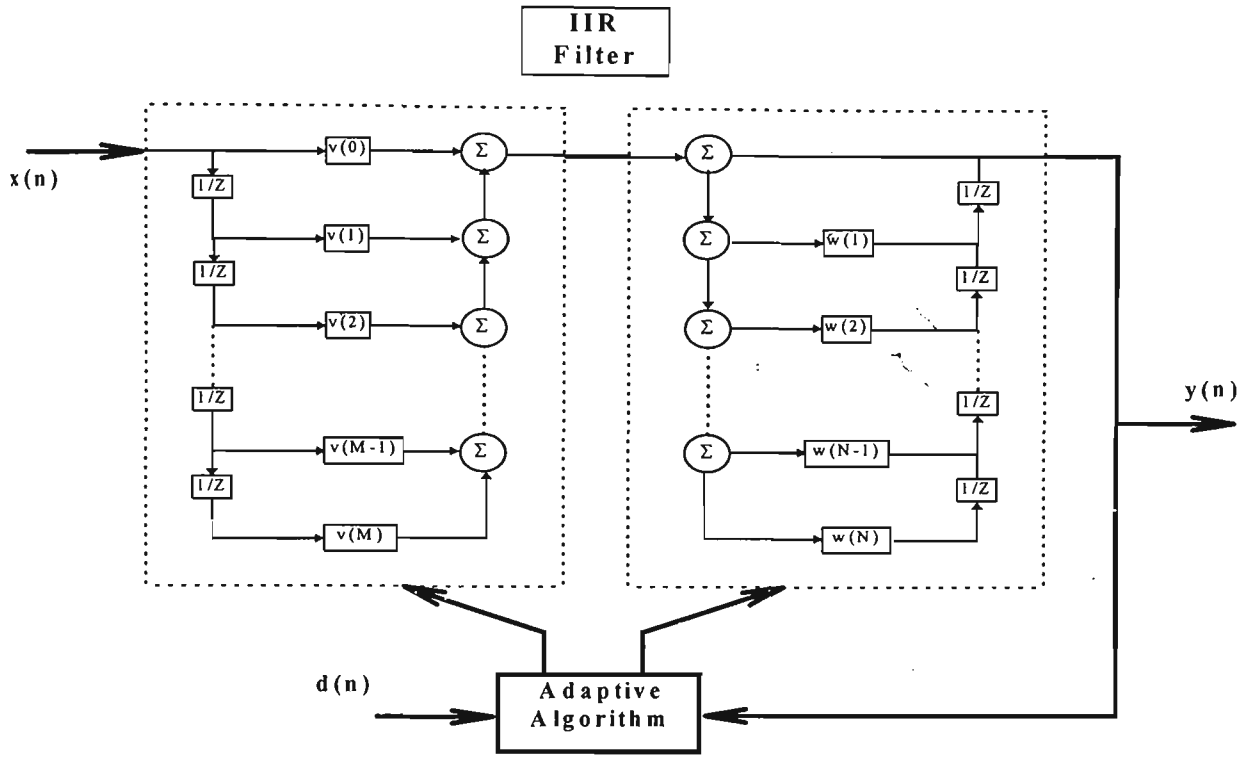


Figure 7.15: Structure of an adaptive IIR filter.

The problem here is to calculate the *w(n)* and *v(n)* coefficients of the filter so that the filter delivers the signal *y(n)* to match the desired response *d(n)*. A novel approach has been suggested to overcome this problem. Instead of applying a deterministic algorithm to search for the minimum of the MSE surface, it was suggested that an intelligent learning algorithm is used [Tang and Mars, 1989]. Specifically, Stochastic Learning Automata (SLA) were considered. This type of automaton is known to have a well-established mathematical foundation and global optimisation capability [Narendra and Thathachar, 1989]. It has been found that this latter capability can be used fruitfully to search a multimodal performance surface [Shapiro and Narendra, 1969]. In this approach the MSE surface is partitioned into a number of hyperspaces and a global search is conducted to find the minimum. Global convergence has been demonstrated in

a well-known reduced-order system identification example where other methods failed to work [Tang and Mars, 1989].

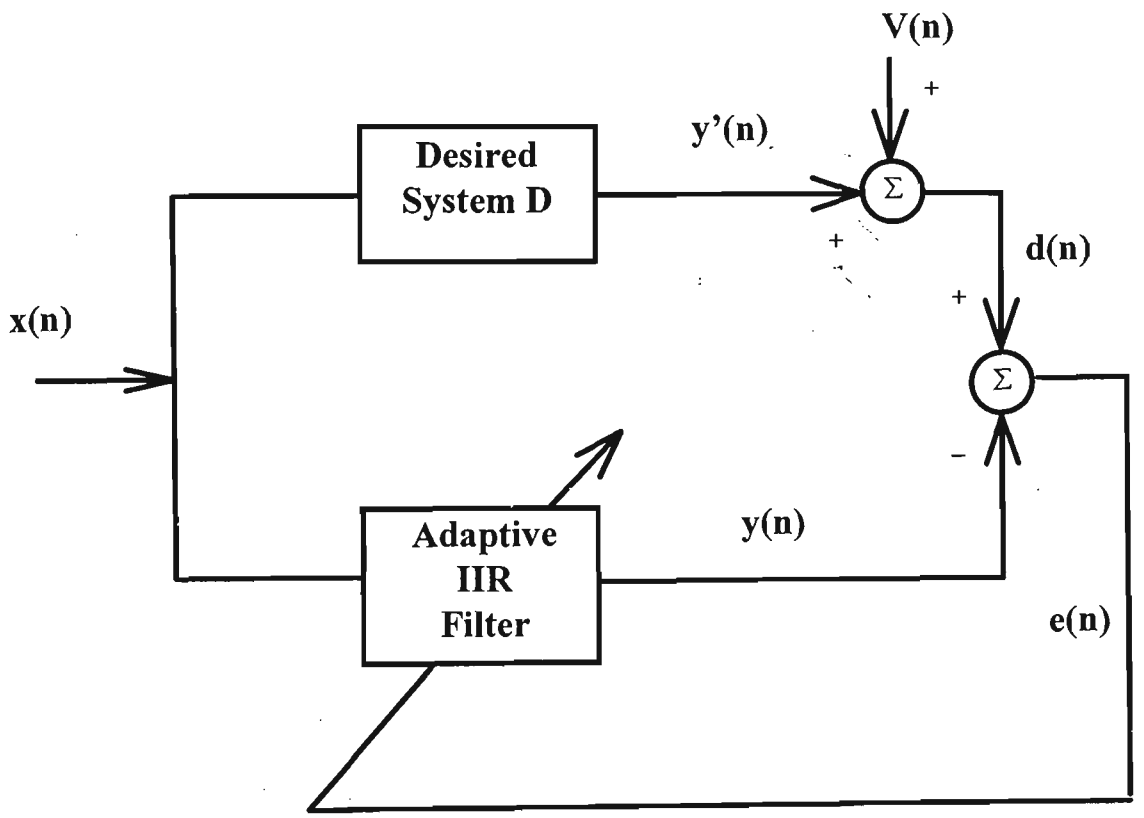


Figure 7.16: A typical system for the adaptive IIR filter.

Since then significant developments have taken place in learning systems. Etter and Masukawa investigated the use of genetic optimisers and linear search algorithms for adaptive delay estimation [Etter and Masukawa, 1981]. They found that when the performance surface is multimodal, or when noise is present, random search algorithms have a better performance than the LMS. Although both genetic optimisers and random search are capable of performing global optimisation, little is known about their use in adaptive IIR filters.

7.3.2 Applying the GAP to adaptive IIR filters

In using a GA for adaptive filtering as in Figure 7.17, the desired system is defined as a fixed IIR filter while the adaptive system is an adaptive IIR filter whose coefficients are updated dynamically by a genetic algorithm.

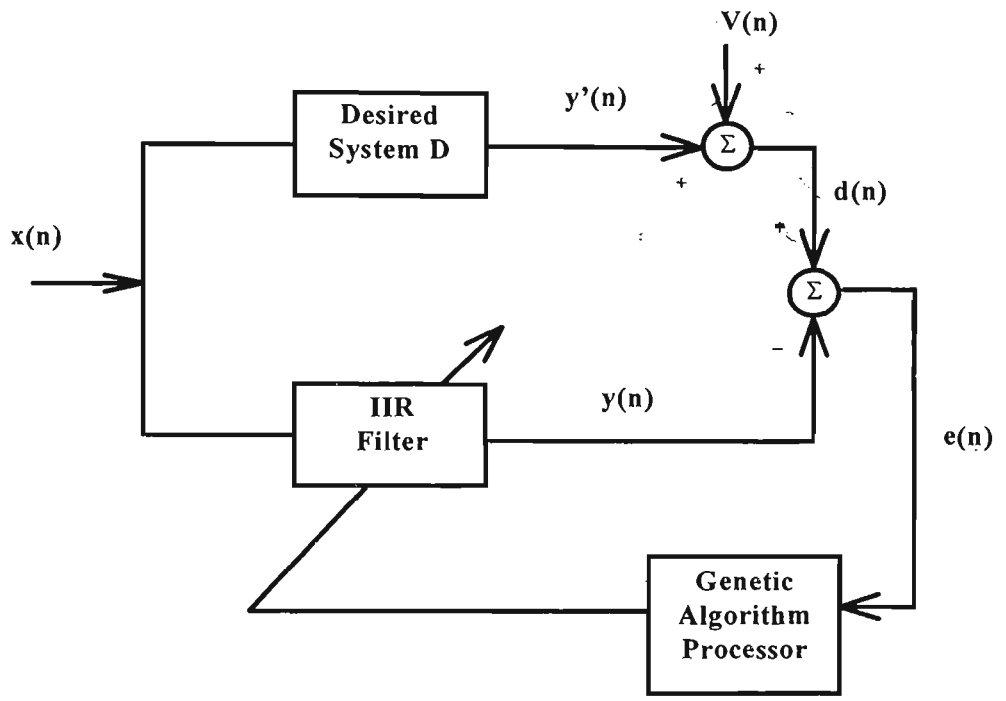


Figure 7.17: The architecture of an adaptive Genetic Algorithm IIR filter.

Tests were conducted to compare the performance of the GAP with the known results for the LMS and SLA algorithms described in the previous section. A reduced-order modelling example is considered, in which a second-order system with a transfer function of:

$$H_D(z^{-1}) = \frac{0.05 - 0.4z^{-1}}{1 - 1.1314z^{-1} + 0.25z^{-2}} \tag{7.16}$$

is modelled by the following first-order filter

$$H_A\left(z^{-1}\right)=\frac{a}{1-bz^{-1}} \tag{7.17}$$

This well-known example was first proposed by Johnson and Larimore [1977] and since then has been considered by others to show that recursive LMS cannot achieve global convergence [Tang and Mars, 1991].

In applying a GA to this problem the fitness value is calculated by converting all transfer functions from the Z domain to discrete values (K domain). The input signal $x(n)$ is white noise consisting of a window of 1000 random normal samples with the standard deviation (sd) equal to one where the additive noise is ignored ($V(n)= 0$). The fitness value is calculated by the Fitness Unit as:

$$Fitness= MSE = \sum_{n=1}^{1000}\left(y(n)-d(n)\right)^2 \tag{7.19}$$

The following parameters are selected for the GAP:

generations	= 128
population size	= 64
fitness value	= 8 bits
crossover rate	= 90%
mutation rate	= 2%
member size	= 16 bits.

All simulations are averaged over ten individual runs started with different initial values. Figure 7.18 compares the result of simulations of the genetic algorithm processor with the LMS and Stochastic Learning Automata. The length of each member is 16 bits and the 'a' and 'b' values are each allocated 8 bits in each member.

This figure shows that the GAP model is much faster than other two algorithms but the SLA is more accurate than the GAP.

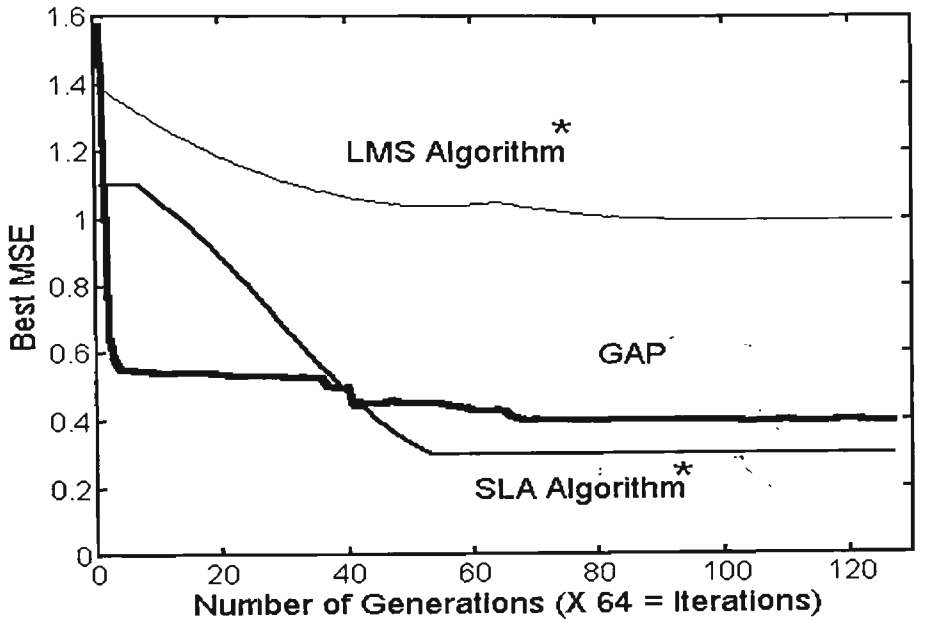


Figure 7.18: The Mean Square Error (MSE) for three algorithms
(* Results from [Tang and Mars, 1991]).

7.3.3 Other GAP configurations for adaptive IIR filters

This section demonstrates the results of simulations for the three different member string lengths in the GAP. The following parameters are selected for the GAP:

generations	= 128
population size	= 64
fitness value	= 8 bits
crossover rate	= 90%
mutation rate	= 2%.

Figures 7.19 to 7.21 show the result of simulations averaged over ten individual runs for GAP configurations of 32, 24 and 16 bits. The figures show the best 'a' and 'b' values and MSE after each generation. In all configurations, half of each member string contains the 'a' value and the other half represents the 'b' value. Table 7.4 summarises the performance of the three configurations. It is apparent that the best results were produced by the 24 bit configuration.

Best values	16 bits	24 bits	32 bits
MSE	0.39	0.36	0.38
'a' value	0.79	0.815	0.81
'b' value	-0.58	-0.54	-0.57

Table 7.4: IIR filter results for three GAP configurations.

From figures 7.19 - 7.21 and Table 7.4, it can be concluded that the 24 bit processor produced consistently better results than the two other configurations. In the 32 bit processor the convergence is faster but the final error is higher than for the 24 bit processor. In the case of the 24 bit processor, there are some oscillations that slow the convergence. The difference in performance is not very large and if the cost of implementing the 32 bit GAP is taken into account then the 16 bit configuration has performed relatively well.

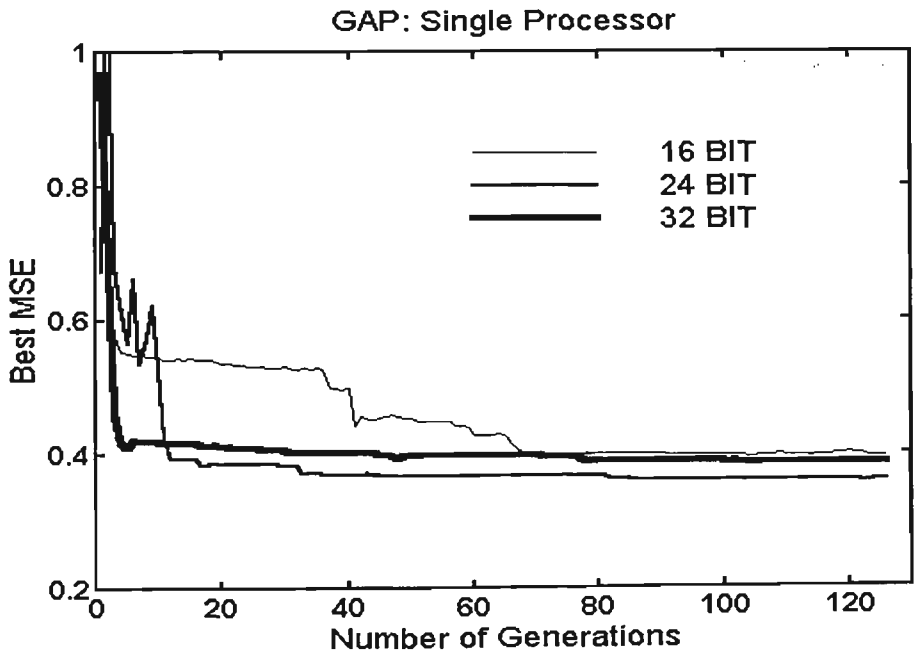


Figure 7.19: The best Mean Square Error (MSE) for the adaptive IIR filters versus number of generations for three GAP configurations (16 bit, 24 bit and 32 bit).

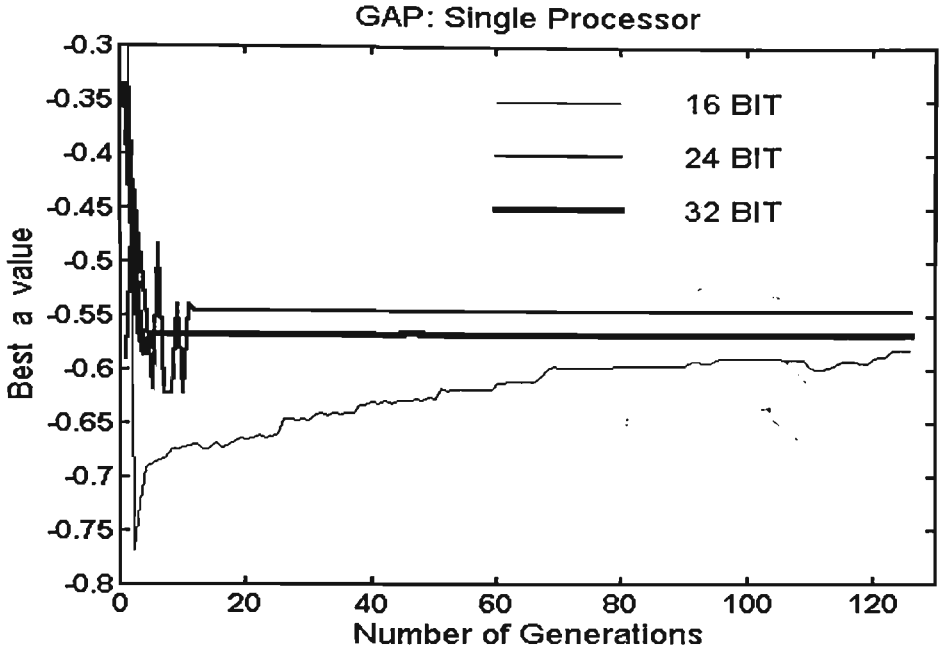


Figure 7.20: The best 'a' value for the adaptive IIR filters versus number of generations for three GAP configurations (16 bit, 24 bit and 32 bit).

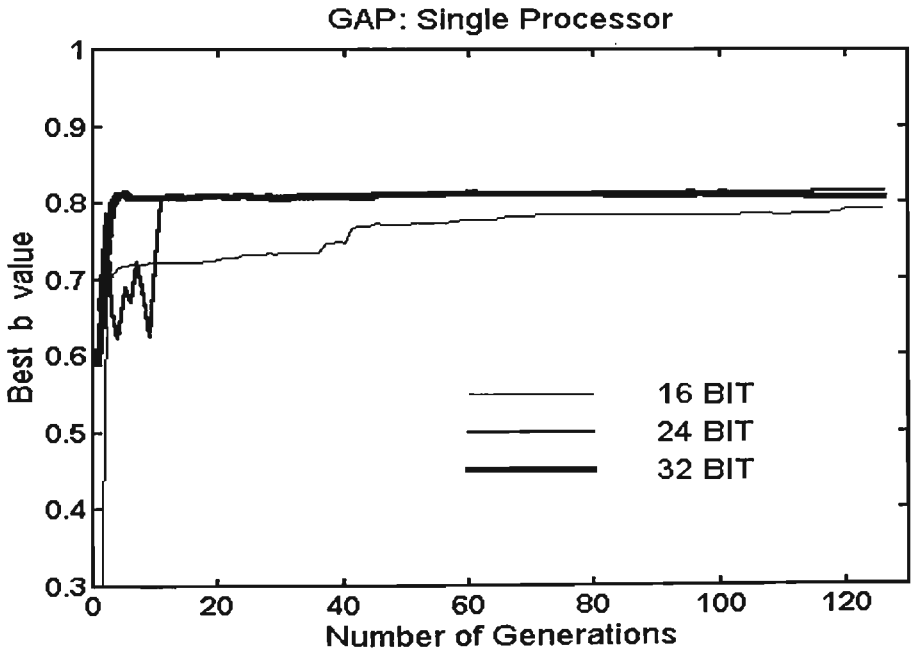


Figure 7.21: The best 'b' value for the adaptive IIR filters versus number of generations for three GAP configurations (16 bit, 24 bit and 32 bit).

7.4 Conclusions

The GAP is capable of handling difficult control and filtering problems without detailed specialised knowledge. A parameter tuning approach is used to adapt a simple linear model to non-linear problems. This appears to avoid the need to have a deep theoretical understanding of the problem and is thus an attractive engineering approach. Further research is required to see how far this approach can be taken.

The GAP does not necessarily need a large member string length to solve a problem. In some applications, the larger bit length configurations seem to have more difficulty in tuning and finding good solutions than the moderate configurations. The small configurations have the additional advantages of higher speed and lower implementation cost. In the next chapter a new configuration of the GAP will be introduced to eliminate the need for implementing large member strings on a single GAP device.

Chapter 8

Multiple GAP architectures

There are technical problems in implementing the GAP with a long member bit string. But the single configuration with small bit string has limited application. This chapter describes the limitations of the single GAP and introduces a new configuration which provides for long member strings while retaining the simplicity of the basic architecture.

8.1 Limitation of a single processor

It was shown in Chapter 6 that as the length of the bit string is increased more FPGA chips are required to implement the design. As the design becomes distributed over several FPGA chips then there is a need for a large number of internal connections between chips. Unfortunately FPGA's have a fixed and limited number of I/O blocks in proportion to the number of CLB's. This means that as the bit length increases we

eventually reach a point where the I/O resources of individual FPGA chips are exhausted and the design cannot be synthesised.

Two problems result from the high levels of connectivity in distributed designs. These are the board layout costs and the additional I/O delays which complicate the design. Interconnection between chips is expensive and increases the board cost.

The delays in FPGAs are mostly due to routing. In most GAP designs about 70% of the critical path delay is due to the routing delay and only 30% to the logic delay. So routing is the major cause of the low processing speed of the GAP. Implementing the GAP with long bit strings means more interconnections and thus greater routing delays and lower speed.

The ideal solution for these hardware limitations is to implement the GAP on a single FPGA chip to minimise the connectivity problem. On the other hand implementing the GAP on one chip means working with small bit strings. However even 8 or 16 bit configurations are not practical for the GAs in real applications.

One way to handle this problem is to connect a linear array of GAPs each handling a small bit string of 8 or 16 bits. This design delivers a large bit string to the problem. Thus eliminating most of the hardware limitations. The performance of this model is investigated in the following sections.

8.2 Multiple architectures

For real applications of the GAP, a means of splitting the bit string of a member between multiple GAPs is needed. One practical way of doing this is by dividing the full member into bit slices. Figure 8.1 shows how a 32 bit member string can be distributed over four GAPs. Each 8-bit slice is assigned to a separate GAP which can be implemented on one FPGA chip. The Fitness Unit operates on a full 32 bit member and delivers an 8 bit fitness value to all GAPs.

All GAPs in Figure 8.1 operate concurrently but at any time only one of them is waiting for a response from the Fitness Unit. Each GAP produces new members and tries to access the FU to calculate the fitness value. Whenever one new member (bit slice) is ready in one of the GAPs (say GAP_A) then GAP_A tests the FU to see if it is free. If so then the GAP_A provides a new member slice on its output.

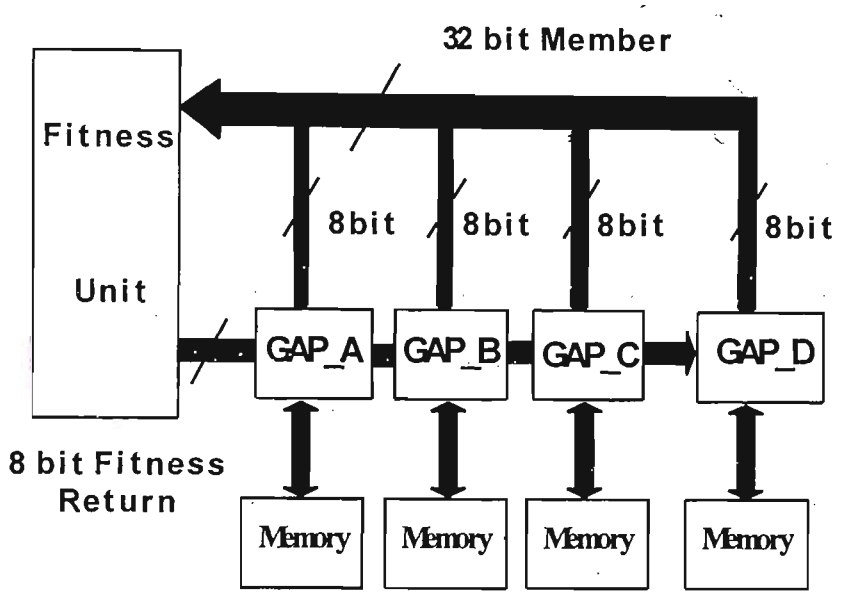


Figure 8.1: Splitting one member between four GAPs.

The FU receives the new member which is made up of the new slice from the GAP_A and the current (unchanged) slices from the other GAPs. The FU calculates the fitness value and returns the 8-bit result on the buss where it is received by GAP_A. This configuration splits the bit string and enables the GAP to handle large bit strings.

The operation of the multiple GAP architecture can be demonstrated with an example. Suppose the fitness function is $F(x,y)=x+y$. The x value is represented with 4 bits and the y value is also represented in 4 bits. Two GAPs can be used to solve this problem. One of them contains the x values (GAP_X) and the other one contains the y values (GAP_Y). The fitness value is represented in 5 bits. If there is a change to be made in x then GAP_X delivers its new string. The string for y remains unchanged at the last value

delivered by GAP_Y. Table 8.1 demonstrates how the changes in x and y values affect the fitness values.

Action	x (4 bits)	y (4 bits)	Fitness value (5 bits)	Fitness Unit busy signal '1' ≡ busy
Initial values	0	0	0	'0'
Change in x value	2	0	---	'1'
Respond to GAP_X	2	0	2	'0'
Change in y value	2	10	---	'1'
Respond to GAP_Y	2	10	12	'0'
Change in x value	7	10	---	'1'
Respond to GAP_X	7	10	17	'0'
Change in y value	7	14	---	'1'
Respond to GAP_Y	7	14	21	'0'

Table 8.1: A simple example for the multiple GAP configuration.

It is worth considering how this method relates to the current theory of genetic algorithms.

8.3 Justification for the multiple GAPs

According to the schema theory of genetic algorithms from Chapter 3, schema of above average fitness will increase in the new population according to its fitness value. If a similar GA process maintains the schema in the population, then the process should be able to increase the schema number in the new population accordingly. Then distributing a member between multiple GAPs is reasonable as long as the schemata

remain unchanged in the population. For example suppose members of 16 bits are split between two 8 bit GAPs and the following schema has a high fitness in the population:

**********11**0 = ******** (First GAP) + ***11**0 (Second GAP).

This implies that the second 8 bit schema must have a high fitness in the second GAP as well. While generating a new population, the second GAP should increase the occurrence of strings matching the schema in the next population. It is unimportant whether the first part of the member comes from the first GAP or the second GAP. Thus the multiple genetic algorithm processor should be able to increase the occurrence of strings matching the total schema according to the schema theorem.

On the other hand if the high fitness schema for the same configuration of two GAPs is:

******0101****** = ******01 (First GAP) + 01****** (Second GAP).

The active part is divided between two GAPs. Under this schema we cannot guarantee that either part of the schema (i.e. ******01 or 01******) has high fitness in the population. It is still possible to apply this configuration to see how it works, but we cannot explain the process with the GA schema theory.

It is apparent that under the schema theory, the multiple GAP configuration is not expected to work in all cases. Let's assume we know nothing about the problem and choose an unsuitable representing schema for the member strings. If we then arbitrarily partition the members across several GAPs, there is a good chance that correlations will occur between the different sections and this will adversely affect the optimising capability of the system. The test examples in this chapter avoid this problem mainly by using some knowledge of the environment to partition the member strings over the multiple GAPs.

8.4 Simulation of the multiple GAPs

To simulate the operation of multiple GAPs we need a concurrent programming environment. Simulation programs can be handwritten in a concurrent programming language like Parallel Pascal or Parallel C to compare how the single and multiple GAPs solve problems. Fortunately the VHDL simulation tools can handle concurrent processes and it is possible to directly test models of the multiple GAPs. The three examples in Chapter 7 were simulated using multiple configurations and the results are compared with the single configuration.

8.4.1 PID controller

Figures 8.2 to 8.5 show the result of simulations for configurations of two, three, four and six GAPs averaged over ten individual runs for the PID controller system. The same parameters in Section 7.1.2 are used for all GAPs during simulations. In Figure 8.2 the normalised error value is shown after each generation. All three configurations deliver 24 bits to the objective function. Table 8.2 shows the best final values for the multiple configurations. Note that the final error values are close, but the K values are very different. This suggests that the problem surface has many local optimum points and each configuration merges to one of them.

Best Values	Single GAP	Two GAPs	Three GAPs	Four GAPs	Six GAPs
Normalised error value	7e-2	5.8e-2	4.8e-2	5.7e-2	6e-2
Ki	5	3	13	3	-40
Kd	63	42	105	45	45
Kp	95	83	185	95	103

Table 8.2: The best final values for the five configurations.

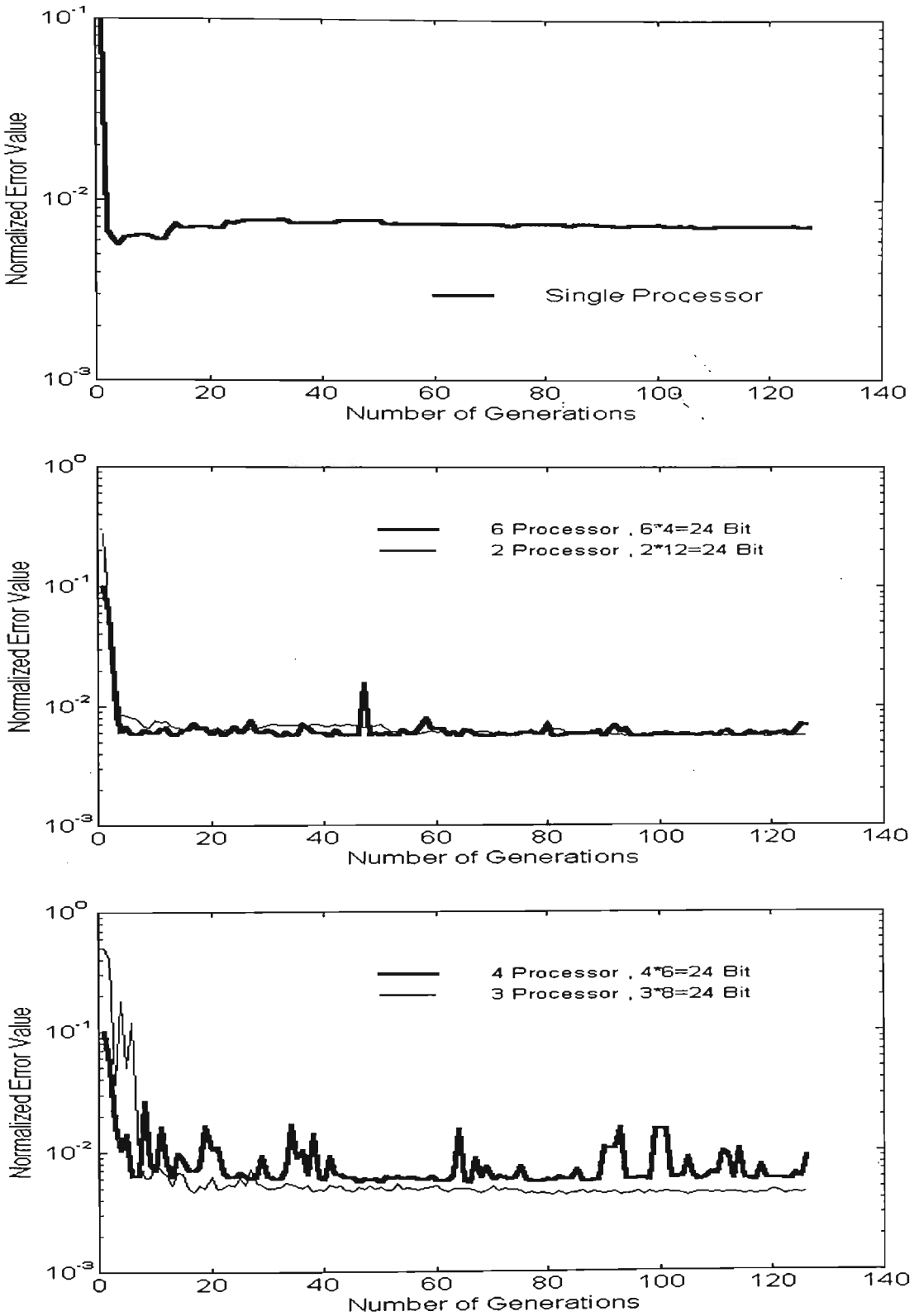


Figure 8.2: The results of the PID controller simulation with single, 2, 3, 4 and 6 processors (Error Value).

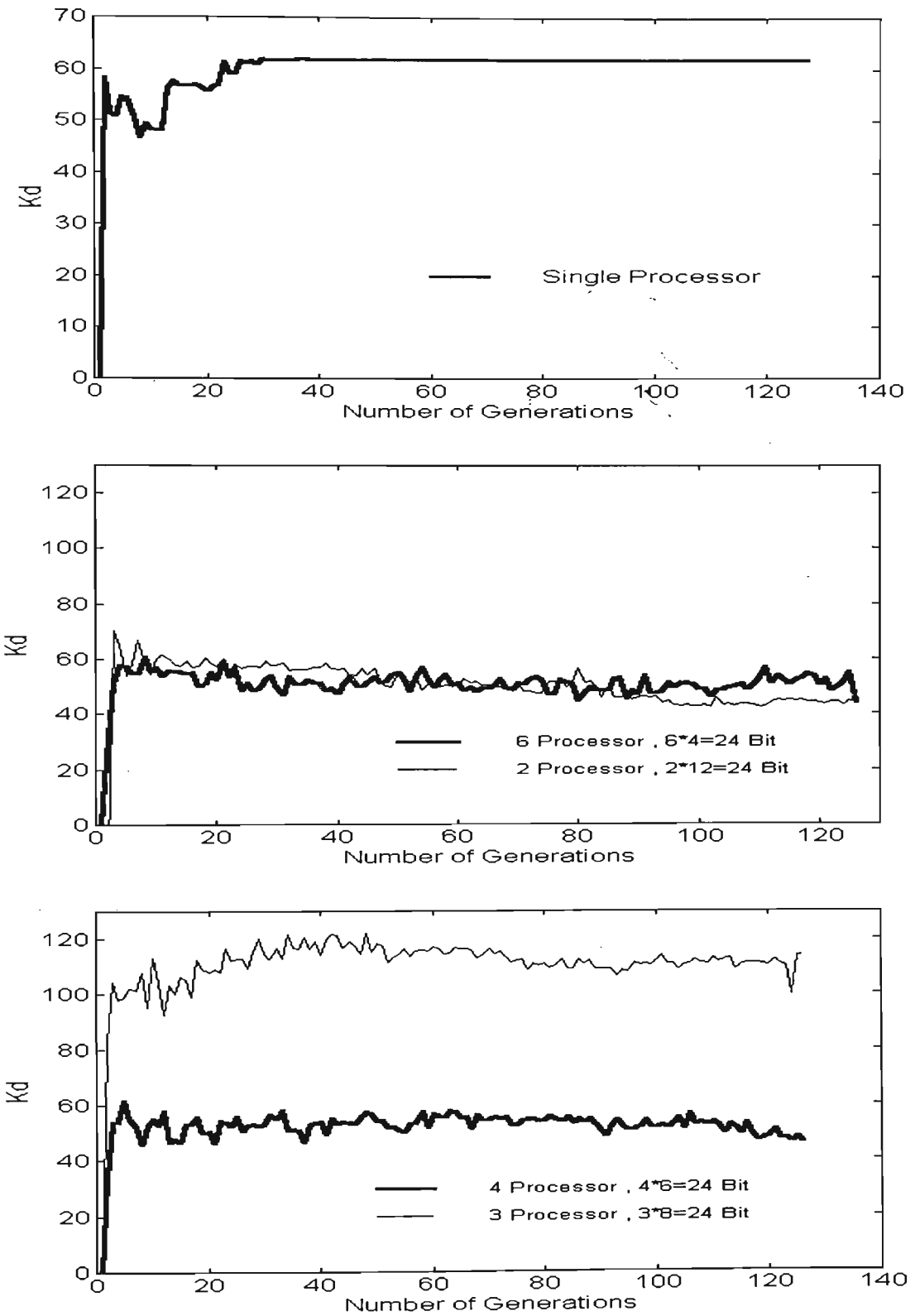


Figure 8.3: The results of the PID controller simulation with single, 2, 3, 4 and 6 processors (K_d Value).

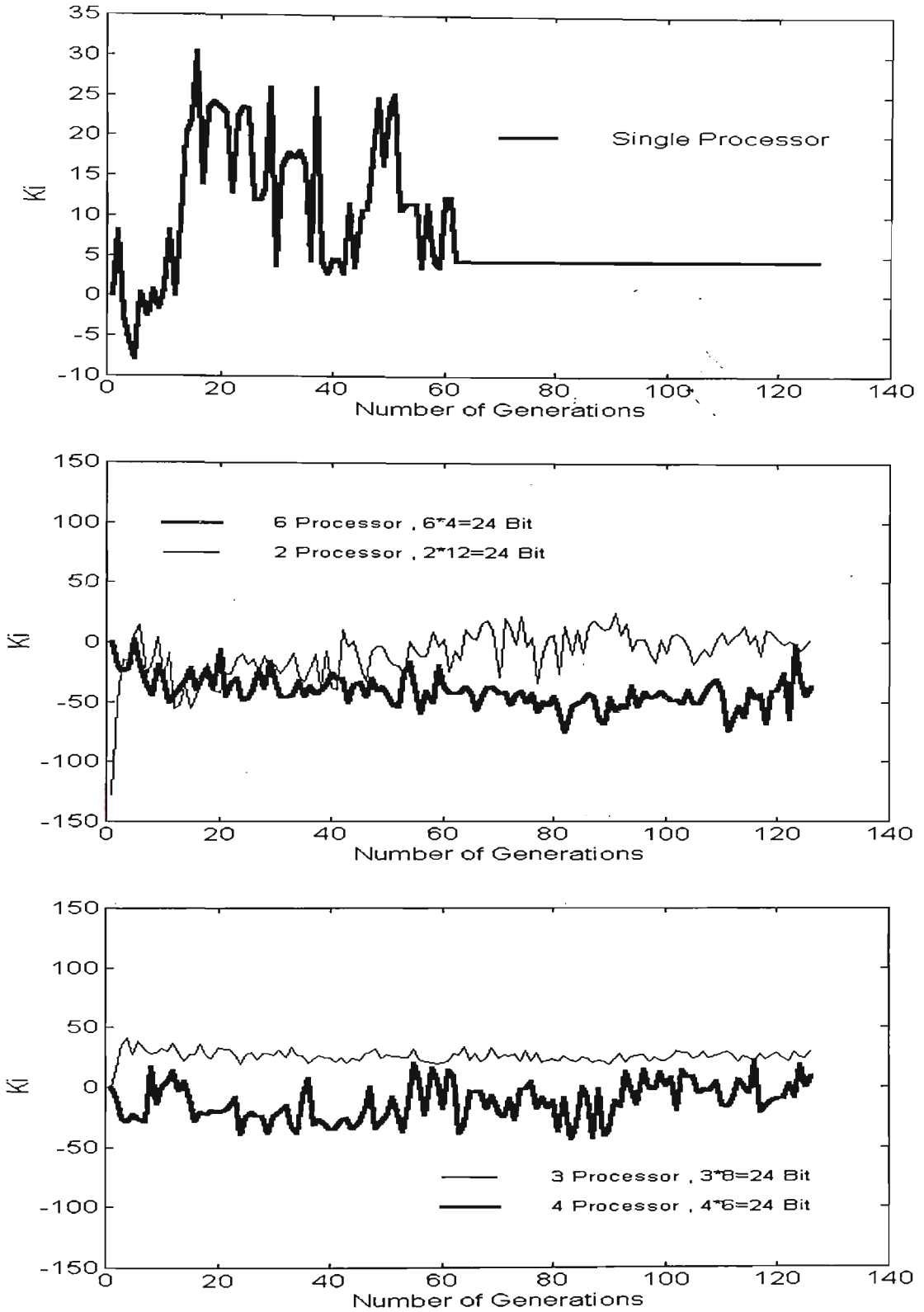


Figure 8.4: The results of the PID controller simulation with single, 2, 3, 4 and 6 processors (K_i Value).

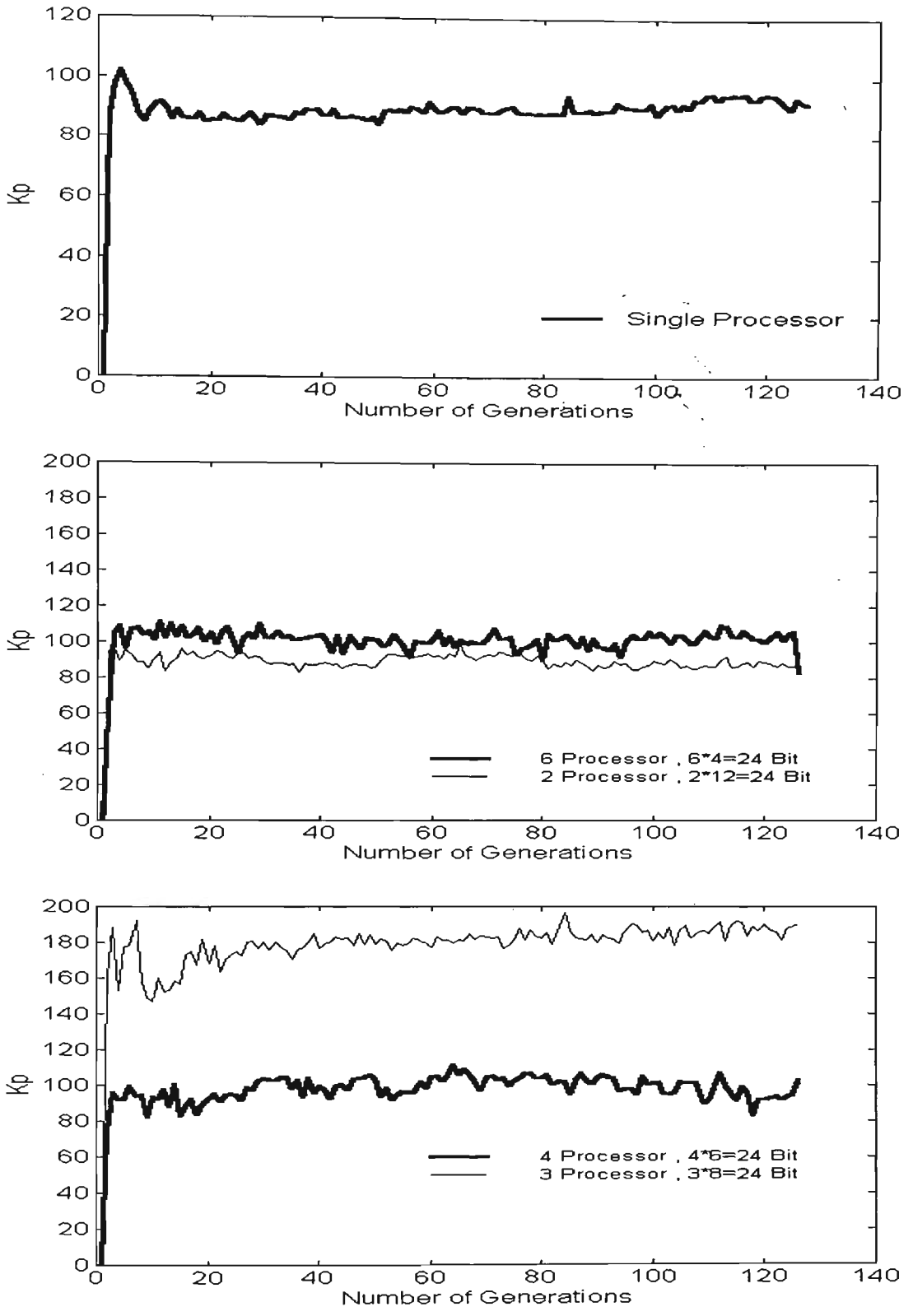


Figure 8.5: The results of the PID controller simulation with single, 2, 3, 4 and 6 processors (K_p Value).

The normalised error value column in Table 8.2 shows that all multiple processor configurations perform better than the single processor. This is because they work on a smaller search space with the same resolution for the objective function. On the other hand single processor configurations give a smoother curve than the multiple configurations. The graphs show that the 2 and 6 GAP configurations converge faster than the 3 and 4 GAP configurations and the curves are smoother but the error result in the final generation is higher. In the 3 GAP configuration there are no correlations between bit strings as each GAP is optimising one K value. This may be the reason why the 3 GAP configuration achieves the best error result.

8.4.2 Economic power dispatch problem

Figures 8.6 to 8.8 show the result of simulations for configurations of one, four and eight GAPs averaged over ten individual runs for the EPD problem. The same GAP parameters as in Section 7.2.2 are used for all GAPs during simulations. In the figures the minimum and maximum costs are shown after each generation. All three configurations deliver 32 bits to the objective function. Table 8.3 shows the best results and the average result for three configurations after the final generation.

Costs	One GAP 32 bit	4 GAP 4*8=32	8 GAP 8*4=32
The best minimum ever	8212.30	8211.47	8211.56
Average	8343	8243	8242

Table 8.3: The best ever minimum costs and the average minimum costs.

Figures 8.6 to 8.8 and Table 8.3 show that the 4 and 8 processor configurations both outperform the single processor. This is mainly because they are working on a smaller search space with the same resolution for the objective function. The only major

difference between the 4 and 8 processor configurations is the maximum cost curve which for 4 processors, is lower and smoother than for 8 processors.

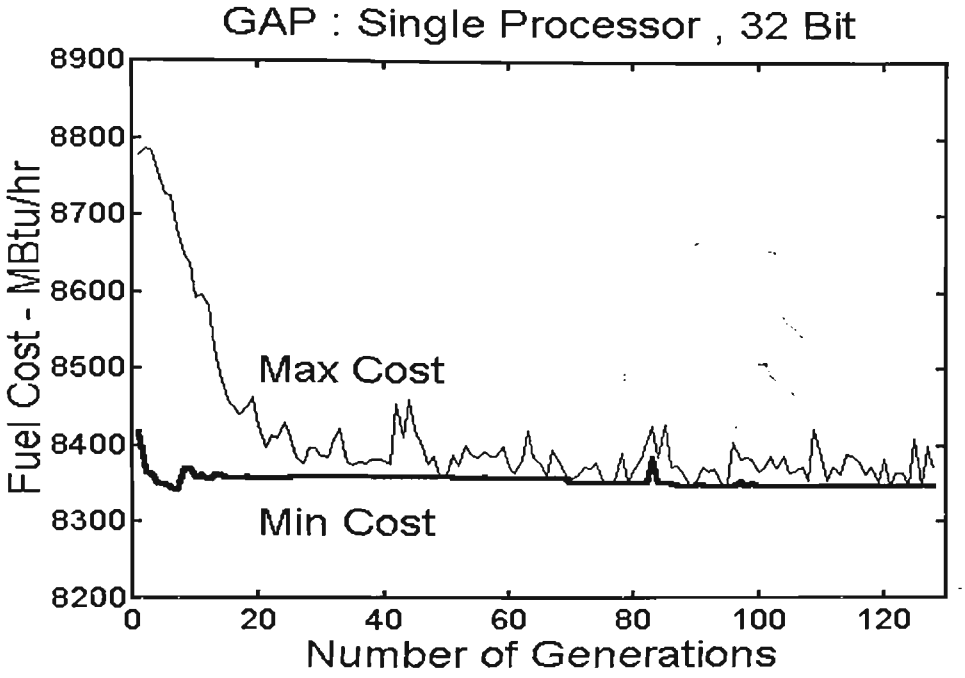


Figure 8.6: Maximum cost and minimum cost versus number of generations for the single processor.

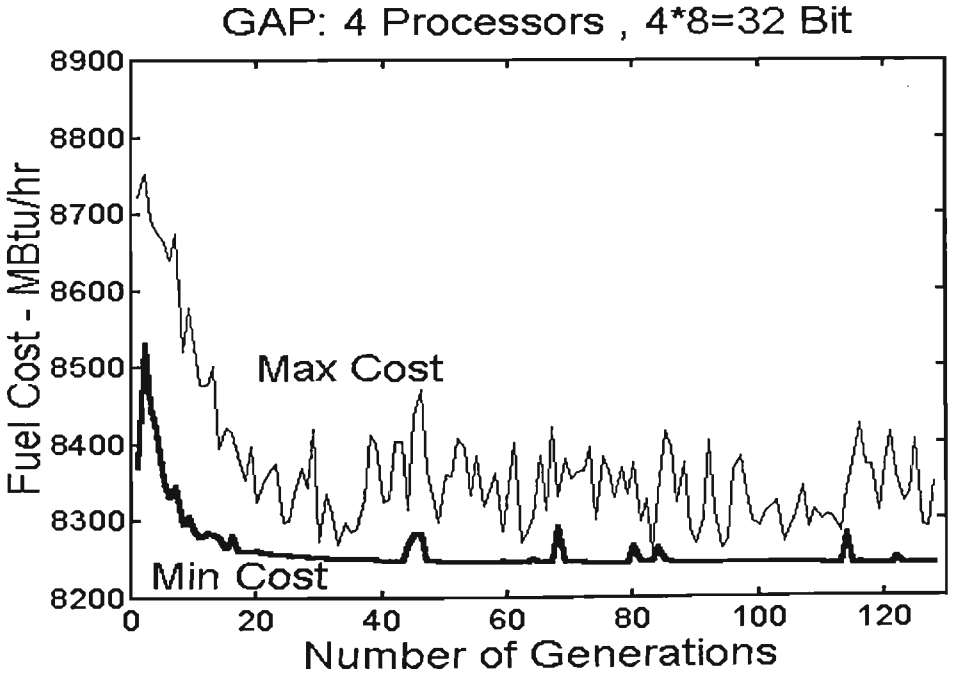


Figure 8.7: Maximum cost and minimum cost versus number of generations for the 4 processors.

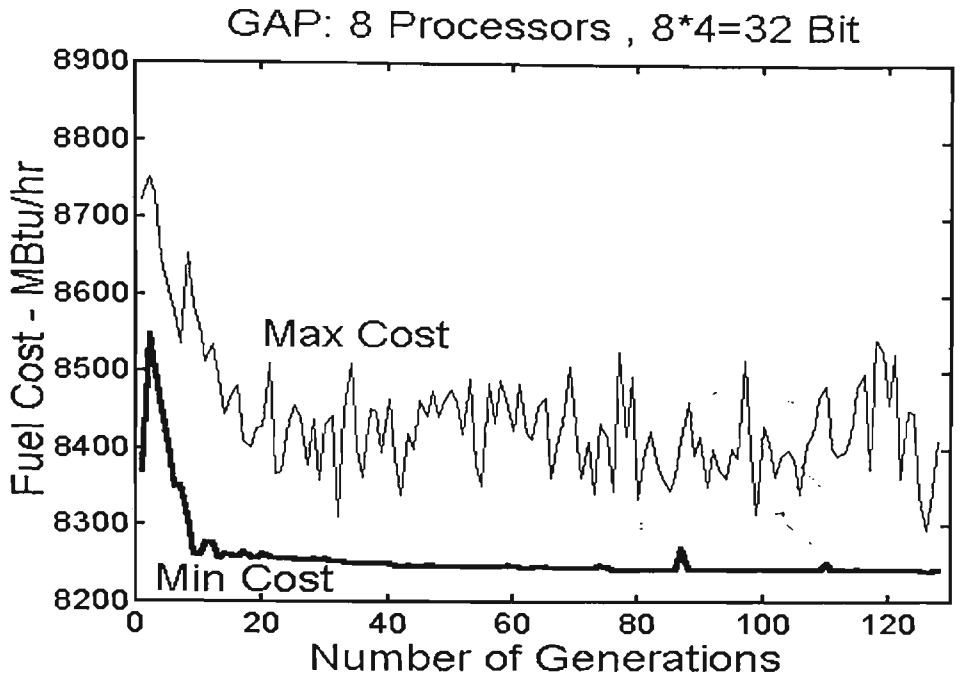


Figure 8.8: Maximum cost and minimum cost versus number of generations for the 8 processors.

8.4.3 Adaptive IIR filters

The multiple GAP configuration has been simulated on the adaptive IIR filter problem of Section 7.3. Figures 8.9 to 8.11 show the result of simulations for configurations of one, two and four GAPs averaged over ten individual runs. The GA parameters are the same as in Section 7.3.2 except that the member size is 32 bits. The graphs show the best values from the population for the 'a' and 'b' parameters of (7.17) and the MSE after each generation. We have shown only 50 generations because after that point there are no changes in the values. Table 8.4 shows the best final values from each configuration.

Best values	Single Processor	Two Processors	Four Processors
MSE	0.38	0.46	0.42
'a' value	-0.57	-0.54	-0.58
'b' value	0.81	0.75	0.74

Table 8.4: The final results of each configuration.

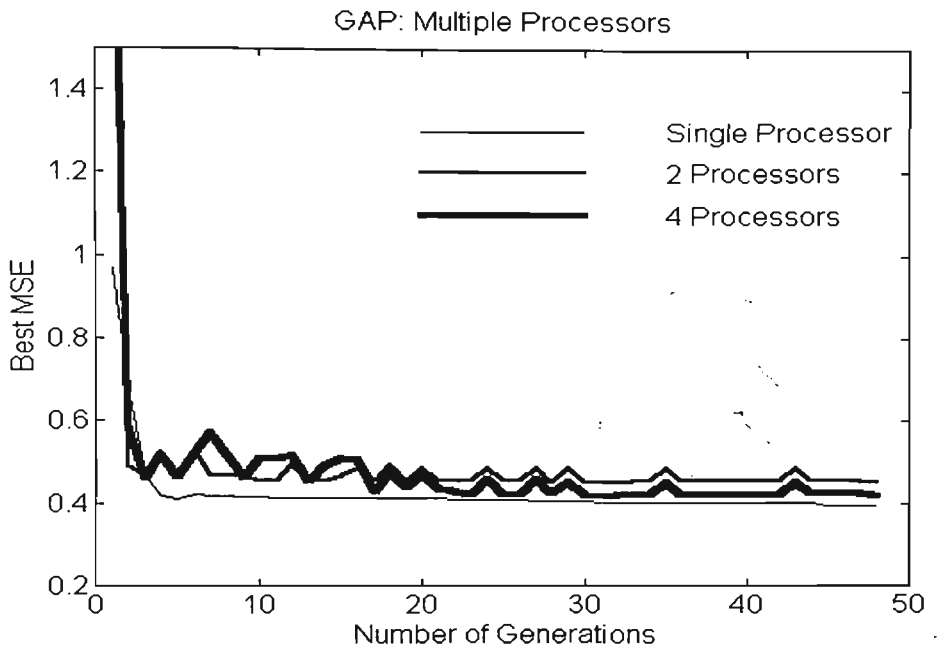


Figure 8.9: The best adaptive IIR characteristics (Mean Square Error (MSE)) versus number of generations with multiple processors (single, 2 and 4 processors)

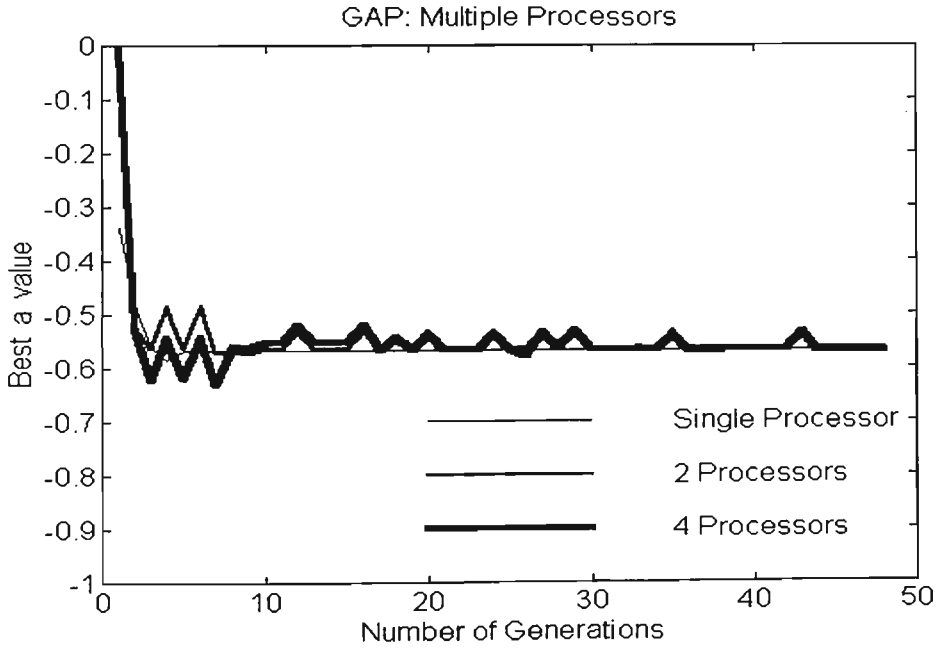


Figure 8.10: The best adaptive IIR characteristics ('a' value) versus number of generations with multiple processors (single, 2 and 4 processors).

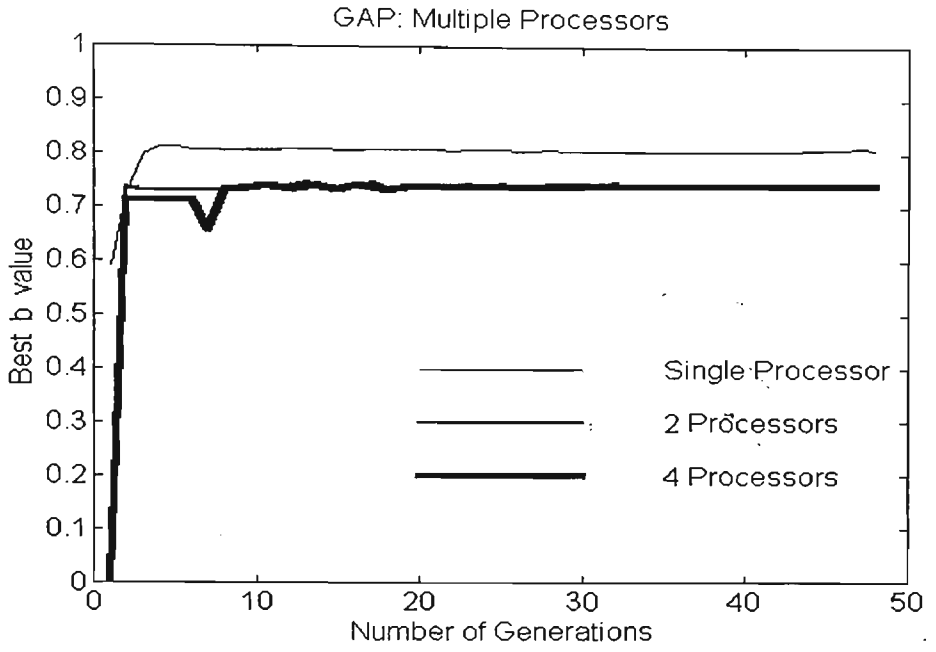


Figure 8.11: The best adaptive IIR characteristics ('b' value) versus number of generations with multiple processors (single, 2 and 4 processors)

The figures (8.9 to 8.11) and Table 8.4 show that the 2 and 4 processor configurations both minimise the MSE in a reasonable number of generations. Indeed the 2 processor configuration settles faster than the single processor but the final error value (MSE) and oscillation is higher than for the single processor. The 4 processor configuration takes longer to settle than the single configuration but the final error value is less than for the 2 processor configuration. The oscillation in the values for 4 processors is greater than for the other two configurations. It appears that the multiple configurations are capable of finding stable values, but the accuracy is not as good as for the single processor.

8.5 Conclusions

This research shows that the multiple GAP configurations have potential to be used in real applications. The multiple GAP has been applied to the three applications in Chapter 7 and all simulations show this configuration is capable of competing with the

single GAP. In two of the applications the multiple GAP appeared to optimise the application better than the single GAP but it has not been proven that the multiple GAP is always better than the single GAP. For example in the filter application, performance of the multiple processors appears to be worse than for the single GAP. To investigate the operation of the multiple configurations further mathematical analysis is required but this is outside the scope of this thesis

There are two issues concerning multiple GAP configurations. The first one is the high oscillation observed during search and the second one is selection of the optimum number of GAPs for a particular problem. The oscillation problem may be advantageous to the operation of the multiple GAP. Oscillation suggest a wider search in the problem space and thus a higher chance of finding the global optimum. The second problem, the number of GAPs required to solve a particular problem, depends on the coding of the problem and this will affect the performance of the search. In selecting a suitable configuration of GAP's it is desirable to utilise some problem-specific knowledge. A good place to start is with a model which can be used to approximate a solution. If the GAP is operating in a parameter tuning role then it is appropriate to select the number of GAPs to be equal to the number of model parameters to be tuned. If this fails then the model may be inappropriate.

Chapter 9

Adaptive behaviour of the GAP

In the previous chapters the performance of the GAP has been examined in a static environment in which the Fitness Unit responses for the same inputs are similar. This chapter considers how a GAP might be applied to a dynamic system. For static fitness functions all members of the population eventually converge to the optimum point at the end of a successful search cycle. If the optimum point then moves as a function of time, the GAP cannot respond efficiently if there is insufficient genetic diversity left in the population. We will apply our GAP model to the same applications as in Chapter 7, but this time the environment (and thus the fitness evaluation) is to be varied in time. The objective is to test the performance of the GAP model in dynamic situations.

9.1 Adaptive behaviour

A system is adaptive if during changes in the external environment, at least one internal control variable in the system changes to produce better behaviour. If a system is adaptive then any larger system based on that system can be adaptive. Figure 9.1 shows how an adaptive GAP can be applied to a system and produce adaptive behaviour.

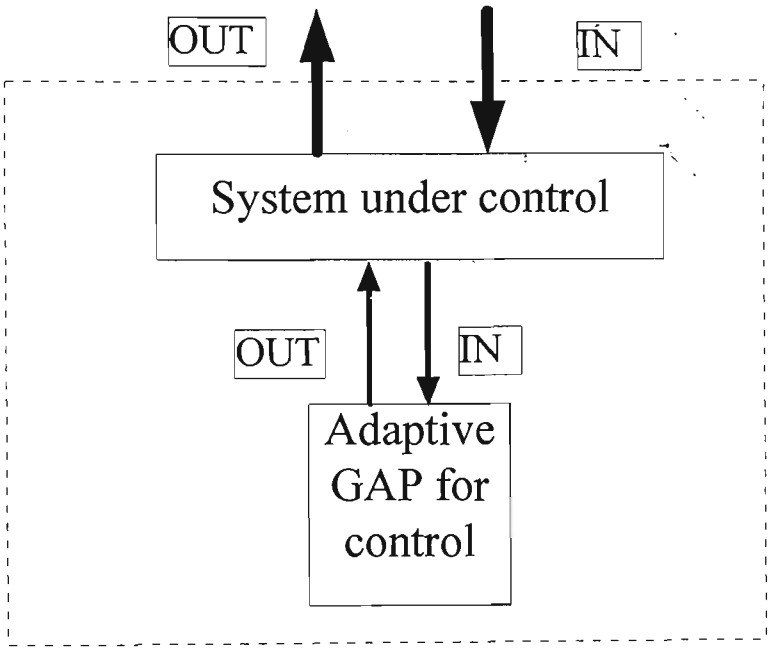


Figure 9.1: An adaptive system based on the adaptive GAP.

Suppose the system being made adaptive is a PID controller and a GAP is used to optimise the K parameters of (7.5). Whenever changes due to external causes occur in the plant, these changes affect the output of the system and thus eventually affect the response to the GAP. If the GAP is adaptive, then it will respond to these changes and vary the K values. The new K values change the behaviour of the plant and finally change the output of the system. If the process works then the whole system has become adaptive.

There are two ways in which the behaviour of a system might change. The first is a slight change in the operating point due to changes in environmental factors such as

temperature or pressure. The second is a sudden change in a key parameter that affects the whole operation of the system.

The GAP has difficulty in keeping track of small changes in the environment. If it is necessary to use the GAP in such situations then it should be periodically restarted with a population of random members. Recognising that the environment has drifted is difficult and restarting from an initialised population slows down the optimisation.

As an example to show how the GAP operates in a slowly varying environment, it has been simulated on the Economic Power Dispatch problem when the demand power is varied between 800 and 850 according to the Figure 9.2.

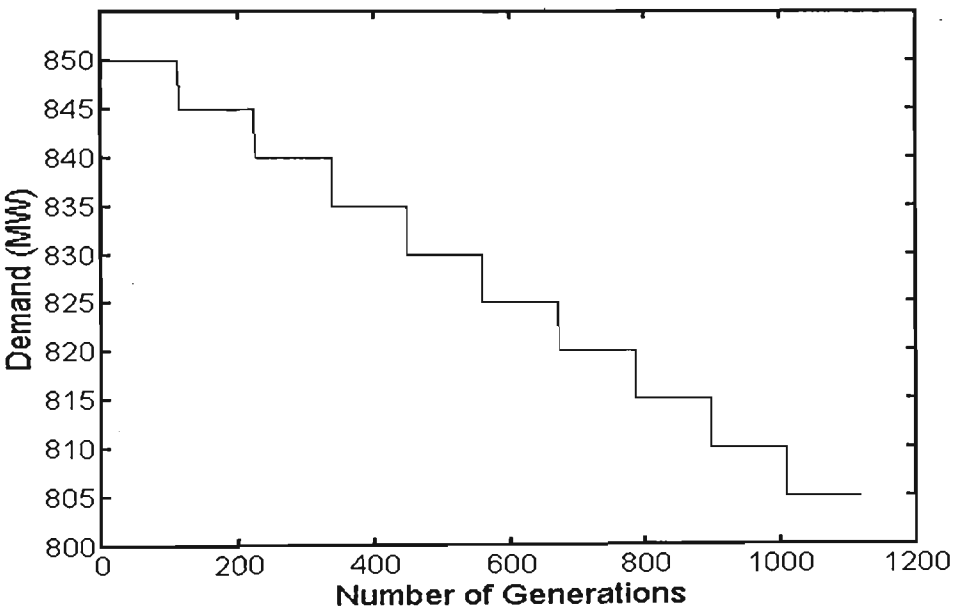


Figure 9.2: The changes in the demand power.

Figure 9.3 shows the results of simulation for the variable demand. It can be seen that because the demand only changes 5 MW (0.5%) in each step, then the GAP has difficulty in adjusting itself to this situation and often returns the same cost after a change.

The next section considers the performance of the GAP with larger changes in the environment.

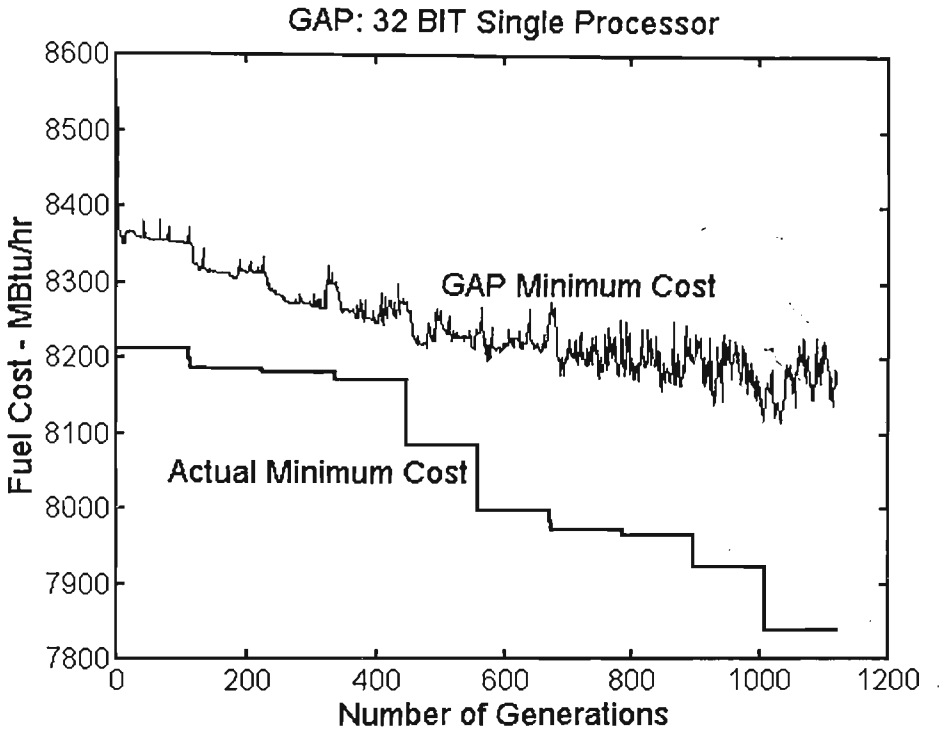


Figure 9.3: The results of simulations when the demand power is varied in small steps.

9.2 Adaptive GAP

The operation of the adaptive GAP can be explained with an example. Consider the optimisation task shown in Figure 9.4. The load RL changes in time due to unknown external factors. The Optimisation Part (OP) is required to adjust a resistor RS to equalize the voltage drop on RL and RS. In theory this task is achieved by measuring RL or the current (I) through the resistors and then adjusting RS. For these systems, an adaptive mechanism is needed which changes its internal parameters to match with the system. This example demonstrates the need for a different approach to optimisation. We can use a genetic algorithm for the required adaptive mechanism without the need for embedding any knowledge of the current/resistance relationship inside the GA.

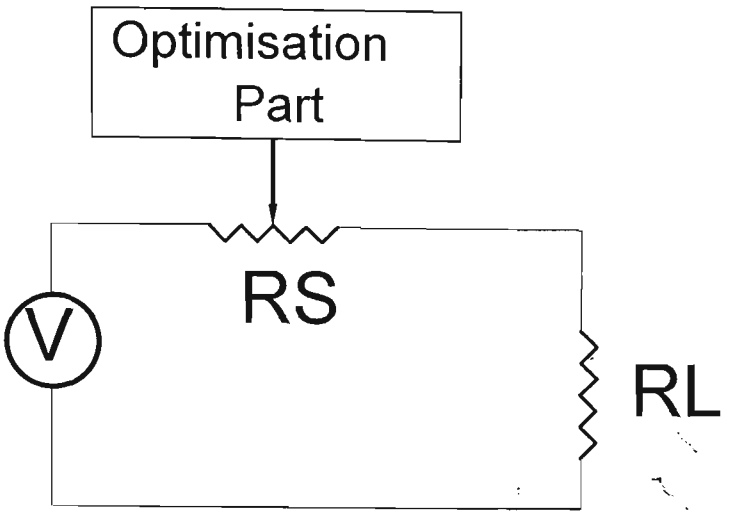


Figure 9.4: A simple optimisation task.

One way to make the GAP adaptive is to increase the mutation rate thus adding some randomness to the population. So when the GAP detects a change in the fitness values (either by a signal from the Fitness Unit or by examining the fitness values in population) it will change the probability of mutation to 100% for one generation and then return the mutation rate to its normal value. In this case the population can be spread out from the current optimum point. There are other ways of making the GAP adaptive such as starting with a totally random population or changing the coding of the members, but these are complicated and time consuming. Changing the probability of mutation is simple and it is easy to change the duration of altered mutation for different applications.

The only problem is informing the GAP that a sudden change has occurred in the fitness evaluation unit. This can be done by arranging for the GAP to respond to a signal from an external detector which is activated by a sudden change in the load (RL). The second method involves examining the fitness values of the population to detect the change internally. Whenever a change occurs in the load the fitness values are expected to drop suddenly to very low values and this could be detected by an algorithm in the GAP. It can be expected that such an algorithm would sometimes falsely report changes and thus

decrease the efficiency of the GAP. In all simulations in this chapter the first method is used because it does not contribute any additional error to the tests. For the tests, the fitness function was computed from the output power function and can be expressed as:

$$F(R_l, R_s) = R_l * R_s * V^2 / (R_l + R_s)^2 \tag{9.1}$$

For the example, the value of V is equal to 32 volts. Figure 9.5 shows the surface of this function. The optimum solution is achieved by maintaining RL=RS for RL changing randomly in time.

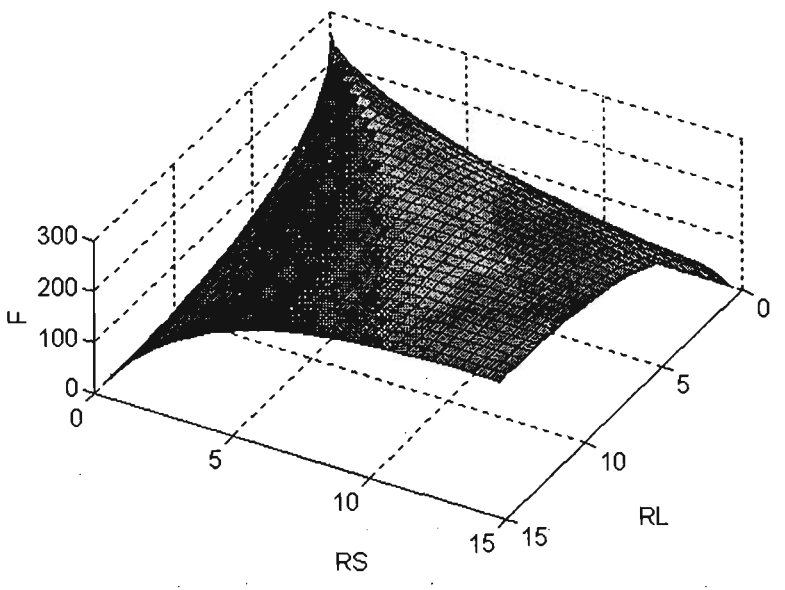


Figure 9.5: Problem space of a dynamic fitness function.

In the genetic algorithm processor, the following parameters are used:

mutation rate	= 1%
crossover rate	= 90%
population size	= 32
generations	= 128
member size	= 16 bits
fitness value	= 8 bits.

The simulations were conducted for 75 random changes in RL and the GAP optimised after every change. In the first test a one point crossover algorithm was used. In the second one uniform crossover was used and finally in the third test, the coding routine was changed to Gray coding [Davis, 1991] with uniform crossover. The results of the tests averaged over ten runs are shown in Table 9.1. In this table the column marked "Search Failed" designates cases where the GAP failed to find a solution in which RL=RS.

Test Number	Algorithm	Number of Changes in RL	Search Failed	Error Percentage
1	One Point Crossover	75	15.4	20
2	Uniform Crossover	75	5.2	7
3	Gray Code	75	4.5	6

Table 9.1: The results of the dynamic fitness function simulations.

The table shows that in tests 2 and 3 the genetic algorithm hardware can find the solution with a low percentage of error. It also shows that uniform crossover is better than one point crossover and that using Gray coding of member strings increases the efficiency of the GAP in this example.

9.3 Adaptive performance of the GAP in engineering applications

In this section the GAP model is applied to the applications in Chapter 7. In all examples changes in the environment are represented by changing one of the features of the fitness computation with time. In the PID controller the plant transfer function is varied. The demand power is varied in Economic Power Dispatch problem and in the filter problem one of the filter parameters is varied in time.

9.3.1 PID controller

Genetic Algorithm Processor simulations have been conducted for the PID controller system in Figure 7.1. The reference signal (R(t)) is a step signal as shown in Figure 7.2. For the purpose of manipulating the environment and evaluating the fitness function we provide the following transfer function for the plant:

$$P(S) = \frac{a}{S(S-1)(S+5)} \tag{9.2}$$

To test the adaptive behaviour, the 'a' parameter of (9.2) is varied during the simulations in a series of steps as shown in Figure 9.6. In this figure each step occurs after 100 generations and increases 'a' by one unit. This gives time for the GAP to settle after each change. At each step a signal from the Fitness Unit informs the GAP that a change has occurred in the fitness evaluation and the GAP alters the mutation rate to 100% for one generation. This avoids the necessity for the GAP to recreate a random population after each change. A model of the PID controller is provided (as in Chapter 7) in the form of the following transfer function:

$$U(S) = \left(K_i / S + K_p + K_d * S\right) E(S) \tag{9.3}$$

To handle this problem we used a multiple GAP configuration as depicted in Figure 9.7. Each GAP optimises one K value from the PID controller. The following parameters are selected for all GAPs:

generations	= 3000
fitness value	= 8 bits
member size	= 24 bits
population size	= 64
crossover rate	= 90%
mutation rate	= 2%.

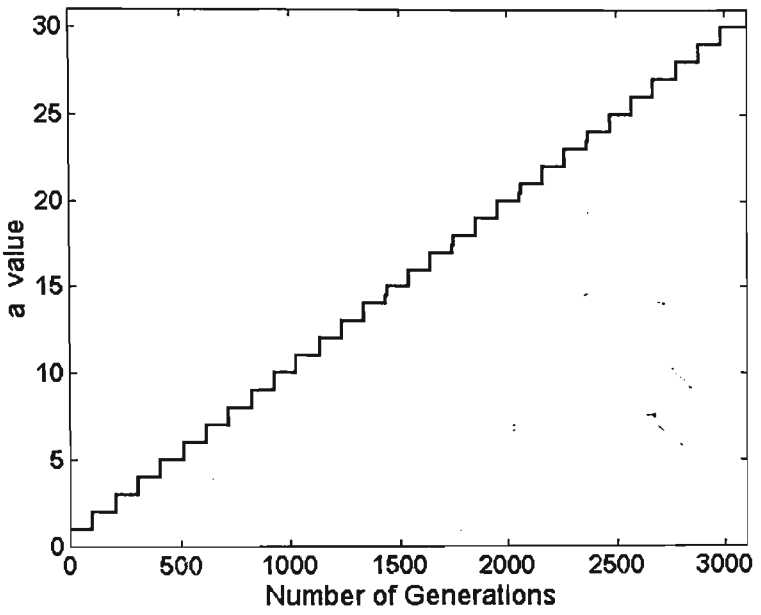


Figure 9.6: The 'a' value changes with the number of generations.

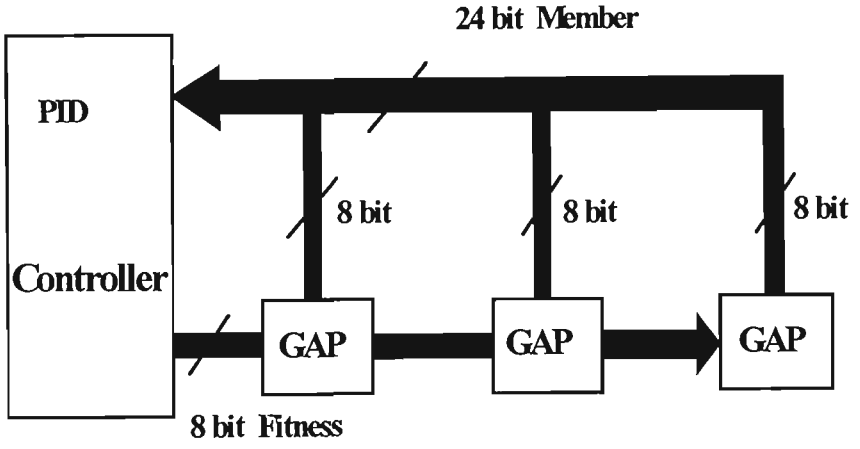


Figure 9.7: The architecture of the adaptive Multiple GAP used for PID controller system.

The fitness function is defined according to Chapter 7 as:

$$Fitness = \sum_{k=1}^{2000} \left((kT) * |e(k)| \right) \tag{9.4}$$

Figure 9.8 and 9.9 show the results of simulations averaged over ten individual runs. In these figures the best of the three K values and the fitness values are shown after each generation. Figure 9.10 shows the best response signal (Y(t)) for a=1 and a=30. The characteristics of the response signal are shown in Table 9.2.

In the Figure 9.8 and 9.9, K_p changes from about 200 for a=1 to 30 for a=30. In the same time K_d changes from 130 to 20 and K_i changes from 35 to 5. This shows that when the 'a' value is altered in the system, K_i and K_p are affected most as the PID controller adapts to the plant. The fitness value graph (Figure 9.8) shows that the GAP always keeps the response signal within a small error margin during changes in the 'a' value. It shows that the PID controller always settles to give a good response.

Parameters	a=1	a=30
K_p	206	23
K_i	131	14
K_d	38	15
Steady State Error	0.000	0.000
Overshoot	0.7676%	1.122%
Rise Time(S)	1.150	1.495
Settling Time(S)	1.575	1.600

Table 9.2: Characteristics of the best member for a=1 and a=30.

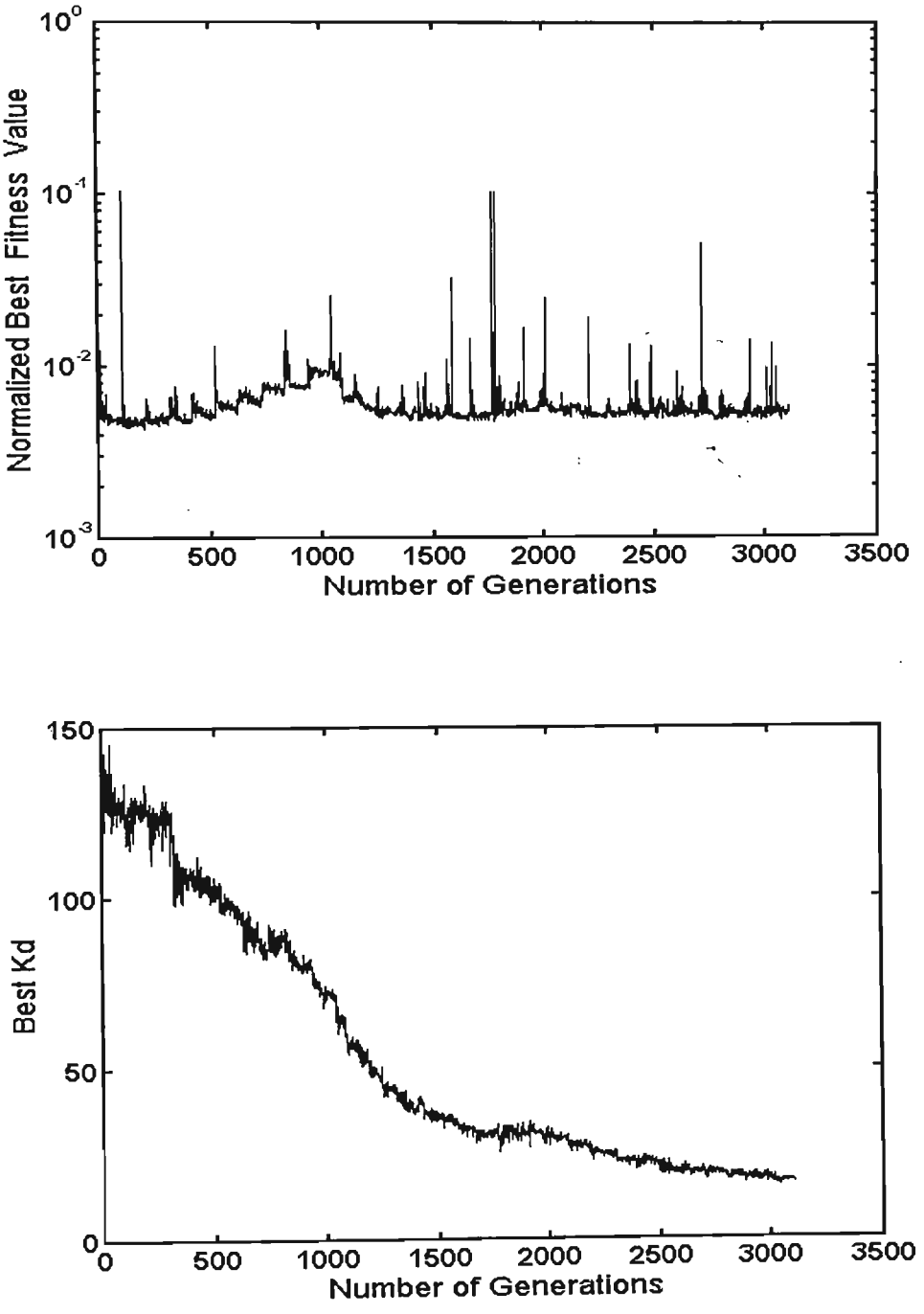


Figure 9.8: The results of the PID controller simulations (normalised error value and K_d).

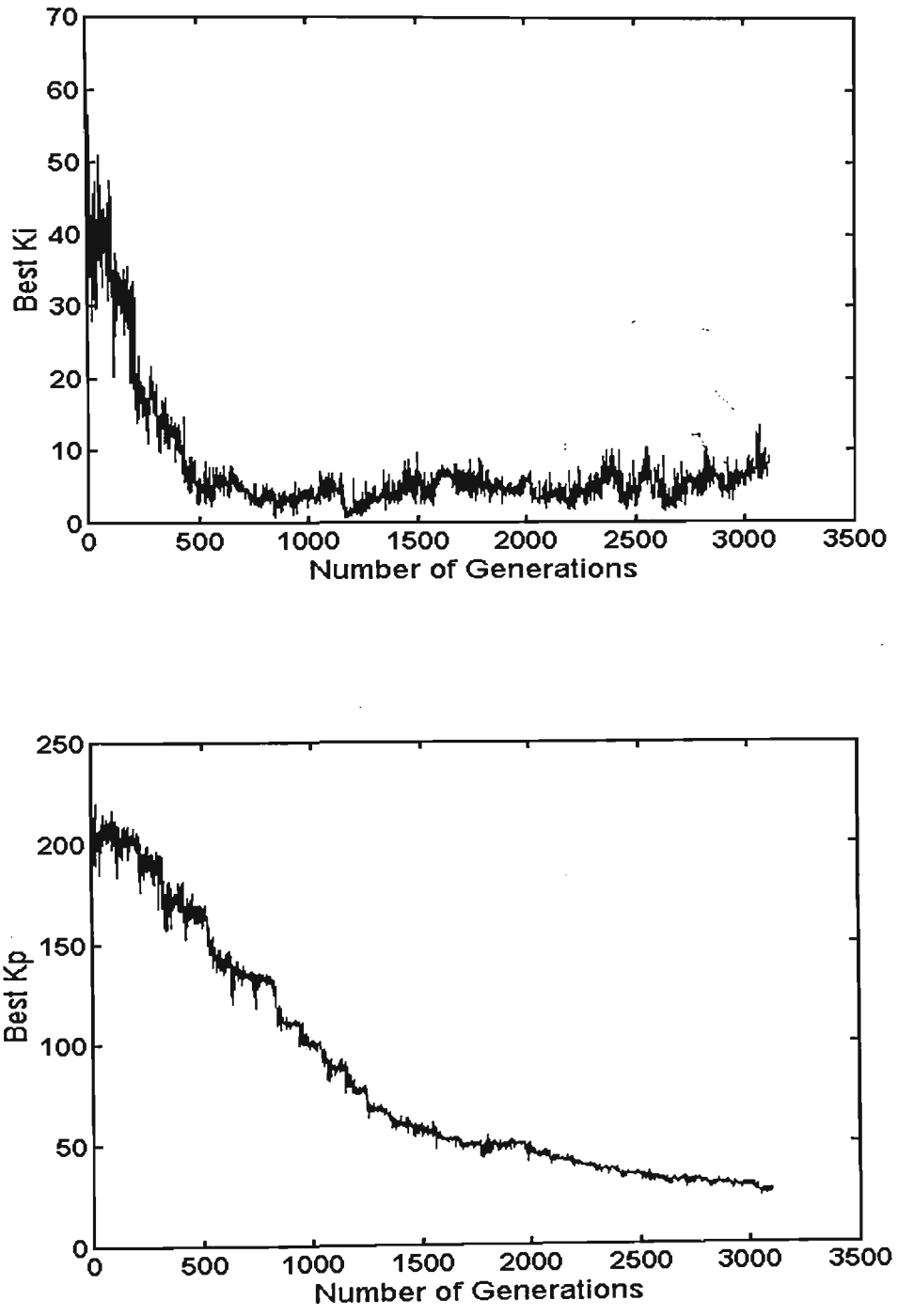


Figure 9.9: The results of the PID controller simulations (K_i and K_p).

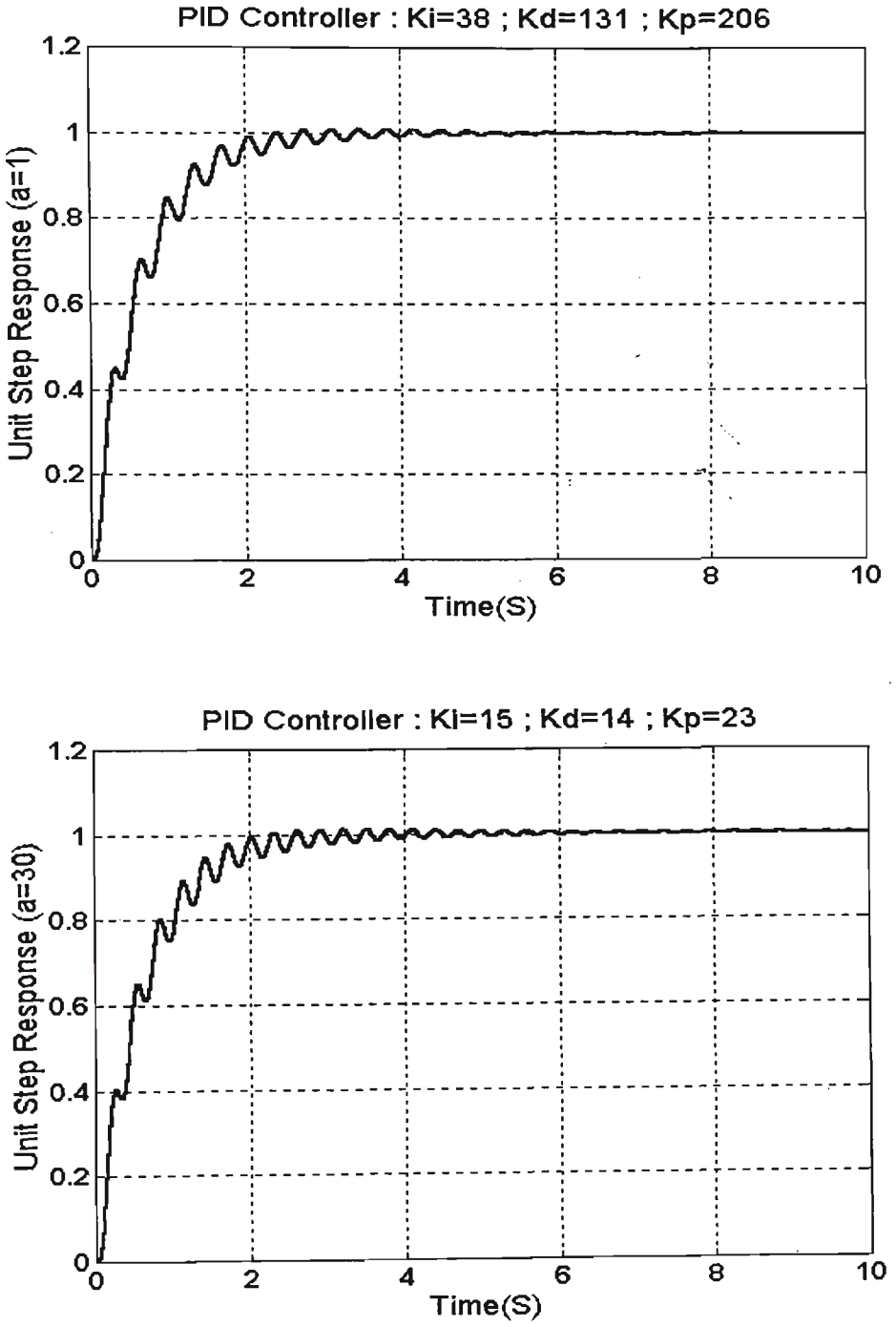


Figure 9.10: The unit step response of the PID controller system for a=1 and a=30.

9.3.2 Economic power dispatch problem

Genetic Algorithm Processor simulations have been conducted for the optimal economic dispatch of the 3 generator power system described in Chapter 7. These simulations were conducted with the multiple GAP configuration (Figure 9.11) with the following parameters:

generations	= 2048
population size	= 32
crossover rate	= 90%
mutation rate	= 2%
fitness value	= 8 bits
member size	= 32 bits.

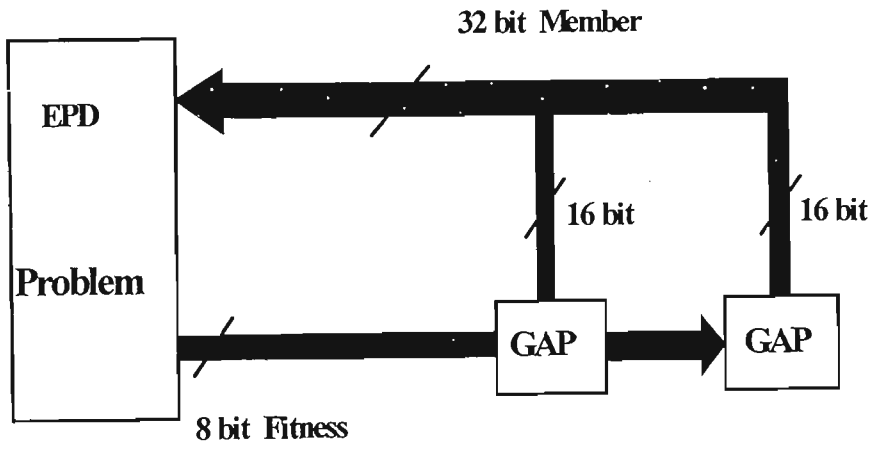


Figure 9.11: The architecture of the adaptive multiple GAP used for EPD problem.

The constraints and coefficients are given in Section 7.2.2. The demand power was varied in steps according to Figure 9.12 and the loss power is ignored.

Figure 9.13 shows the result of the simulations averaged over ten individual runs. The minimum operational costs are shown after each generation.

It can be concluded that the GAP is capable of adapting to sudden changes in the demand power. In all cases the GAP error in finding the best results is in the range 2% to 5%.

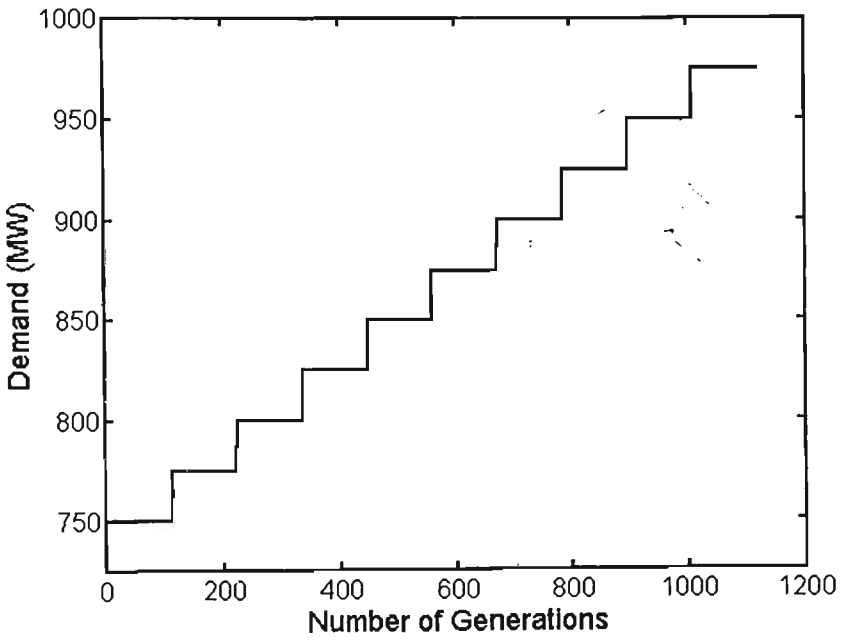


Figure 9.12: The demand power is varied with the number of generations.

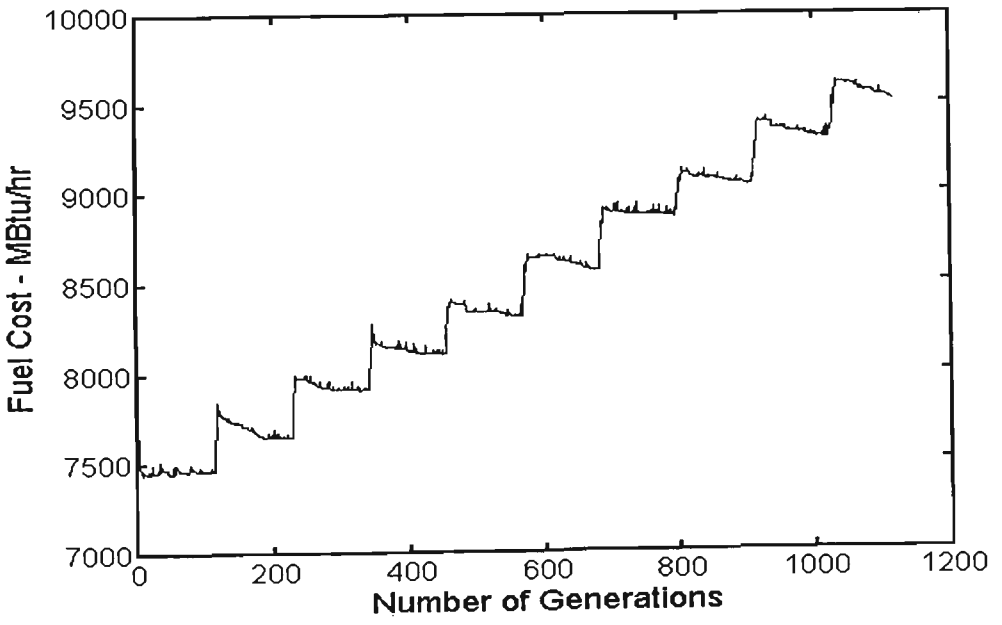


Figure 9.13: Minimum costs versus number of generations when the demand is varied as in Figure 9.12.

9.3.3 Adaptive IIR filters

In this experiment the zero location of the IIR filter from Section 7.3.2 [Johnson and Larimore, 1977] was modified during simulations. The altered transfer function of the desired system $H_D(z)$ (see Equation 7.16) is given as:

$$H_D\left(z^{-1}\right) = \frac{0.05 - \beta z^{-1}}{1 - 1.1314 z^{-1} + 0.25 z^{-2}} \tag{9.5}$$

where the value of β (which controls the zero location) has been artificially stepped up from 0.2 to 1.0 in increments of 0.2 after every 70 generations (Figure 9.14).

The configuration of the multiple GAP for this example is shown in Figure 9.15.

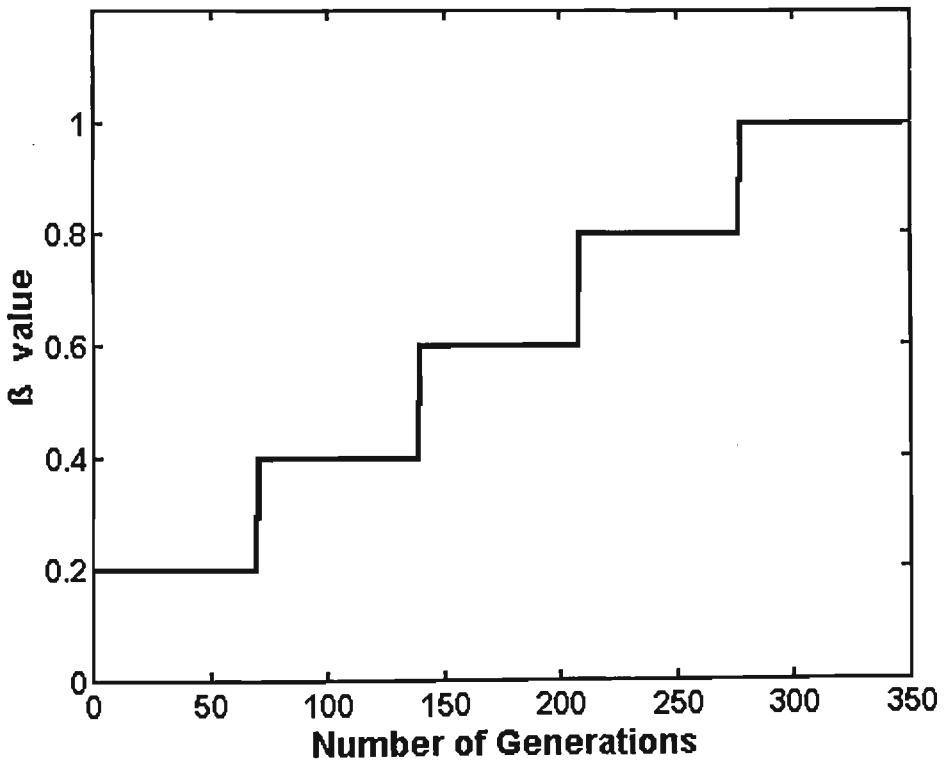


Figure 9.14: The 'β' value changes with the number of generations.

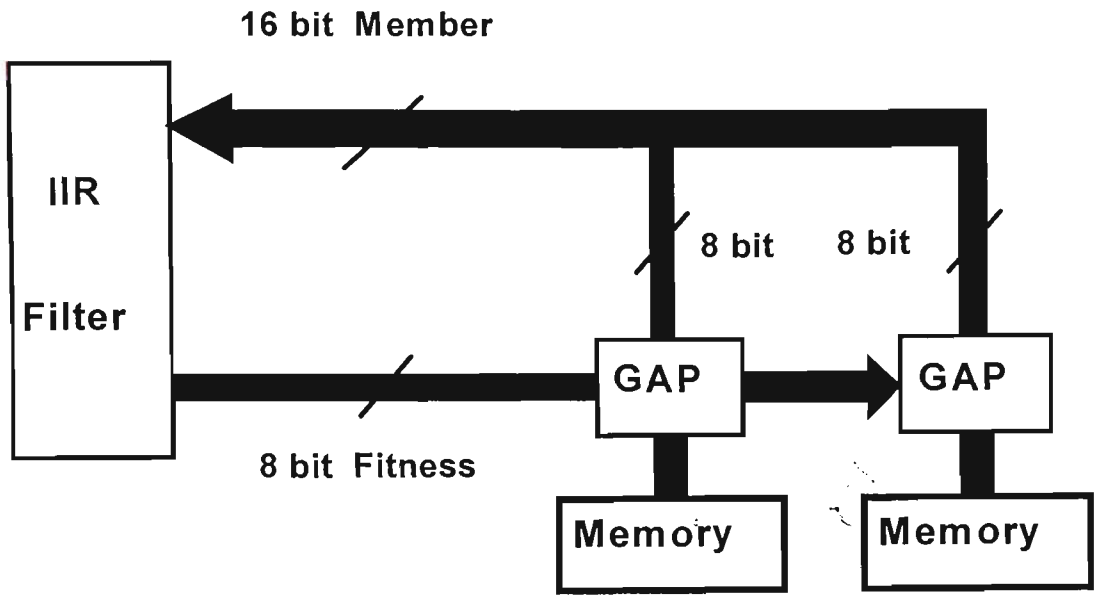


Figure 9.15: The architecture of the adaptive multiple GAP for the IIR filter.

The first GAP optimises the 'a' value and the second one optimises the 'b' value of (7.17) in Chapter 7. The following parameters are selected for all GAPs:

generations	= 350
population size	= 64
crossover rate	= 90%
mutation rate	= 2%
fitness value	= 8 bits
member size	= 16 bits.

Note that the number of generations is increased to 350 for this test.

The results for the simulations are shown in Figures 9.16 to 9.18. For every change in β, there is an immediate change in the 'a', 'b' and the MSE values. After about 60 generations, the MSE stabilises to a value of less than 0.6. Note that there is a limit to the rate at which β can be altered in this manner. This depends on the GAP processing rate and the fitness evaluation rate as the GAP needs to evaluate 30-60 generations to settle after each change in β. Assuming the GAP is clocked at 10 MHz for 70 generations with a population of 64 members then according to Figure 6.12, the total

time required to complete one step in Figure 9.14 is around 100ms. Therefore β can be varied at a rate of up to 10 Hz.

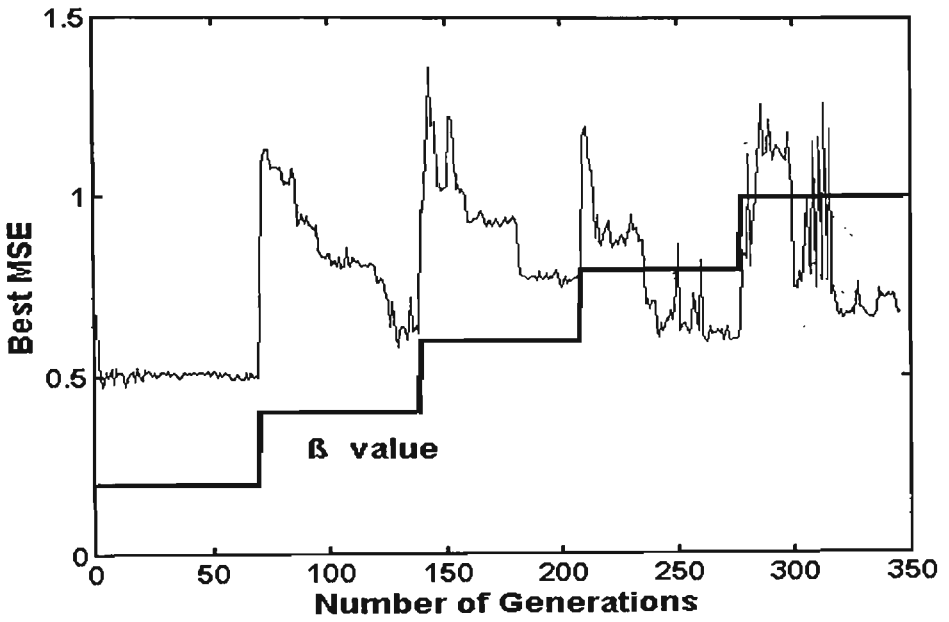


Figure 9.16: The minimum MSE for the adaptive IIR filter when β is varied in (9.5).

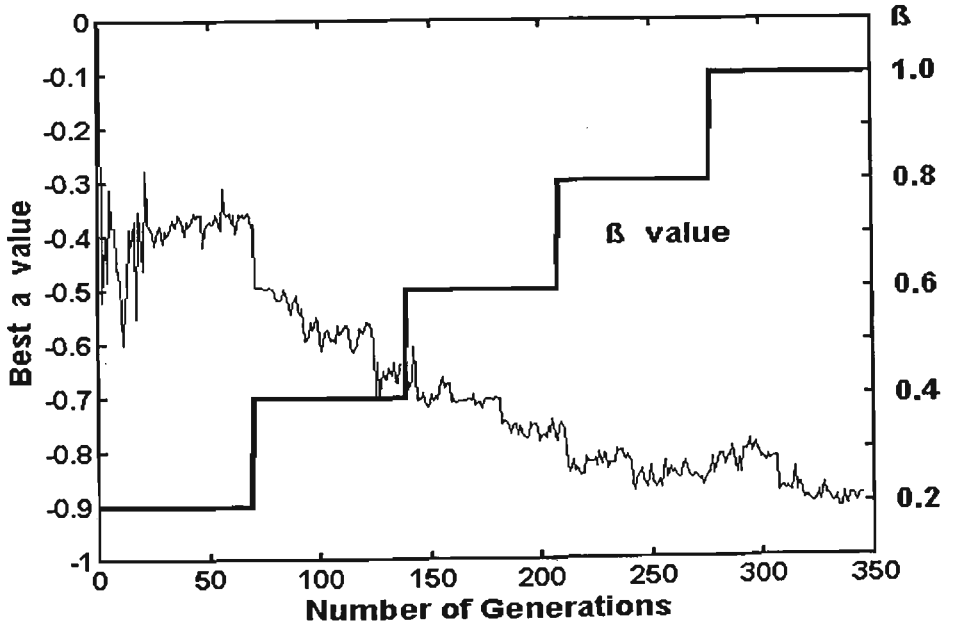


Figure 9.17: The best 'a' value for the adaptive IIR filter when β is varied in (9.5).

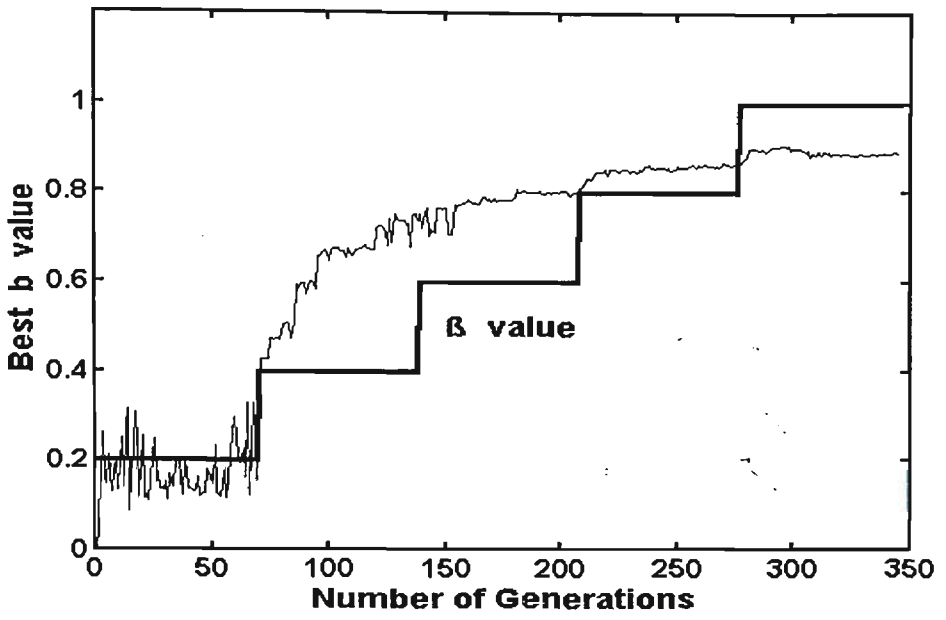


Figure 9.18: The best 'b' value for the adaptive IIR filter when β is varied in (9.5).

9.4 Conclusions

This chapter describes experiments to test the adaptive behaviour of the GAP. The applications show how multiple configurations of the GAP can adapt to sudden changes in the plant. One problem is notifying the GAP when changes occur. If the GAP is not notified of the change, then it is not possible for it to modify the population for adaptation. This notification can be made by an external signal or internally by testing for sudden changes in the fitness values. In all simulations here, an external signal was used. This choice is safe and without error but needs to be provided from the application to the GAP. The other method is better but there are chances of errors in detecting changes from within the GAP. Further research is needed to design a good algorithm for detecting changes in the environment, based on changes in the population's fitness values. Such an algorithm could be included in the GAP hardware.

There are several other constrains and limitations to the application of the GAP to adaptive control. These are discussed in more details in the next chapter.

Chapter 10

Conclusions and future work

Prior to this research there has been little reported work on hardware implementations of Genetic Algorithms. Most GA research to date has concentrated on the design and application of software GAs to search problems with static evaluation functions. It has always been assumed that GAs are simple to implement in hardware and that emphasis should be placed on software development and testing because of the flexibility and versatility of software.

This thesis has taken a different view. A hardware model of the GA was written in VHDL in Chapter 4 and analysed in Chapter 5 and is proposed as a candidate model for hardware implementation. However for hardware implementation of GAs to be useful in engineering applications there are many problems to be answered. In summary, this thesis has addressed a wide range of issues concerning hardware genetic algorithms. These range from the obvious speed advantage over software GAs to the more subtle

issues concerning minimal architectures that still operate satisfactorily on engineering applications.

10.1 Hardware implementation issues

This thesis has addressed the following issues concerned with hardware implementation of GAs.

- What is the speed advantage of the hardware GA over software GA's?

Simple tests demonstrate that the GAP operates about 35 times as fast as a similar software GA assuming the same clock speed. Other design ideas are suggested which should improve the GAP's performance. These improvements could realistically make the GAP 100 times faster than the software-based GA although these speedups ignore the time spent in fitness evaluation.

- Can GAs be implemented on low cost reprogrammable technology and what is the minimum hardware configuration required to implement a simple GA?

It has been shown in Chapter 6 that the GAP can be implemented on FPGA technology. The four bit GAP can be downloaded into a single Xilinx XC4013 chip. To construct a GAP with larger member size in current FPGA technology two or more chips are required. For a typical configuration of the GAP there is a trade off between the number of chips and the speed of the hardware. Using more chips results in higher speed but also higher costs.

- What is the performance of the hardware GA configuration on engineering problems?

In Chapter 6 and 7, the GAP is applied to 10 problems. In all cases it found a good solution to the defined problem but in most cases was unable to produce the best solution. This is due to the limitations of the simple Genetic Algorithm used in the

hardware implementation. If more sophisticated GAs are implemented, then we can expect better performance but the cost will increase. More discussion about applications will be presented in Section 10.2.

- Is there a simple but effective way of handling large member word length?

We can expect that with advances in FPGA technology (through the Xilinx XC6200 chip and beyond), it may soon be possible to fit the 16-bit and 32-bit versions onto a single chip. However many applications in engineering and science required a very large bit string like 64 to 1024 bits. Using more chips in the implementation is expensive and the resulting circuit will be slowed down by interconnection delays. A mechanism for splitting the string among multiple GAPs and using an array of GAPs to solve a problem is explained in Chapter 8. The limitations of this approach will be discussed in Section 10.3.

- What is the adaptive performance of the GAP?

Optimisation with a static evaluation function is often quite a simple problem for the GAP. Of real interest to engineering is its performance in a dynamic environment. Considering the stochastic and probabilistic nature of genetic algorithms, it is clear that this is going to be a very difficult problem for a long time to come. The research described in Chapter 9 demonstrates some success but only with the application of a number of constraints and limitations. The effect of these will be described further in Section 10.4.

10.2 Applications of the GAP

This thesis covered three applications in engineering. The simulations in Chapter 7 have been carried out under the following assumptions.

- A parameterised model of the plant was provided in each case.

The model is only required to compute the evaluation function and carry out the simulations. In real applications the GAP is directly connected to the plant and performance evaluation is usually carried out by direct measurement.

- It is assumed that the plant is capable of responding to any input supplied by the control system or filter which is being "tuned" by the GAP.

In many cases there will be some combinations of parameters which, if supplied by the GAP, will cause the plant to become unstable. Clearly there is a need for protection against instability at various points in the system.

- Maximum real-time performance can only be achieved if the fitness is always computed in a very short time < 1 GAP clock cycle.

To take full advantage of the GAP speed, it is necessary for the plant to return the fitness value within one clock cycle to the GAP. If the plant is slow and forces a wait status on the GAP operation then we will lose some of the speed advantage of the GAP. These timing limitations were described in Section 6.2.5.

The GAP prefers high speed applications where the fitness value can be returned in microseconds. For a slow application the GAP may not be a good choice, but the low cost of implementation should also be considered.

10.3 Multiple GAP configurations

By splitting each member string across several GAPs we can handle more complex problems while retaining the simplicity of the design. The main problems with the multiple GAP architecture can be expressed in the following questions.

- Is the multiple GAP architecture supported by any theory?

It is difficult to verify the operation of the multiple GAP using schema theory. The experimental results in Chapter 8 suggest that this configuration operates as well as the normal GAP. Further theoretical analysis is required to investigate the limitations of this configuration.

- Does the multiple GAP configuration work for all applications?

No: the simulations in Chapter 8 suggest the need to test the performance of the multiple GAP for each application. The performance depends on the fitness landscape of the problem and the type of coding used for the member string.

- What is the optimum configuration needed for any specific application?

This depends in part on the coding scheme selected for the problem. If we have a model of the problem then we would normally select the number of GAP's in the multiple configuration to match the number of independent variables in the model.

Using the GAP in the multiple configuration is more restricted by application because of the schema theory. However the full power of the GAP can be obtained using this configuration. In general, where the numbers of parameters are high or the bit string is long, the multiple GAP is the best choice for implementation.

10.4 Adaptive behaviour

Before any conclusions can be made about the adaptive behaviour of the GAP, it must be stressed that the simulations in Chapter 9 have been carried out under the following idealised conditions:

- Only one parameter of each plant or filter was ever varied in the tests.

Normally when the adaptive system responds to changes correctly it means that it is capable of handling any change in the application. It is possible to change two or more

parameters in the application model and it is expected that the GAP would still operate correctly.

- Changes which occurred in the plant or filter were sudden and the GAP model was notified immediately of any changes.

Chapter 9 explains why the adaptive GAP is best suited to the sudden changes in the fitness evaluation. Most control systems are not designed for sudden changes in the environment and system for handling these situations is thus very useful. Applying the GAP in adaptive applications is a little difficult because the GAP somehow must recognise changes in the application. A watchdog circuit could be designed inside GAP hardware to detect changes and thus make the GAP independent of external notification but this needs to implemented carefully. If the GAP could detect these changes, then it may be applied to many adaptive applications.

- Changes were always followed by a long period of static behaviour to allow the GAP to settle before the next change occurred.

The GAP requires many generations to find an optimum point after every change. During the search it expects the objective function to remain stationary. Adaptive performance will decline and fail if further changes occur before the GAP has had time to settle.

10.5 Some potential applications of the GAP

The three engineering's applications show the robustness of the GAP model in solving various problems. These applications are all quite simple for GAs and conventional search methods are often more capable of finding solutions than the GAP. It is important however to realise that the same generic hardware is capable of handling all applications without changing the core model of the GAP. There are many potential applications of the GAP. Some examples are listed here.

- Chemical process control in which the GAP is required to control flows of chemicals into a reaction chamber in order to meet some objective criteria such as a nominated pH level.

- Telescope focussing systems in which multiple mirrors are mechanically adjusted to focus an image.

- Vehicle engine management systems in which the GAP could adapt the controlling coefficients (in the existing controller) to match the individual engine characteristics.

In most of these applications processing speed is not as important as the advantage of having a compact stand-alone hardware implementation.

Appendix A

A simple Genetic Algorithm

Given a search problem, with a multi-dimensional space of possible solutions, a "genetic code" representation is chosen in a way that each point in the search space is represented by a string of symbols, the chromosome. A random number generator produces a number of initial random chromosomes, which form the initial population. Each of the corresponding points in the search space is evaluated by the appropriate evaluation function. This function gives higher scores to the 'fitter' ones, those nearest the required solution.

The next generation of points is created from the present population by selection and reproduction. The selection process is based on the scores of the present population, such that the fitter chromosomes probabilistically contribute more to the reproductive pool.

From the reproductive pool of selected chromosomes, a new set of chromosomes for the next generation is derived, using such genetic operators as crossover and mutation. The crossover operator works by taking parent chromosomes in pairs, selecting a crossover point somewhere at random along the length of the chromosomes, taking the left-hand section of one parent up to the crossover point and joining it to the right-hand section of the other parent, so that offspring inherits genetic material from both parents. The mutation operator changes at random a very small proportion of the symbols on the offspring chromosomes to some other valid symbol.

The new population has inherited genetic material selectively, but probabilistically, from the parent generation. New points in the search space have been generated exploiting the information from the performance of the parents. Thus an improved performance can be expected from the new population. Then the cycle of selection and reproduction can then be repeated.

Variations on these genetic operators can be used, and decisions have to be made on suitable sizes of populations and the rate at which mutation and other operators are applied. In the choice of selection techniques, it is necessary to maintain a balance between enough selective pressure for a continued improvement and too much selective pressure leading to premature convergence, with loss of the diversity needed to escape from local optima. However, GAs are a very robust search technique that can operate successfully in many varied search domains. Some of these properties can be demonstrated theoretically.

A.1 Theory of Genetic Algorithms

An early theoretical result by Holland [1975] was the Schema Theorem which demonstrates that subject to certain conditions, schemata, which are the building blocks in his representation of above average fitness, will receive exponentially increasing numbers of trials in successive generations. A schema is a similarity template which

characterises a subset of all possible strings that have identical specified values at special positions specified by a template. At other positions often indicated by * in the schema template, any values are allowed. Holland also demonstrated the implicit parallelism of GAs. In a population of size m, the number of these schemata being processed in each generation is of order $O(m^3)$ which indicates that a GA is an efficient search algorithm. These factors contribute to the theoretical support to applications of GAs.

A ternary schema alphabet C = {0,1,*} will be used as an example for a binary string alphabet C' = {0,1}. For example, a schema H = 01*1 will represent strings 0111 and 0101. For a given schema H, its "order" o(H) is the number of its fixed positions (e.g. if H = 0*1* then o(H) = 2). Finally, for a given schema H, its "defining length" δ(H) is the distance between the first and last specific string positions (e.g. if H = 0*1* then δ(H) = 3 - 1 = 2).

Typically, GAs are used on well-defined problems where the search space is so large that other approaches are computationally impractical. The problems the user faces are: what choice of genetic coding to use, what variations to make on the standard genetic operators how to select members for reproduction, and how to set the balance of selective pressure.

A.2 A simple example of a Genetic Algorithm

As a simple example, imagine a population of four strings, each with five bits. Also imagine an objective function f(x) = 4x, which simply returns the integer value of four times the binary integer (e.g. f(00000) = 0, f(00001) = 4, f(00010) = 8, etc.). The goal is to optimise (in this case maximise) the objective function over the domain $0 \leq x \leq 31$. Now imagine a population of the four strings in Table A.1, generated at random before GA execution. The corresponding fitness values and percentages come from the objective function f(x).

i	String *x(i)*	Fitness *f(i)* *f(x(i))=4x(i)*	% of Total *f(i)/Σf(i)*	Expected Count *f(i)/Avg*	Actual Count
1	11100	112	41.8	1.670	2
2	01101	52	19.4	0.776	1
3	10010	72	26.9	1.047	1
4	01000	32	11.9	0.477	0
	Sum	268	100	4.000	4
	Avg	67	25.0	1.000	1
	Max	112	41.8	1.670	2

Table A.1: Four random strings and their fitness values.

The values in the "% of Total" column provide the probability of each string's selection. Initially 11100 has a 41.8% chance of selection, 01101 has 19.4% chance, and so on. The selection process can be thought of as spinning a weighted roulette wheel like in Figure A.1. The results from the spins are given in the "Actual Count" column of Table A.1. As expected, these values are similar to those in the "Expected Count" column.

After selecting the strings, the GA randomly pairs the newly selected members and looks at each pair individually. For each pair (e.g. A = 11100 and B = 01101), the GA decides whether or not to perform crossover. If it does not, then both strings in the pair are placed into the population with possible mutations as described below. If it does, a random crossover point is selected and crossover proceeds as indicated in Figure A.2. Then the children A' and B' are placed in the population with possible mutations. The GA invokes the mutation operator on the new bit strings. It generates a random number for each bit and flips that particular bit only if the random number is less than or equal to the mutation probability which is usually less than 0.01 per bit .

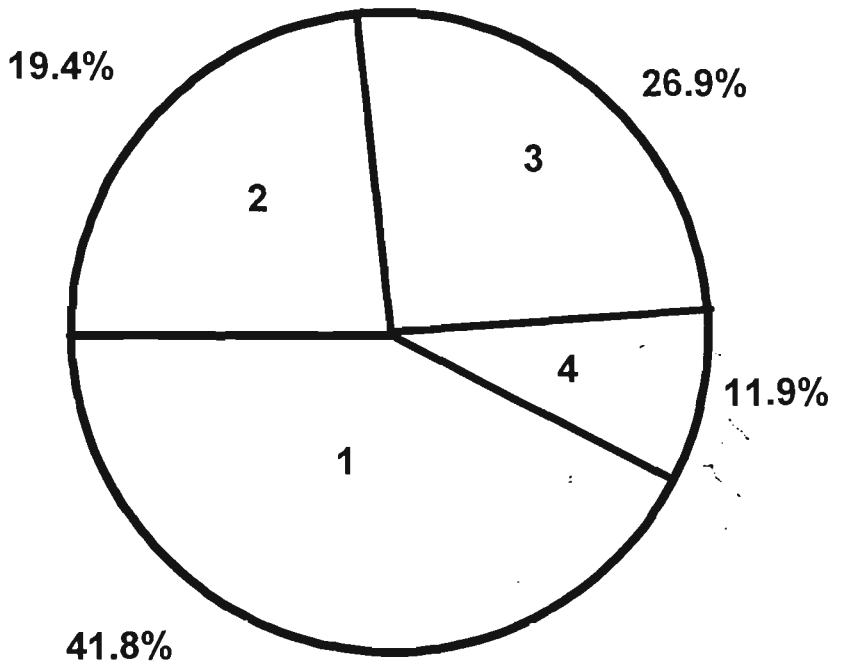


Figure A.1: A weighted roulette wheel.

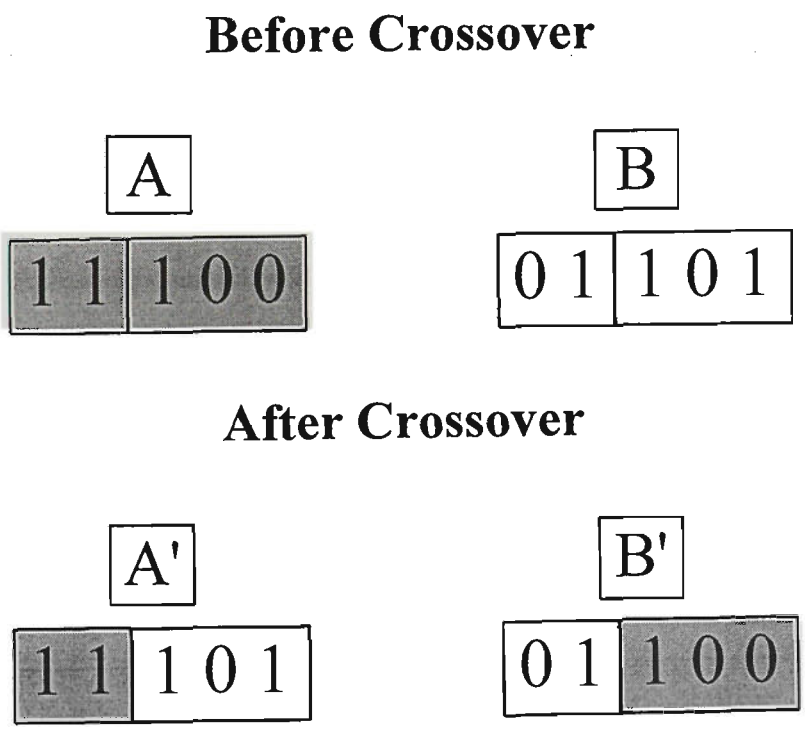


Figure A.2: An example of crossover.

After the operations are completed, on the current generation, the new strings are placed in a new population, representing the next generation as shown in Table A.2. In this example the average fitness increased by approximately 29% in one generation and the sum of fitness increased by 30%. This simple process would continue for several generations until a stopping criterion is met.

After Reproduction	Associate Parent	Crossover Point	After Crossover	Fitness f(x(i))=4x(i)
11|100	x3	2	11101	116
111|00	x4	3	11110	120
01|101	x1	2	01100	48
100|10	x2	3	10000	64
Sum				348
Avg.				87
Max				120

Table A.2: The population after applying selection and crossover operators.

Appendix B

Gray code conversion

Translation of a decimal number into the Gray code equivalent is carried out by first transforming the decimal number into a binary representation. Modulo-2 addition is then performed on each bit with its immediate neighbour on the left. Translation of this result into decimal notation then gives the Gray code equivalent. Thus 11_{10} become 1011_2 and modulo-2 addition of pairs of bits gives 1110_2 or 14_{10}.

This may be expressed in terms of a binary bit string as follows. A binary number may be expressed as

$$B=(b_m, b_{m-1}, \ldots, b_0)_2 \tag{B.1}$$

where b_i gives the position in the binary number. This is given in the Gray code as

$$B_g=(g_m, g_{m-1}, \ldots, g_0)_2 \tag{B.2}$$

where $g_i=b_i+b_{i+1}$.

A Gray code conversation table is given in Table B.1 for the first 16 decimal numbers. Table B.2 shows the Gray code conversion for 5 bit or the first 32 decimal number.

Decimal	Binary code	Gray Code	Decimal	Binary code	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

Table B.1: Binary code to Gray code for four bits.

Decimal	Binary code	Gray Code	Decimal	Binary code	Gray code
0	00000	00000	16	10000	11000
1	00001	00001	17	10001	11001
2	00010	00011	18	10010	11011
3	00011	00010	19	10011	11010
4	00100	00110	20	10100	11110
5	00101	00111	21	10101	11111
6	00110	00101	22	10110	11101
7	00111	00100	23	10111	11100
8	01000	01100	24	11000	10100
9	01001	01101	25	11001	10101
10	01010	01111	26	11010	10111
11	01011	01110	27	11011	10110
12	01100	01010	28	11100	10010
13	01101	01011	29	11101	10011
14	01110	01001	30	11110	10001
15	01111	01000	31	11111	10000

Table B.2: Binary code to Gray code for 5 bits.

For converting Gray codes to the binary codes suppose the Gray code is defined as

$$B_g = (g_m, g_{m-1}, \ldots, g_0)_2 \tag{B.3}$$

where g_i gives the position in the Gray code. This may be expressed in terms of a binary bit string as follows.

$$B = (b_m, b_{m-1}, \ldots, b_0)_2 \tag{B.4}$$

where $b_i = \sum_{j=m}^{i} g_j$.

A Gray code conversation to binary table is given in Table B.3 for the first 16 decimal numbers. Table B.4 shows the Gray code conversion to binary for 5 bits or the first 32 decimal numbers.

Decimal	Gray code	Binary Code	Decimal	Gray code	Binary code
0	0000	0000	8	1000	1111
1	0001	0001	9	1001	1110
2	0010	0011	10	1010	1100
3	0011	0010	11	1011	1101
4	0100	0111	12	1100	1000
5	0101	0110	13	1101	1001
6	0110	0100	14	1110	1011
7	0111	0101	15	1111	1010

Table B.3: Gray code to binary code for four bits.

Decimal	Gray code	Binary Code	Decimal	Gray code	Binary Code
0	00000	00000	16	10000	11111
1	00001	00001	17	10001	11110
2	00010	00011	18	10010	11100
3	00011	00010	19	10011	11101
4	00100	00111	20	10100	11000
5	00101	00110	21	10101	11001
6	00110	00100	22	10110	11011
7	00111	00101	23	10111	11010
8	01000	01111	24	11000	10000
9	01001	01110	25	11001	10001
10	01010	01100	26	11010	10011
11	01011	01101	27	11011	10010
12	01100	01000	28	11100	10111
13	01101	01001	29	11101	10110
14	01110	01011	30	11110	10100
15	01111	01010	31	11111	10101

Table B.4: Gray code to Binary code for 5 bits.

Appendix C

VHDL code

This appendix presents a sample of the VHDL code for the basic GAP design and was used for implementation and most GAP simulations. The functionality of each module is described in Chapter 4. The code was written and compiled using the Design Architect and QuickVHDL programs from Mentor Graphics and then synthesized using the AutoLogic VHDL synthesiser.

This VHDL code is the intellectual property of the Victoria University of Technology and will be available for use under licence. For further details contact:

The Secretary,
Faculty of Engineering,
Victoria University of Technology,
PO BOX 14428, MCMC,
Melbourne VIC., 3001
Australia

C.1 Random Number Generator Module

```
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;
USE work.sizing.all;

ENTITY random IS
PORT   (
        init            :IN  qsim_state;                    -- Initial signal
        done            :OUT qsim_state;                    -- finish signal
        clk             :IN  qsim_state;                    -- clock signal
        reqmem          :OUT qsim_state;                    -- bus request to memory controller
        ackmem          :IN  qsim_state;                    -- bus ack from memory controller
        domut           :OUT qsim_state_vector(p-1 DOWNTO 0); -- random numbers of xover and
mutation
        doxover :       OUT qsim_state_vector(p-1 DOWNTO 0);
        mutpt           :OUT qsim_state_vector(logn-1 DOWNTO 0);
        xoverpt :       OUT qsim_state_vector(logn-1 DOWNTO 0);
        addr            :OUT qsim_state_vector(lognumparam-1 DOWNTO 0);-- address of initial
parameter
        param           :IN  qsim_state_vector(valw-1 DOWNTO 0);-- value of RNG seed
        randsel1        :OUT qsim_state_vector(r-1 DOWNTO 0);  -- random nos. to selection
module
        randsel2        :OUT qsim_state_vector(r-1 DOWNTO 0)
        );
END random;

ARCHITECTURE behave OF random IS
TYPE states IS (idle, awaitackmem1, awaitackmem2, active);
SIGNAL state:states:=idle;
BEGIN
randomprocess:PROCESS(clk,init)
VARIABLE rn,rn2 : qsim_state_vector(casize-1 DOWNTO 0);
VARIABLE evenodd : qsim_state;
BEGIN
IF init='0' THEN
        state<=idle;
        reqmem<='0';
        done<='0';
ELSIF(clk'EVENT and clk='1' and clk'LAST_VALUE='0') THEN
        CASE state IS
                WHEN idle =>
                        reqmem<='1';
                        state<=awaitackmem1;
                WHEN awaitackmem1 =>
```

```
                    IF ackmem='1' THEN
                        addr<=rngseeda;
                        reqmem<='0';
                        state<=awaitackmem2;
                    END IF;
                WHEN awaitackmem2 =>
                    IF ackmem='0' THEN
                        rn:=param(valw-1 DOWNTO 0) & param(valw-1 DOWNTO 0) &
param(valw-1 DOWNTO 0) & param(valw-1 DOWNTO 0);
                        done<='1';
                        state<=active;
                    END IF;
                WHEN active =>
                    domut <= rn(casize-1 DOWNTO casize-p);
                    doxover <= rn(casize-4 DOWNTO casize-p-3);
                    mutpt <= rn(casize-6 DOWNTO casize-logn-5);
                    xoverpt <= rn(casize-5 DOWNTO casize-logn-4);
                    randsel1 <= rn(casize-3 DOWNTO casize-2-r);
                    randsel2 <= rn(casize-6 DOWNTO casize-5-r);
                    rn2(casize-1) :='0' XOR rn(casize-1) XOR rn(casize-2);
                    evenodd :='1';
                    FOR i IN casize-2 DOWNTO 1 loop
                        rn2(i) :=rn(i+1) XOR rn(i-1);
                        IF evenodd='1' THEN
                            rn2(i):=rn2(i) XOR rn(i);
                        END IF;
                    evenodd:= NOT evenodd;
                    END LOOP;
                rn2(0) := rn(1) XOR '0';
                IF evenodd='1' THEN
                    rn2(0):=rn2(0) XOR rn(0);
                END IF;
                rn:=rn2;
        END CASE;
END IF;
END PROCESS randomprocess;
END behave;
```

C.2 Memory Interface Module

```
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;
USE work.sizing.all;

ENTITY memory IS
PORT    (
        go              :IN  qsim_state;        -- go ahead signal
        done            :OUT qsim_state;        -- completion signal
        reset           :IN  qsim_state;        -- asynchronous reset
```

```
        init            :OUT  qsim_state;                   -- initialize signal
        clk             :IN  qsim_state;                    -- clock signal
        address         :OUT qsim_state_vector(addrw-1 DOWNTO 0);      -- Memory buses
        datain          :IN  qsim_state_vector(valw -1 DOWNTO 0);      --
        dataout         :OUT qsim_state_vector(valw -1 DOWNTO 0);      --
        rw              :OUT qsim_state;                    -- read/write signal to memory
        cs              :OUT qsim_state;                    -- chip select signal to memory
        oe              :OUT qsim_state;                    -- output enable signal to memory
        memacc          OUT qsim_state;                     -- tri-states data data, addr and ctrl lines to
memory
        toggle          :IN  qsim_state;                    -- population access for fitness and
memory sequencer
        toggleout       :OUT qsim_state;                    -- final value of toggle
        reqrng          :IN  qsim_state;            -- Handshake signals with random number
generator
        ackrng          :OUT qsim_state;            --
        addrrng         :IN  qsim_state_vector(lognumparam-1 DOWNTO 0);        --
        reqxov          :IN  qsim_state;            -- Handshake signals with crossover
        ackxov          :OUT qsim_state;--
        addrxov         :IN  qsim_state_vector(lognumparam-1 DOWNTO 0);        --
        reqseq          :IN  qsim_state_vector( 1 DOWNTO 0);     -- Handshake signals with read
        ackseq          :OUT qsim_state;--
        addrseq         :IN  qsim_state_vector(logm-1 DOWNTO 0);       --
        reqfit          :IN  qsim_state_vector( 1 DOWNTO 0);     -- Handshake signals with fitness
        ackfit          :OUT qsim_state;--
        addrfit         :IN  qsim_state_vector(logm-1 DOWNTO 0);       --
        valfitin        :IN  qsim_state_vector(valw-1 DOWNTO 0);       --
        valout          :OUT qsim_state_vector(valw-1 DOWNTO 0);       --
        fitdone         :IN  qsim_state;                    --
        flag            :IN  qsim_state                     -- adaptive signal
        );
END memory;

ARCHITECTURE behave OF memory IS
TYPE states IS (start1, start2, idle, rng, xov, fit0, fit1, seq0, seq1,
                done1, done2, write1, write2 , write3 , read1, read2);
SIGNAL state: states:=start1;
constant pop0base :integer:=numparam;
constant pop1base :integer:=numparam+m;

BEGIN

memoryprocess:PROCESS (clk,reset)
VARIABLE runstats : integer RANGE 0 TO 65535 :=0;
VARIABLE base :integer RANGE 0 TO memsize-1;
VARIABLE finish :qsim_state:='0';
VARIABLE tempflag:qsim_state;

BEGIN
IF reset='1' THEN
        init<='0';
```

```
        done<='1';
        rw<='1';
        cs<='1';
        oe<='1';
        memacc<='1';
        runstats:=0;
        finish:='0';
        ackrng<='0';
        ackxov<='0';
        ackseq<='0';
        ackfit<='0';
        state <=start1;
ELSIF(clk'EVENT and clk='1' and clk'LAST_VALUE='0') THEN
        CASE state IS
                WHEN start1 =>
                        init<='0';
                        done<='1';
                        rw<='1';
                        cs<='1';
                        tempflag:=flag;
                        oe<='1';
                        finish:='0';
                        memacc<='1';
                        ackrng<='0';
                        ackxov<='0';
                        ackseq<='0';
                        ackfit<='0';
                        IF go='0' THEN
                                done<='0';
                                state<=start2;
                        END IF;
                WHEN start2 =>
                        init<='0';
                        done<='0';
                        rw<='1';
                        cs<='1';
                        oe<='1';
                        finish:='0';
                        memacc<='1';
                        ackrng<='0';
                        ackxov<='0';
                        ackseq<='0';
                        ackfit<='0';
                        IF go='1' THEN
                                init<='1';
                                memacc<='0';
                                cs<='0';
                                state<=idle;
                        END IF;

                WHEN idle  =>
```

```
            IF flag/=tempflag THEN
                    runstats :=0;
                    tempflag := NOT tempflag;
            END IF;
            runstats :=runstats+1;
            memacc <='0';
            finish:='0';
            init <='1';
            cs<='0';
            IF fitdone='1' THEN
                    init<='0';                              -- shut down GA
                    address <= to_qsim_state(numparam+m+m,addrw);
                    finish:='1';
                    state <=write1;
            ELSIF reqrng='1' THEN
                    ackrng<='1';
                    state<=rng;
            ELSIF reqxov='1' THEN
                    ackxov<='1';
                    state<=xov;
            ELSIF reqseq(0)='1' THEN
                    ackseq<='1';
                    state<=seq0;
            ELSIF reqfit(0)='1' THEN
                    ackfit<='1';
                    state<=fit0;
            ELSIF reqfit(1)='1' THEN
                    ackfit<='1';
                    state<=fit1;
            ELSIF reqseq(1)='1' THEN
                    ackseq<='1';
                    state<=seq1;
            END IF;

    WHEN rng  =>
            runstats :=runstats+1;
            IF reqrng='0' THEN
                    address<=to_qsim_state(0,addrw-lognumparam) & addrrng;
                    state<=read1;
            END IF;

    WHEN xov  =>
            runstats :=runstats+1;
            IF reqxov='0' THEN
                    address<=to_qsim_state(0,addrw-lognumparam) & addrxov;
                    state<=read1;
            END IF;

    WHEN fit0  =>
            runstats :=runstats+1;
            IF reqfit(0)='0' THEN
```

```
                        address<=to_qsim_state(0,addrw-lognumparam) &
addrfit(lognumparam-1 DOWNTO 0);
                        state<=read1;
                    END IF;

                WHEN fit1  =>
                    runstats :=runstats+1;
                    IF reqfit(1)='0' THEN
                        IF toggle='0' THEN
                            base:=pop1base;
                        ELSE
                            base:=pop0base;
                        END IF;
                        address<=to_qsim_state(base+to_integer('0'& addrfit),addrw);
                        dataout <= valfitin;
                        state<=write1;
                    END IF;

                WHEN seq0  =>
                    runstats :=runstats+1;
                    IF reqseq(0)='0' THEN
                        address<=to_qsim_state(0,addrw-lognumparam) &
addrseq(lognumparam-1 DOWNTO 0);
                        state<=read1;
                    END IF;

                WHEN seq1  =>
                    runstats :=runstats+1;
                    IF reqseq(1)='0' THEN
                        IF toggle='0' THEN
                            base:=pop0base;
                        ELSE
                            base:=pop1base;
                        END IF;
                        address<=to_qsim_state(base+to_integer('0' & addrseq),addrw);
                        state<=read1;
                    END IF;

                WHEN done1  =>
                    rw<='0';
                    state<=done2;

                WHEN done2  =>
                    toggleout<=NOT toggle;
                    init<='0';
                    rw<='1';
                    cs<='1';
                    oe<='1';
                    memacc<='1';
                    runstats:=0;
                    done<='1';
```

```
                state<=start1;

            WHEN write1 =>
                runstats :=runstats+1;
                rw <='0';
                state<=write2;

            WHEN write2 =>
                runstats :=runstats+1;
                rw <='1';
                state<=write3;

            WHEN write3 =>
                IF finish='1' THEN
                    address <= to_qsim_state(numparam+m+m+1,addrw);
                    state <= done1;
                ELSE
                    ackfit <='0';
                    state<= idle;
                END IF;
            WHEN read1 =>
                runstats :=runstats+1;
                rw<='1';
                oe<='0';
                state<=read2;

            WHEN read2 =>
                runstats :=runstats+1;
                valout <= datain;
                ackrng<='0';
                ackxov<='0';
                ackseq<='0';
                ackfit<='0';
                oe <='1';
                state<= idle;

        END CASE;
END IF;
END PROCESS memoryprocess;
END behave;
```

C.3 Read Module

```
LIBRARY mgc_portable;
USE mgc_portable.qsim_logic.all;
USE work.sizing.all;

ENTITY read IS
PORT    (
```

```
        init    :IN  qsim_state;                    -- initial signal
        done    :OUT qsim_state;
        clk     :IN  qsim_state;                    -- clock signal
        reqmem :OUT qsim_state_vector(1 DOWNTO 0);          -- signal to memory
        addr    :OUT qsim_state_vector(logm-1 DOWNTO 0);
        value   :IN  qsim_state_vector(valw-1 DOWNTO 0);
        outputn :OUT qsim_state_vector(n-1 DOWNTO 0);
        outputf :OUT qsim_state_vector(f-1 DOWNTO 0);
        ackmem :IN  qsim_state
        );
END read;

ARCHITECTURE behave OF read IS
TYPE states IS (idle,awaitackmem1,awaitackmem2,getmember1n,
        getmember2n,getmember1f,getmember2f);
SIGNAL state : states :=idle;
BEGIN
readprocess:PROCESS(clk,init)

        VARIABLE membf : qsim_state_vector(f-1 DOWNTO 0);
        VARIABLE membaddr, psize: INTEGER RANGE 0 TO m;

BEGIN
        IF init='0' THEN            -- not OK to run, should be idle
                state <=idle;               -- Make sure idle
                done <= '0';                -- tell control unit shut down
                reqmem <= "00";             -- reset memory request signal;
                membf := to_qsim_state(0,f);        -- reset current member
                outputn <= to_qsim_state(0,n);      -- output the reset member
                outputf <= to_qsim_state(0,f);      -- output the reset member
        ELSIF(clk'EVENT and clk='1' and clk'LAST_VALUE='0') THEN
                CASE state IS
                        WHEN idle =>                        -- state is idle, So get the population size
                                reqmem(0)<='1';             --request memory access
                                state<= awaitackmem1;       -- wait for memory acknoledgement
                        WHEN awaitackmem1 =>                -- waiting for memory acknoledgement
                                IF ackmem='1' THEN          -- memory acknoledgement
                                        addr <=to_qsim_state(to_integer('0' & popsizea),logm);
        -- send population size address
                                        reqmem(0)<='0';             -- address sent
                                        state<= awaitackmem2;       -- wait for memory
acknoledgement for sending population size
                                END IF;
                        WHEN awaitackmem2 =>                        -- waiting for memory
acknoledgement for sending population size
                                IF ackmem='0' THEN
                                        psize := to_integer('0' & value(logm-1 DOWNTO 0));
                                        membaddr :=0;               -- initialize membaddr
                                        done <= '0';
                                        reqmem(1) <= '1';           -- request memory access to get
member
```

```
                        state <= getmember1n;-- wait for memory acknoledgement
                    END IF;
                WHEN getmember1n =>                    -- waiting for memory to offer
member
                    done <='0';                        -- reset dup acknoledgement
                    IF ackmem='1' THEN                 -- got memory acknoledgement
                        addr <=to_qsim_state(membaddr,logm);-- send member's
address
                        reqmem(1)<='0';                -- tell memory memory's address
sent
                        state<= getmember2n;           -- wait for memory
acknoledgement
                    END IF;
                WHEN getmember2n =>                    -- wait for memory to send
member
                    IF ackmem='0' THEN                 -- memory says member sent
                        outputn <= value(n-1 DOWNTO 0);                --
output member to selection module
                        membaddr :=membaddr+1;         -- increment the
memory's address mod popsize
                        reqmem(1) <='1'; -- request fitness
                        state <= getmember1f;          -- wait for memory to offer
member
                    END IF;
                WHEN getmember1f =>                    -- waiting for memory to offer
member
                    IF ackmem='1' THEN                 -- got memory acknoledgement
                        addr <=to_qsim_state(membaddr,logm);       -- send
member's address
                        reqmem(1)<='0';                -- tell memory memory's address
sent
                        state <= getmember2f;          -- wait for memory
acknoledgement
                    END IF;
                WHEN getmember2f =>                    -- wait for memory to send
member
                    IF ackmem='0' THEN                 -- memory says member sent
                        done <= '1';                   -- start reading values
                        membf := value(f-1 DOWNTO 0);          -- store member
                        outputf <= membf;                      -- output member to
selection module
                        membaddr :=membaddr+1;         -- increment the
memory's address mod popsize
                        IF membaddr > psize -1 THEN
                            membaddr :=0;
                        END IF;
                        reqmem(1) <='1'; -- request next member
                        state <= getmember1n;          -- wait for memory to offer
member
                    END IF;
            END CASE;
```

```
        END IF;
END PROCESS readprocess;
END behave;
```

Appendix D

A brief description of Xilinx FPGAs

A field-programmable gate array is an inexpensive user-programmable component which allows for cheap prototyping. FPGAs are generally composed of programmable elements including logic blocks, I/O cells which connect the logic blocks to the chip pins, and interconnection lines. Programming of these components is allowed with the use of static RAM cells, anti-fuses, EPROM transistors or EEPROM transistors.

Xilinx FPGAs use static RAM technology to implement hardware designs. They are reprogrammable and frequently used in prototyping. Commonly used Xilinx FPGAs today are from the XC4000 family, which is currently Xilinx's most advanced line of FPGAs and includes the devices used in this research. Most of the information in this section is from the Xilinx technical literature (e.g. Xilinx, 1994).

D.1 Architecture of FPGAs

A Xilinx FPGA consists of a two-dimensional array of configurable logic blocks (CLBs), a set of surrounding input/output blocks (IOBs) and programmable inter connections between CLBs and IOBs (Figure D.1).

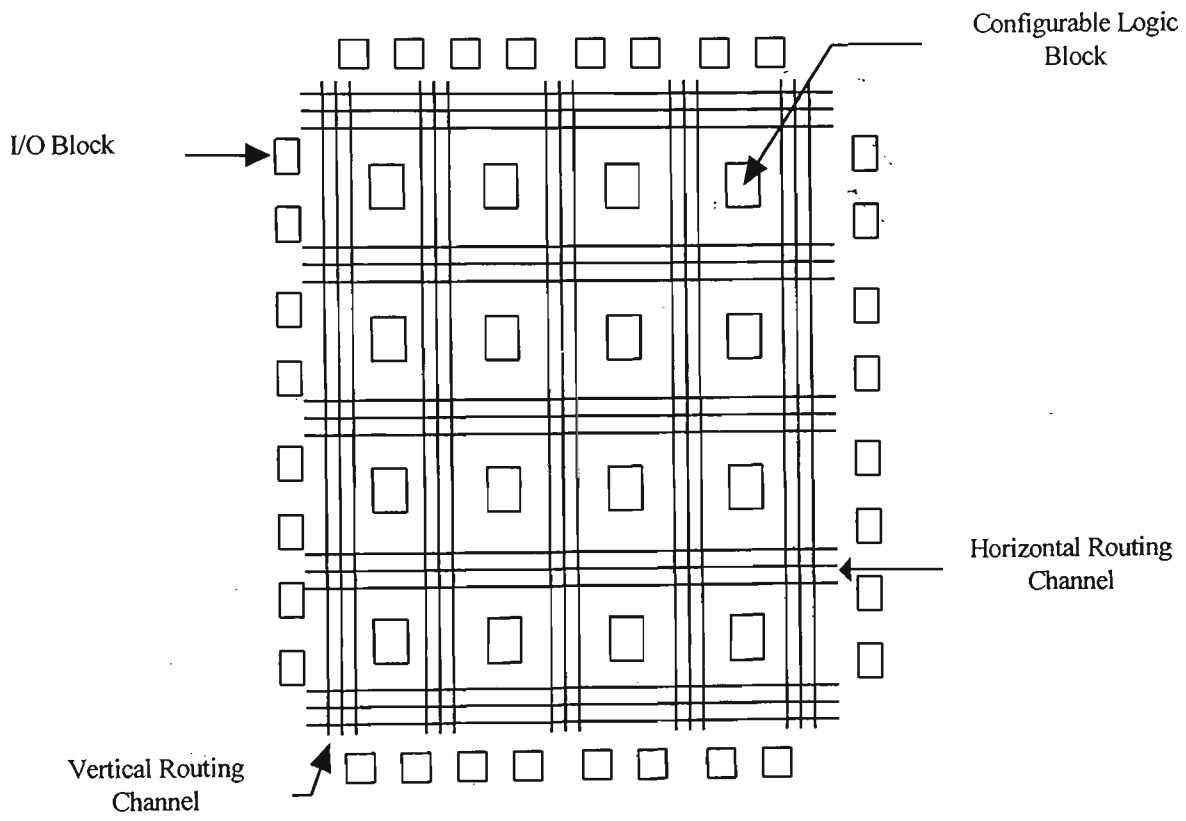


Figure D.1: Overall view of a Xilinx XC4000 series FPGA.

Each CLB (Figure D.2) can implement two arbitrary, independent four-input boolean functions, F and G. The outputs of F and G can be combined with another input in a third boolean function H. The outputs of F and G can be latched in edge-triggered D flip-flops. Each CLB also has the capacity to implement fast-carry logic. Alternatively, a CLB can be used as a 16 x 2 or a 32 x 1 array of memory cells.

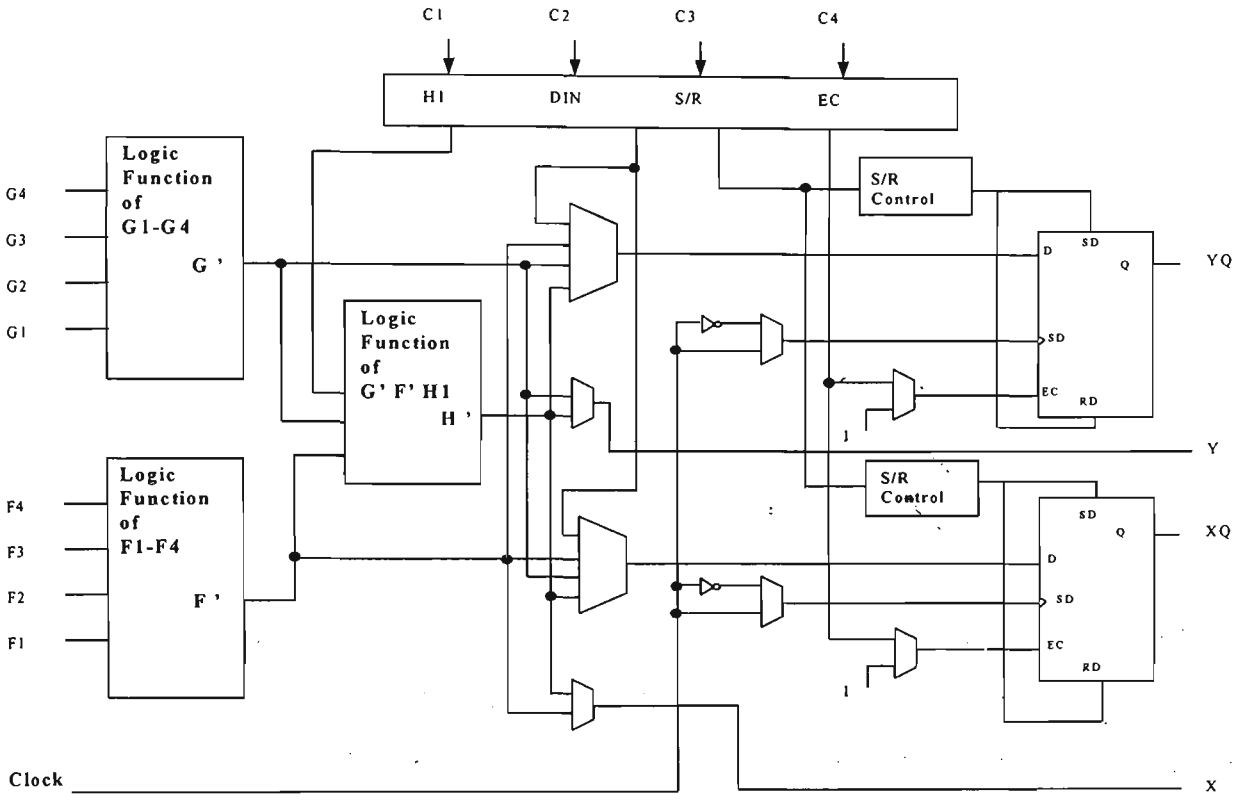


Figure D.2: Simplified schematic of a CLB.

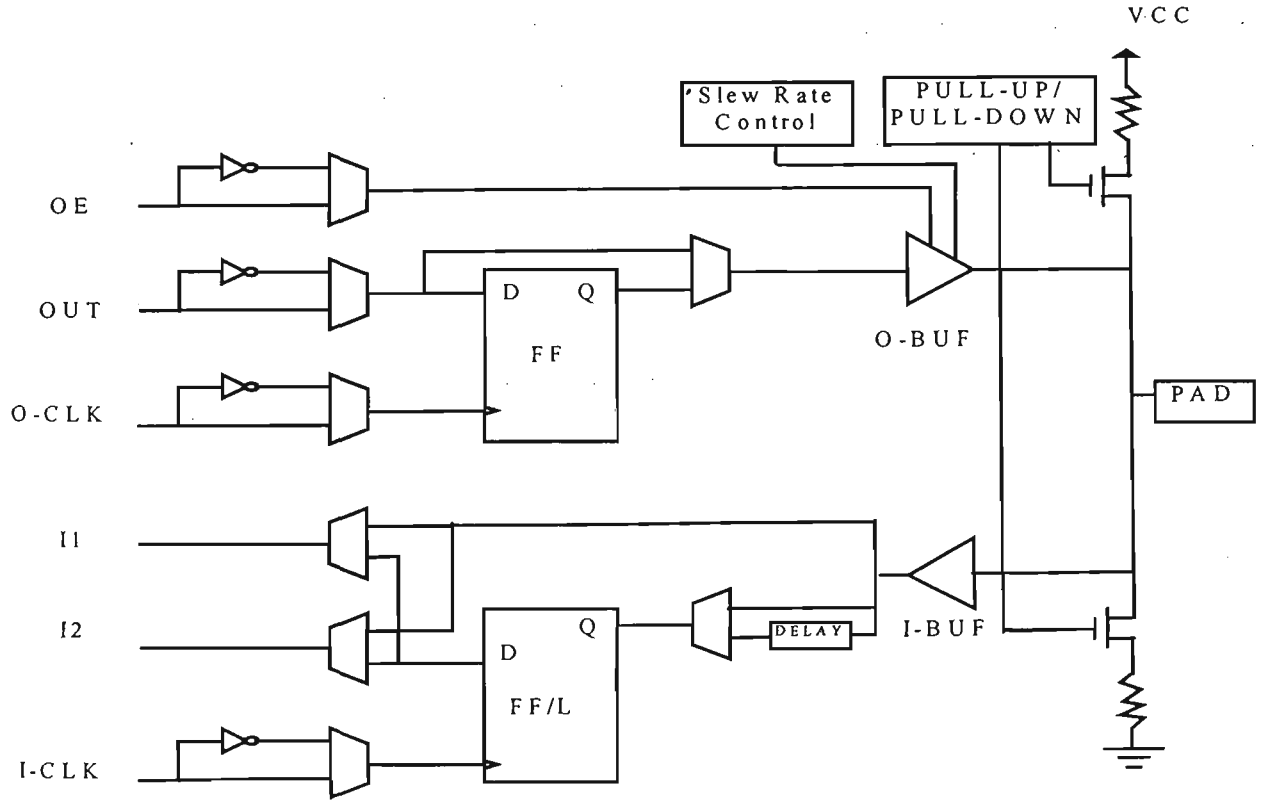


Figure D.3: An XC4000 IOB.

Connections between CLBs and the chip pads are provided by IOBs (Figure D.3). The IOBs offer many user-programmable options in I/O control, including tri-state logic for bidirectional I/O, direct connection of lines to pads or connection through flip-flops, and programmable pull-up or pull-down resistors. The IOBs also provide logic for boundary-scan testing and output slew rate control.

The interconnection between different CLBs and between CLBs and IOBs is also programmable. XC4000 interconnections are made through the use of single-length lines, double-length lines and longlines. Single-length lines (Figure D.4) intersect at a switch matrix between neighbouring CLBs in the horizontal and vertical directions. Any input to a switch matrix can be routed to any arbitrary outputs which then feed into other CLBs and switch matrices. Single-length lines are normally used to conduct signals within a localised area and to provide branching for nets with fanout greater than one. Double-length lines (Figure D.5) are similar to single-length lines except that they intersect after every two CLBs. Double-length lines provide the most efficient implementation of intermediate length, point-to-point interconnections. Longlines (Figure D.6) span the entire array of CLBs and are intended to carry time-critical signals. Longlines intersect single-length lines at programmable interconnect points. Double-length lines do not connect to other lines.

Logic densities for the most common XC4000 FPGAs are approximately 2000-10000 gates per chip. The costs for these FPGAs are approximately $50-$800 per chip. These logic densities are lower than those for fully customised VLSI chips and for mask-programmed gate arrays (MPGAs). FPGAs are also slower than fully customised VLSI chips and MPGAs. However, the costs per chip for low volumes and the short turnaround times make Xilinx FPGAs a better choice for prototyping than MPGAs and full-custom designs. Additionally, the reprogrammability of Xilinx FPGAs makes them more flexible than fully customised VLSI chips and MPGAs. Xilinx FPGAs can be

applied to systems that utilise reconfigurable hardware as described in Section 2.4 and as presented in this thesis.

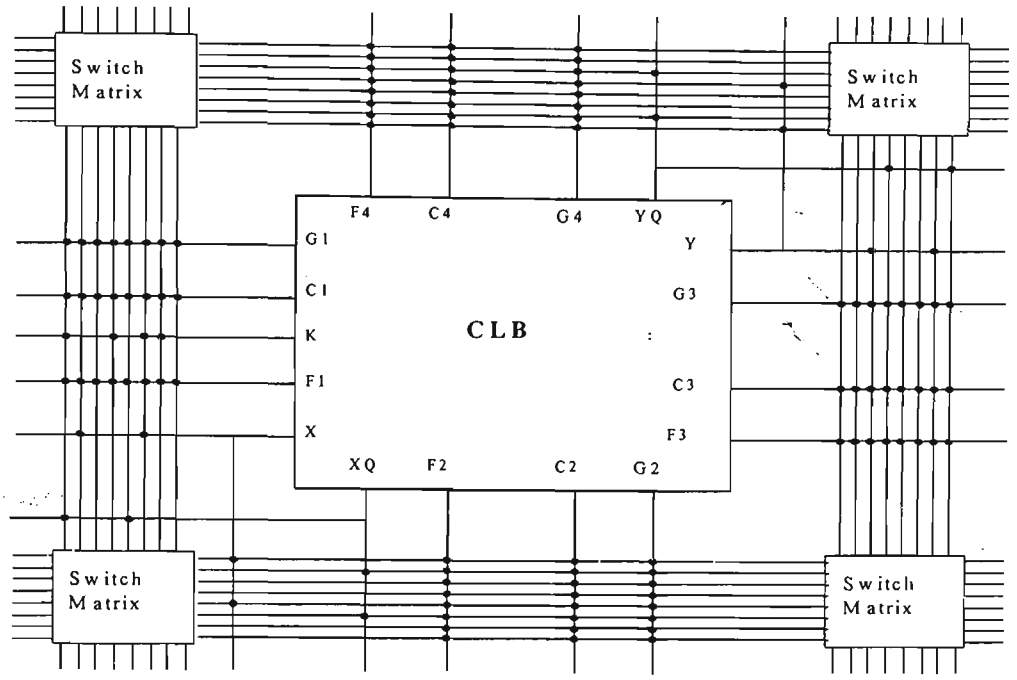


Figure D.4: CLB connections to single-length lines.

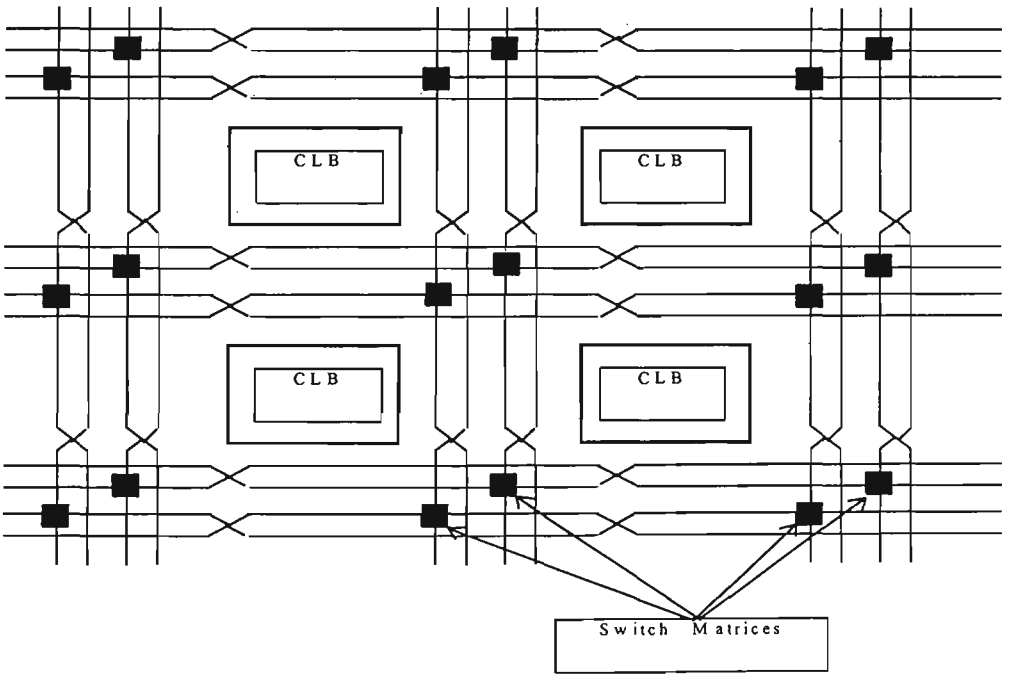


Figure D.5: Double-length lines.

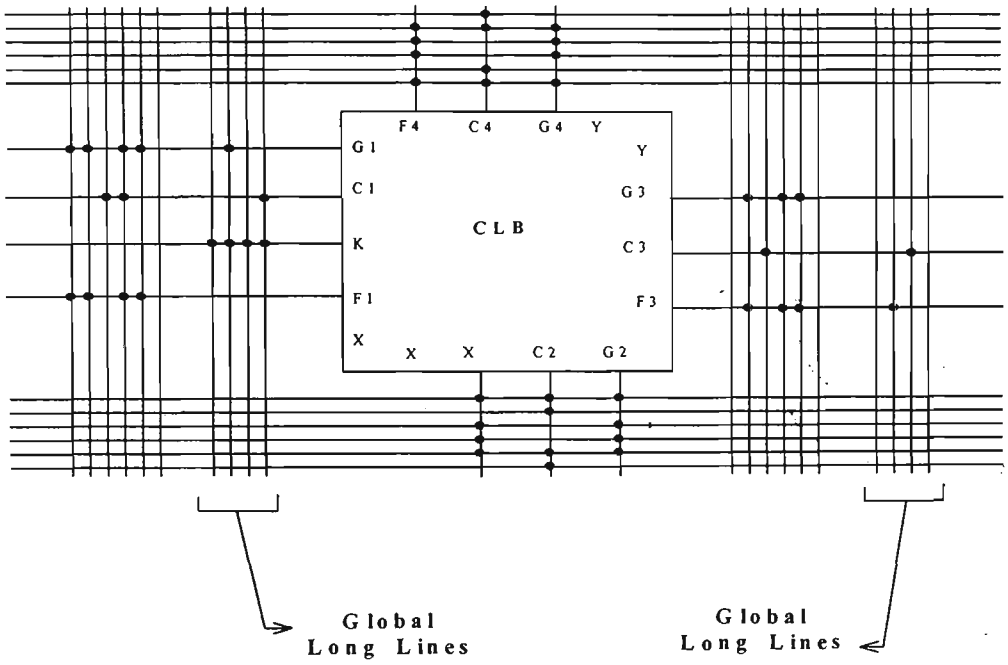


Figure D.6: Longlines with CLB connections.

D.2 Comparing FPGAs with other technologies

Recently, field-programmable gate arrays have been used widely due to several advantages related to their high gate count, short design cycle, and low prototyping cost. They can be used in all digital applications that currently use Small-Scale Integration (SSI), Medium-Scale Integration (MSI), and PLDs. They also replace mask-programmable gate arrays in many applications that are limited to 10,000 gates and where a high operational speed is not required.

Among the advantages of FPGAs are:

- replacement of SSI and MSI chips,
- availability of parts off the shelf,
- rapid turnaround,

• low risk,

• reprogrammability.

Compared with SSI and MSI chips, FPGAs offer larger gate counts and more design flexibility. If we consider a circuit constructed solely of NAND gates and its gate count is 1,000, we need about 250 Transistor-Transistor Logic (TTL) 7400 SSI chips to build it. The same circuit can, however, be replaced by one Xilinx chip (XC2064 or XC3020) [Xilinx, 1994]. FPGAs are more flexible because the logic does not have to be mapped in terms of standard SSI chips. Wire wrapping and soldering are also not required for a single-chip design, thus making it easier to realise appropriate engineering changes. While MSI chips have specific functions to which the design has to be mapped, FPGAs allow any random logic.

PLDs, the precursors of FPGAs, have actually been used to replace SSI in fixed logic, however, they did not help much in prototyping. In addition, PLDs implement logic in AND/OR gates and all the flip-flops are at the periphery of the devices. This type of logic arrangement restricts the designer and minimises the flexibility.

FPGAs combine the versatility of gate arrays and the programmability of PLDs. Unlike gate arrays, they do not require custom fabrication and are obtained off the shelf as are SSI and MSI chips. Because FPGAs are field programmable, they are definitely more suitable for prototyping than SSI and MSI chips. For example, Xilinx devices have been used in prototyping Intel's P5 microprocessor [Intel, 1992]. The hardware emulation made it possible to simulate the microprocessor at reasonable speed. Thus, the development time of the microprocessor was reduced. Reprogrammable FPGAs also allow the design to be altered and the chip to be reconfigured quickly. This ease of reprogrammability will facilitate design changes.

However, FPGAs have their limitations. For the same design implemented with FPGAs and PLDs, it is more likely that the PLDs will operate faster than the FPGAs. PLD performance is independent of the logic implemented. But for FPGAs, the circuit delay depends on the performance of the design implementation tools. The delay parameters can be extracted after placement and routing, typically a time-consuming process. Also, the mapping of the logic design into the FPGA's architecture requires more sophisticated design implementation (CAD) tools than PLDs. Compared with traditional gate arrays, FPGAs are less dense and operate at a lower speed. However, the rapid advance in FPGA technology is quickly closing the gap between the two realisations. The next consideration is cost.

Cost is an important factor favouring FPGAs. ASIC costs consists of fixed and variable components. Fixed costs include the initial cost needed to prepare masks, buy design tools, etc. Fixed costs for FPGAs include the development system and the platform which can vary from a personal computer to a sophisticated workstation. The development system includes the CAD tools (for design entry, simulation, and implementation) and a device programmer. The variable costs for FPGAs include the component costs which vary according to the number of logic blocks.

Manufacturing test costs for gate arrays includes of the costs of testing and packaging the individual chips. These costs for FPGAs are greatly reduced because test generation is done once for the unprogrammed chip. It is left up to the user to test the programmed chip in the field. At present, gate densities for FPGAs are on average, lower than those of MPGAs. As the chip density is constantly increasing, FPGAs are becoming more competitive with gate arrays.

Bibliography

Abramson D.A., (1992), "*A Very High Speed Architecture to Support Simulated Annealing*", IEEE Computer, Vol. 25, No. 5, pp. 27-38.

Abramson D.A., de Silva A., Randall M. and Postula A., (1995), "*Special Purpose Computer Architectures for High Speed Optimisation*", Proceedings of the Parallel and Real Time Computing Conference (PART-95), Perth, September 1995.

Ackley D.H., (1987), "Stochastic Iterated Genetic Hillclimbing", Ph.D. Thesis, Computer Science Department, Carnegie Mellon University, Pittsburge, PA.

Actel Co., (1991), "*Actel Family Field Programmable Gate Array Databook*", Actel Corporation, Santa Clara, CA.

Athanas P.M. and Silverman H.F., (1993), "*Processor Reconfiguration through Instruction-Set Metamorphosis*", IEEE Computer, Vol. 26, No. 3, pp. 11-18.

Athanas P.M., (1992), "*An Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*", Ph.D. Thesis, Brown University, Providence, Rhode Island, USA.

Baeck T., (1992), "*A User's Guide to Genesys 1.0*", Software Package Documentation, Computer Science Department, University of Dortmond, Germany.

Baker J.E., editor, (1985), "*Adaptive Selection Methods for Genetic Algorithms*", Proceedings of an International Conference on Genetic Algorithms, Lawrence Earlbaum, Hillsdale, NJ.

Belew R.K. and Booker L.B., editors, (1991), "*Proceedings of the Fourth International Conference on Genetic Algorithms*", Morgan Kaufmann, San Mateo, CA.

Bertin P., Roncin D. and Vuillemin J., (1993), "*Programmable Active Memories: A Performance Assessment*", PRL Research Report, No. 24, Technical Report, Digital Equipment Corporation, Paris Research Laboratory, Cedex, France.

Bethke A.D., (1981), "*Genetic Algorithms as Function Optimisers*", Ph.D. Thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI.

Bramlette M.B. and Bouchard E.E., (1991), *"Genetic Algorithms in Parametric Design of Aircraft"*, Chapter 10, pp. 109-123 of: Davis L., editor, "*Handbook of Genetic Algorithms"*, Van Nostrand Reinhold, New York.

Caruana R.A. and Shaffer J.D., (1988), "*Representation and Hidden Bias: Gray vs Binary Coding for Genetic Algorithms*", Proceedings of the 5th International Conference on Machine Learning, Morgan Kaufmann, Los Altos CA, pp. 153-161.

Casselman S., (1993), "*Virtual Computing and the Virtual Computer"*, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Los Alamitos, CA, pp. 43-48.

Chen R.J., Meyer R.R. and Yackel J., (1993), "*A Genetic Algorithm for Diversity Minimization and its Parallel Implementation*", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 163-170.

Choudhury B.H. and Rahman S., (1990), "*A Review of Recent Advances in Economic Dispatch*", IEEE Traansaction on Power systems, Vol. 5, No. 4, pp. 1248-1259.

Coelho D.R., (1989), "*The VHDL Handbook*", Kluwer Academic Publishers, Boston, MA.

Cohen A.I. and Sherkat V.R., (1987), "*Optimization Based Methods for Operations scheduling*", Proceedings of IEEE, Vol. 75, No. 12, pp. 1574-91.

Darwen P. and Yao X., (1995), "*A Dilemma for Fitness Sharing with a Scaling Function*", Proceedings of The Second IEEE International Conference on Evolutionary Computing (ICEC'95), The University of Western Australia, Perth, Australia, December 1995, pp. 166-171.

Data I/O, (1983), "*Programmable Logic: A Basic Guide for Designers*", Data I/O Corporation, Santa Clara, CA.

Davidor Y., (1991), "*A Genetic Algorithm Applied to Robot Trajectory Generation*" Chapter 12, pp. 144-165 of: Davis L., editor, "*Handbook of Genetic Algorithms*". Van Nostrand Reinhold, New York.

Davis L. and Coombs S., (1987), "*Genetic Algorithms and Communication Link Speed Design: Theoretical Considerations*", Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Earlbaum, Hillsdale NJ, pp. 252-256.

Davis L., (1989), "*Adapting Operator Probabilities in Genetic Algorithms*", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, Hillsdale, NJ, pp. 61-69.

Davis L., (1991a), "*Bit Climbing, Representational Bias and Test Suite Design*", Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 18-23.

Davis L., editor, (1987), "*Genetic Algorithms and Simulated Annealing*", Pitman Press, London.

Davis L., editor, (1991b), "*Handbook of Genetic Algorithms*", Van Nostrand Reinhold, New York.

DeJong K.A., (1975), "*An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*", Ph.D. Thesis, Dissertation Abstracts International 36(10), 5140B University of Michigan, Ann Arbor, MI.

Dorf R.C., (1991), "*Modern Control Systems*", Addison-Wesley Publishing, Reading, MA, 6th Edition.

Dorne R. and Hao J.K., (1995), "*An Evolutionary Approach for Frequency Assignment in Cellular Radio Networks*", Proceedings of The IEEE International Conference on Evolutionary Computing (ICEC'95), The University of Western Australia, Perth, Australia, December 1995, pp. 539-545.

El-Hawary M.E. and Christensen G.S., (1979), "*Optimal Economic Operation of Electric Power Systems*", Academic Press, New York.

Eldredge J.G. and Hutchings B.L., (1993), "*Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration*", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Los Alamitos, CA, pp. 180-188.

Eshlman L.J., editor, (1995), "*Proceeding of the Sixth International Conference on Genetic Algorithms*", Morgan Kaufmann, San Fransisco, CA.

Etter D.M. and Masukawa M.M., (1981), "*A Comparison of Algorithms for Adaptive Estimations of the Time Delay Between Sampled Signals*", Proceedings of ICASSP81:

IEEE International Conference on Acoustics, Speech and Signal Processing, The Institute Press, New York, pp. 1253-1256.

Fang H.L., Ross P. and Corne D., (1993), "*A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling and Open-Shop Scheduling Problems*", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 375-382.

Fogel D.B. and Atmar J.W., (1990), "*Comparing Genetic Operators with Gaussian Mutations in Simulated Evolutionary Processes Using Linear Systems*", Biological Cybernetics, 63, No. 2, pp. 111-114.

Fogel D.B., (1995), "*Evolutionary Coimputing*", IEEE Press, New York, NY.

Forrest S., editor, (1993), "*Proceedings of the Fifth International Conference on Genetic Algorithms*", Morgan Kaufmann, San Mateo, CA.

Gage P. and Kroo I., (1995), "*Representation Issues for Design Topological Optimization by Genetic Methods*", Proceedings of The Eighth International Conference on Industrial Application of Artificial Intelligence & Expert Systems (IEA95AIE), Melbourne, Australia, June 1995, pp. 383-388.

Gilson K.L., (1993), "*The nano Processor: A Low Resource Reconfigurable Processor*", Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society Press, Los Alamitos, CA, pp. 23-30.

Gokhale M., Holmes W., Kosper A., Lucas S., Minnich R., Sweely D. and Lopresti D., (1991), "*Building and Using a Highly Parallel Programmable Logic Array*", IEEE Computer, Vol. 24, No. 1, pp. 81-89.

Goldberg D.E. and Deb K., (1991), "*A Comparative Analysis of Selection Schemes Used in Genetic Algorithms*" In: Rawlins G.J.E., editor, "*Foundations of Genetic Algorithms*", Morgan Kaufmann, San Mateo, CA.

Goldberg D.E. and Smith R.E., (1987), "*Nonstationary Function Optimization Using Genetic Algorithms with Dominance and Diploidy*", Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Earlbaum, Hillsdale, NJ, pp. 59-68.

Goldberg D.E., (1987), "*Simple Genetic Algorithms and the Minimal Deceptive Problem*", pp. 74-88 of: Davis L., editor, "*Genetic Algorithms and Simulated Annealing*", Pitman, London.

Goldberg D.E., (1989a), "*Genetic Algorithms and Walsh Functions: Part 2, Deception and its Analysis*", Complex Systems, Vol. 3, pp. 129-152.

Goldberg D.E., (1989b), "*Genetic Algorithms in Search Optimization and Machine Learning*" Addison-Wesley, Reading, MA.

Goldberg D.E., (1990), "*Real-coded Genetic Algorithms, Virtual Alphabets and Blocking*" Technical Report IlliGAL 90001, The Illinois Genetic Algorithms Laboratory, IL.

Grefenstette J.J. and Schraudolph N., (1992), "*GENESIS 1.4ucsd. GA Software*", Software Documents, Naval Research Laboratory, available by anonymous ftp from (cs.ucsd.edu/pub/GACUSD).

Grefenstette J.J., (1986), "*Optimization of Control Parameters for Genetic Algorithms*" IEEE Transactions on Systems, Man and Cybenetics, SMC16(1), pp. 122-128.

Grefenstette J.J., (1990), "*A User Guide to Genesis 5.0.*", Software Documents, available from (http://www.aic.nrl.navy.mil/galist/src).

Grefenstette J.J., editor, (1985), "*Proceedings of an International Conference on Genetic Algorithms and Their Applications*", Lawrence Erlbaum Associates, Hillsdale, NJ.

Grefenstette J.J., editor, (1987), "*Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*", Lawrence Erlbaum Associates, Hillsdale, NJ.

Gruau F., (1993), "*Genetic Synthesis of Modular Neural Networks*", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 318-325.

Hancock P.J.B., (1989), *"Pruning Neural Nets by Genetic Algorithm"*, Proceedings of the International Conference of Artificial Neural Networks 1989 (ICANN-89), Elsevier, Amsterdam, Netherlands, pp. 991-4.

Happ H.H., (1977), "*Optimal Power Dispatch - A Comprehensive Survey*", IEEE Transaction on Power Apparatus and Systems, Vol. PAS-96, No. 3, pp. 841-54.

Herdy M., (1991), "*Application of Evolution strategies to Discrete Optimization Problems*", pp. 188-192 of: Schwefel H.P., Manner R., editors, "*Parallel Problem Solving from Nature*", Springer Verlag, Berlin.

Higuchi T., Iba H. and Manderick B., (1994), "*Applying Evolvable Hardware to Autonomous Agents* " Proceedings of the Third Conference on Parallel Problem Solving from Nature (PPSN III), Spering-Verlag, Berlin, Germany.

Holland J., (1975), "*Adaptation in Natural and Artificial Systems*", MIT Press, Cambridge, MA.

Hollis E.E., (1987), "*Design of VLSI Gate Arrays ICs*", Prentice Hall, Englewood Cllifs, NJ.

Hollstien R.B., (1971), "*Artificial Genetic Adaptation in Computer Control Systems*". Ph.D. Thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI.

Homaifar A., Guan S. and Liepins G.E., (1993), "*A New Approach on the Travelling Salesman Problem by Genetic Algorithms*", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 460-466.

Hunter A., (1995), "*SUGAL User Manual V2.0 : Software Package Documentation*", University of Sunderland, UK.

Husband P. and Mill F., (1991), "*Simulated Co-Evolution and the Mechanism for Emerging Planning and Scheduling*", Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 264-270.

Hwang W.R. and Thompson W.E., (1993), "*An Intelligent Controller Design Based on Genetic Algorithms*", Proceedings of the 32nd Conference on Decision and Control, San Antonio, Texas, pp. 1266-7.

Johnston C.R. Jr. and Larimore M.G., (1977), "*Comments on and Addition to "An Adaptive Recursive LMS Filter*"", Proceedings of IEEE, 65, (9), pp. 1399-1401.

Jullif K., (1993), "*A Multi-Chromosome Genetic Algorithm for Pallet Loading*", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 467-473.

Jun C., Fogarty T.C. and Gammack j.G., (1993), "*Searching Databases Using Parallel Genetic Algorithms on a Transputer Computing Surface*", Future Generation Computer Systems, Vol. 9, No. 1, pp. 33-40.

Kacprzyk J., (1995), "*Multistage Control of a Fuzzy System Using a Genetic Algorithm*", Proceedings of The Second IEEE International Conference on Evolutionary

Computing (ICEC'95), The University of Western Australia, Perth, Australia, December 1995, pp. 842-845.

Kenyon P., Seth S., Agrawal P., Clematis A., Dodero G., Gianuzzi V. and Agraval P., (1992), "*Programming Pipelined CAD Applications on Message Passing Architectures*", Proceedings of EWPC92, The European Workshop on Parallel Computing, IOS Press, Amsterdam, Netherlands, pp. 550-3.

Kirkpatrick S., Gelatt C.D. and Vecci M.P., (1983), "*Optimization by Simulated Annealing*", Science, Vol. 220, May 1983, pp. 671-680.

Kunz D., (1991), "*Channel Assignment for Cellular Radio Using Neural Networks*", IEEE Transaction on Vehicular Technology, Vol. 40, pp. 188-193.

Louis S.J. and Murray A., (1995), "*Adapting Control Strategies for Situated Autonomous Agents*", Proceedings of the Eight Florida artificial Intelligence Research Symposium (FLAIRS-95), Melbourne Beach, Florida, April 1995, pp. 274-278.

Lucasius C.B. and Kateman G., (1989), "*Application of Genetic Algorithm in Chemometrics*", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, pp. 170-176.

Manner R. and Manderick B., editors, (1992), "*Parallel Problem Solving from Nature, 2*", Amsterdam, North-Holland.

McLeod J., (1994), "*Reconfigurable Computer May Reconfigure Market*", Electronics, Vol. 67, No. 8, page 5.

Mohammadian M. and Stonier R.J., (1994), "*Generating Fuzzy Rules by Genetic Algorithms*", Proceedings of the Third International Workshop on Robot and Human Communication, Nagoya, Japan, pp. 362-367.

Mohammadian M. and Stonier R.J., (1995), "*Adaptive Two Layer Fuzzy Control of a Mobile Robot System*", Proceedings of The Second IEEE International Conference on Evolutionary Computing (ICEC'95), The University of Western Australia, Perth, Australia, December 1995, pp. 204-208.

Monolithic Memories, (1986), "*PAL/PLE Device Programmable Logic Array Handbook*", Monolithic Memories, Santa Clara, CA.

Montana D.J. and Davis L., (1989), "*Training Feedforward Neural Networks Using Genetic Algorithms*", Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI89), Morgan Kaufmann, Palo Alto, CA, pp. 762-767.

Nakano K., Hiraki H. and Ikeda S., (1995), "*A Learning Machine that Evolves*", Proceedings of The Second IEEE International Conference on Evolutionary Computing (ICEC'95), The University of Western Australia, Perth, Australia, December 1995, pp. 808-813.

Narendra K.S. and Thathachar M.A.L., (1989), "*Learning Automata - An Introduction*", Prentice Hall, Englewood Cliffs, NJ.

Newell S.B., de Geus A.J. and Roher R.A., (1983), "*Design Automation for Integrated Circuits*", Science, Vol. 220, No. 4596, pp. 465-474.

Ogata K., (1990), "*Modern Control Engineering*", 2nd Edition, Prentice-Hall, Englewood Cliffs, NJ.

Paraskevopoulos P.N., (1988), "*On the Design of PID Controller for Linear Multivariable Systems*", IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. IECI-27, No. 1, , pp. 19-27.

Petrovic R. and Kralj B., (1993), "*Economic and Environmental Power Dispatch*", European Journal of Operational Research, Vol. 64, No. 1, pp. 2-11.

Radcliffe N., (1990), "*Genetic Neural Networks on MIMD Computers*", Ph.D. Thesis, Department of Computer and Science, The University of Edinburgh, Edinburgh, UK.

Rawlins G.J.E., (1991), *"Foundation of Genetic Algorithms"*, Morgan Kaufmann, San Mateo, CA.

Rayfield J.T. and Silverman H.F., (1988), "*System and Application Software for the Armstrong Multiprocessor*", IEEE Computer, 21(6), pp. 38-52.

Reynolds R.G., (1995), "*Solving Design Problems Using Cultural Algorithms*", Proceedings of the Eight Florida artificial Intelligence Research Symposium (FLAIRS-95), Melbourne Beach, Florida, April 1995, pp. 279-283.

Richardson J.T., Palmer M.R., Liepens G. and Hilliard M., (1989), "*Some guidelines for Genetic Algorithms with Penalty Functions*", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 191-197.

Roberts R.A. and Mullis C.T., (1987), "*Digital Signal Processing*", Addison-Wesley, Reading, MA.

Sasson A.M. and Merrill H.M., (1974), *"Some Application of Optimization Techniques to Power System Problems*", Proceedings of the IEEE, Vol. 2 , No. 7, pp. 959-971.

Schaffer J.D., editor, (1989), "*Proceedings of the Third International Conference on Genetic Algorithms"*, Morgan Kaufmann, San Mateo, CA.

Schraudolph N.N. and Belew R.K., (1992), "*Dynamic Parameter Encoding for Genetic Algorithms"*, Machine Learning, Vol. 9, No. 1, pp. 9-22.

Schwefel H.P. and Manner R., editors, (1991), "*Parallel Problem Solving from Nature*", Springer-Verlag, Berlin, Germany.

Schwefel H.P., (1981), "*Numerical Optimization of Computer Models*", Wiley, Chichester, New York.

Serra M., Slater T., Muzio J.C. and Miller D.M., (1990), "*The Analysis of One-Dimensional Linear Cellular Automata and their Aliasing Properties*", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 9, No. 7, pp. 767-778.

Shaefer C.G., (1987), "*The ARGOT Strategy: Adaptive Representation Genetic Optimizer Technique*", Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Earlbaum, Hillsdale NJ, pp. 50-58.

Shapiro I.J. and Narendra K.S., (1969), "*Use of Stochastic Automata for parameter self-Optimisation with Multimodal Performance Criteria*", IEEE Trans., SSC-5, (4), pp. 352-360.

Song Y.H., Li F., Morgan R. and Williams D., (1995), "*Environmentally Constrained Electric Power Dispatch with Genetic Algorithms*", Proceedings of The Second IEEE International Conference on Evolutionary Computing (ICEC'95), The University of Western Australia, Perth, Australia, December 1995, pp. 17-20.

Spears W.M., (1989), "*Using Neural Networks and Genetic Algorithms as Heuristics for NP-Complete Problems*", M.S. Thesis, George Mason University.

Spiessens P. and Manderick B., (1991), "*A Massively Parallel Genetic Algorithm Implementation and First Analysis*", Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 279-285.

Syswerda G., (1989), "*Uniform Crossover in Genetic Algorithms*", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, pp. 2-9.

Tanese R., (1987), "*Parallel Genetic Algorithm for a hypercube*", Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Earlbaum, Hillsdale, NJ, pp. 177-183.

Tanese R., (1989), "*Distributed Genetic Algorithms*", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, pp. 434-439.

Tang C.K.K. and Mars P., (1989), "*Intelligent Learning Algorithms for Adaptive Digital Filters*", Electronics Letters, 25, (23), pp. 1565-1566.

Tang C.K.K. and Mars P., (1991), "*Stochastic Learning Automata and Adaptive IIR Filters*", IEE Proceedings-F, Vol. 138, No. 4, August, pp. 331-340.

Thangiah S.R., Vinayagamoorthy R. and Gubbi A.V., (1993), "*Vehicle Routing with Time Deadlines using Genetic and Local Algorithms*", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 506-513.

Vose M.D., (1991), "*Generalizing the Notion of Schema in Genetic Algorithms*", Artificial Intelligence, Vol. 50, No. 3, pp. 385-96.

Walker D.A. and Potter W.D., (1995), "*A Genetics-Based Learning Approach to 3-D Pursuer/Evader Strategies*", Proceedings of the Eight Florida artificial Intelligence Research Symposium (FLAIRS-95), Melbourne Beach, Florida, April 1995, pp. 284-288.

Walters D.C. and Sheble Z.C., (1993), "*Genetic Algorithm Solution of Economic Dispatch With Valve Point Loading*", IEEE Transaction on Power Systems, Vol. 8, No. 3, pp. 1325-32.

Weste N.H.E. and Eshraghian K., (1993), "*Principles of CMOS VLSI Design: A Systems Perspective*", Addison-Wesley Publishing Company, Incorporated Reading, MA, 2nd edition.

Whitley D. and Kauth J., (1988), "*GENITOR: a Different Genetic Algorithm*", Proceedings of the Rocky Mountain Conference on Artificial Intelligence, Denver, Colorado, pp. 118-130.

Whitley D., (1989), "*The Genitor Algorithm and Selection Pressure: Why Rank-Based Allocation of Trials is Best*", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo CA, pp. 116-121.

Whitley D., (1991), "*Fundamental Principles of Deception in Genetic Search*", pp. 221-241 of: Rawlins G.J.E., editor, "*Foundations of Genetic Algorithms*", Morgan Kaufmann, San Mateo, CA.

Whitley D., Starkweather T. and Bogart C., (1990), "*Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity*", Parallel Computing, 14-3, pp. 347-361.

Widrow B. and Stearns S.D., (1985), "*Adaptive Signal Processing*", Prentice-Hall Inc., Englewood Cliffs, NJ.

Willsky A.S., (1979), "*Digital Signal Processing and Control and Estimation Theory*", The MIT Press, Cambridge, Massachusetts.

Willsky A.S., (1985), "*Adaptive Signal Processing*", Prentice-Hall Inc., Englewood Cliffs. NJ.

Wirbel L., (1992), "*Compression Chip is First to Use Genetic Algorithms*", Electronic Engineering Times, page 17, December 1992.

Wolfram S., (1984), "*University and Complexity in Cellular Automata*" P*hysica,* 10D, pp. 1-35.

Xilinx Inc., (1994), "*The Programmable Logic Data Book"* Xilinx Incorporated, San Jose, CA.

Yao X., (1993), "*An Empirical Study of Genetic Operators in Genetic Algorithms*", Microprocessing and Microprogramming, 38(1-5), pp. 707-714.