# Genetic Programming in Hardware

Peter N. Martin

A Thesis submitted for the degree of Ph.D

Department of Computer Science

University of Essex

Spring, 2003

## Dedication

*To Lynette and Nicole*

## ABSTRACT

### Genetic Programming in Hardware

This thesis describes a hardware implementation of a complete *Genetic Programming* (GP) system using a *Field Programmable Gate Array*, which is shown to speed-up GP by over 400 times when compared with a software implementation of the same algorithm. The hardware implements the creation of the initial population, breeding operators, parallel fitness evaluations and the output of the final result.

The research was motivated by the observation that GP is usually implemented in software and run on general purpose computers. Although software implementations are flexible and easy to modify, they limit the performance of GP thus restricting the range of problems that GP can solve. The hypothesis is that implementing GP in hardware would speed up GP, allowing it to tackle problems which are currently too hard for software based GP.

FPGAs are usually programmed using specialised hardware design languages. An alternative approach is used in this work that uses a high level language to hardware compilation system, called Handel-C.

As part of this research, a number of general GP issues are also explored. The parameters of GP are described and arranged into a *taxonomy of GP attributes*. The taxonomy allows GP problems to be categorised with respect to their problem and GP specific attributes. The role that the GP algorithm plays in problem solving is shown to be part of a larger process called *Meta-GP*, which describes the overall process of developing a GP system and evolving a viable set of parameters to allow GP to solve a problem. Three crossover operators are investigated and a new operator, called *single child limiting* crossover, is presented. This operator appears to limit the tendency of GP to suffer from bloat. The economics of implementing GP in hardware are analysed and the costs and benefits are quantified. The thesis concludes by suggesting some applications for hardware GP.

## Acknowledgments

First of all, I would like to express my heartfelt thanks to my supervisor Professor Riccardo Poli for agreeing to supervise me and for all the help and guidance he gave me throughout the work. I would like to thank him especially for always answering my questions promptly and for reviewing and commenting on many drafts and papers, often at short notice.

A special note of thanks goes to Dr. Stuart Wray, a former colleague from Marconi, for help and encouragement, for thoroughly reviewing many papers and drafts and for many thought provoking discussions about this work. Stuart also planted the idea of Meta-GP which is described in Chapter 2. I would also like to thank Icarus Sparry, again a former colleague from Marconi, for his valuable comments.

This work was sponsored by Marconi plc whom I would like to thank for their support and for generously allowing me time to undertake this work. In particular, I would like to thank Stuart Barratt who initially introduced Handel-C to me and made it possible for me to have access to Handel-C software while at Marconi. He also lent me the RC1000 development board which was used for the experimental work. I would also like to thank Steffan Westcott who provided useful guidance in the use of Handel-C and for some perceptive comments and suggestions on early drafts of my first paper.

Celoxica have been generous in their support of this work, in particular Graham McKenzie and Roger Gook for making a copy of Handel-C available to me after I had left Marconi. Xilinx Inc. supplied the Alliance Tool set which was used for place and route.

I would also like to thank the GP community for their help, especially the anonymous reviewers of papers I have submitted who have made constructive comments and who have helped me to improve my published papers. I would also like to thank Bill Langdon for maintaining that most valuable resource – the GP bibliography.

Finally, I would like to thank my wife Lynette and daughter Nicole for their forbearance during the past 3 years while I carried out this work. Without their understanding and encouragement, and for not complaining when I disappeared into the study to spend many hours at my computers, it would have been impossible to complete this thesis. I would also like to thank Lynette for proof reading several papers and parts of this thesis.

## Trademarks

# Contents

**Dedication** ii

**Abstract** iii

**Acknowledgments** iv

**List of Tables** vii

**List of Figures** viii

**Acronyms and Terms** x

**1 Introduction** 1
- 1.1 Contributions 2
- 1.2 Organisation 3

**2 Genetic Programming: A Survey and Taxonomy** 6
- 2.1 Genetic Programming - A Historical Perspective 6
- 2.2 Principles of Genetic Programming 9
- 2.3 Attributes of Genetic Programming 14
- 2.4 A Taxonomy of Genetic Programming Attributes 17
- 2.5 A Summary of the Taxonomy 31
- 2.6 Results and Discussion 34
- 2.7 Summary 40

**3 FPGAs and Hardware Compilation** 42
- 3.1 Introduction 42
- 3.2 Field Programmable Gate Arrays 44
- 3.3 FPGAs and Evolutionary Computing 47
- 3.4 Configuring FPGAs and Hardware Compilation 51
- 3.5 Handel-C 52
- 3.6 Summary 57

**4 Implementing GP in Hardware** 59
- 4.1 A Complete GP System On a Chip 59
- 4.2 Parallelism 62
- 4.3 Random Number Generator. 63
- 4.4 Initial Program Creation 65
- 4.5 Breeding Policy and Operators 65
- 4.6 Performance Comparison Methodology 67

4.7 Experimental Setup . . . . . 68
4.8 Experiment Descriptions and Results . . . . . 71
4.9 Discussion . . . . . 79
4.10 Summary . . . . . 82

**5 An Optimised Implementation 84**
5.1 System Architecture . . . . . 84
5.2 Experimental Setup . . . . . 92
5.3 Experiment Descriptions and Results . . . . . 94
5.4 Discussion . . . . . 102
5.5 Summary . . . . . 104

**6 Behavioural Analysis 105**
6.1 Analysis of the Crossover Operator . . . . . 105
6.2 Discussion . . . . . 120
6.3 Analysis of the Random Number Generator . . . . . 122
6.4 Random Number Generator Implementations . . . . . 125
6.5 Experimental Results . . . . . 130
6.6 Discussion . . . . . 131
6.7 Summary . . . . . 132

**7 Economic Analysis of Hardware GP 134**
7.1 Hardware and Software Costs . . . . . 134
7.2 Software Engineering Issues . . . . . 135
7.3 Genetic Programming Costs . . . . . 137
7.4 Quantifying the Costs . . . . . 138
7.5 Summary . . . . . 142

**8 Conclusions 144**
8.1 Recapitulation of the Contributions . . . . . 144
8.2 Further Work . . . . . 149
8.3 Conclusions About the Hypothesis . . . . . 152

**Bibliography 154**

**A Attributes of the Experimental Problems 185**

**B Taxonomy Data 190**
B.1 Data Used to Construct the Taxonomy of Genetic Programming . . . . . 190

**C Experimental Setup 197**
C.1 Experimental Platforms . . . . . 197
C.2 Software Tools . . . . . 198
C.3 Other Tools . . . . . 200

**D Random Number Generator Test Results 201**

# List of Tables

2.1 Langdon’s taxonomy of GP. . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 GP applications suggested by Koza. . . . . . . . . . . . . . . . . . . . . . . . . 16
2.3 Application areas for GP from Banzhaf *et al.* . . . . . . . . . . . . . . . . . . . . 16
2.4 GP applications suggested by Langdon. . . . . . . . . . . . . . . . . . . . . . . 17
2.5 Popular software implementations of GP systems . . . . . . . . . . . . . . . . . . 29
2.6 A uniform taxonomy of parallel genetic algorithms . . . . . . . . . . . . . . . . . 31
2.7 Summary of the attributes of GP and their characteristics. . . . . . . . . . . . . . . 32

4.1 Parameters for the regression problem . . . . . . . . . . . . . . . . . . . . . . . 74
4.2 Results of running the regression problem . . . . . . . . . . . . . . . . . . . . . 74
4.3 Parameters for the XOR problem . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.4 Results of running the XOR problem . . . . . . . . . . . . . . . . . . . . . . . . 77
4.5 Cycle counts and gate estimates for various stages of the GP. . . . . . . . . . . . . 78

5.1 Possible population sizes when using three 2 MiB memory banks. . . . . . . . . . 85
5.2 Results of running the XOR problem. . . . . . . . . . . . . . . . . . . . . . . . . 94
5.3 Parameters for the artificial ant problem . . . . . . . . . . . . . . . . . . . . . . 97
5.4 Results of running the artificial ant problem . . . . . . . . . . . . . . . . . . . . 97
5.5 Comparison of the performance of the artificial ant running in hardware to the artificial ant running on a 1.4 GHz processor, using both the hardware algorithm and lilgp. The hardware implementation used 32 parallel fitness evaluations. . . . . . . 100
5.6 Parameters for the Boolean even-6-parity problem . . . . . . . . . . . . . . . . . 101
5.7 Results of running the Boolean even-6-parity problem . . . . . . . . . . . . . . . 101

6.1 Summary results of running the Diehard tests on the random number generators. . . 130
6.2 Summary of GP performance for all random number generators tested. . . . . . . . 131

7.1 Costs for implementing hardware GP and software GP. . . . . . . . . . . . . . . . 139

A.1 Simple regression attributes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
A.2 Exclusive OR problem attributes . . . . . . . . . . . . . . . . . . . . . . . . . . 187
A.3 Santa Fe Artificial Ant problem attributes . . . . . . . . . . . . . . . . . . . . . 188
A.4 Boolean even-6-parity problem attributes . . . . . . . . . . . . . . . . . . . . . 189

D.1 Diehard test results for all RNGs examined. . . . . . . . . . . . . . . . . . . . . 202

# List of Figures

2.1 The phylogeny of GP as a timeline. . . . . 8
2.2 Example tree for an evolving program in standard GP. . . . . 9
2.3 Flow chart of standard Genetic Programming algorithm. . . . . 11
2.4 Crossover operating on two trees. . . . . 14
2.5 A tree based taxonomy of the attributes of Genetic Programming . . . . . 33
2.6 Distribution of top level problem categories for 158 examples of GP. . . . . 34
2.7 Spread of program sizes found in 78 Genetic Programming examples . . . . . 35
2.8 Number of generations for 120 problems. . . . . 35
2.9 Population sizes for 125 problems. . . . . 35
2.10 Population sizes for the 101 problems where the size is less than 10 000. . . . . 36
2.11 Flow chart for the meta GP process. . . . . 38

3.1 General model of an FPGA. . . . . 44
3.2 General model of a Configurable Logic Block or Slice. . . . . 45
3.3 Consequences of illegal FPGA gate configuration. . . . . 46
3.4 Overview of the process of translating code into hardware using Handel-C. . . . . 56

4.1 Layout of an instruction. . . . . 61
4.2 *Linear Feedback Shift Register* (LFSR) random number generator. . . . . 64
4.3 Truncating crossover operator . . . . . 66
4.4 Block diagram of the Celoxica RC1000 FPGA board. . . . . 70
4.5 Photograph of the RC1000 development board. . . . . 70
4.6 Cycles per evaluation vs. parallel fitness evaluations. . . . . 78
4.7 Total cycles vs. parallel fitness evaluations. . . . . 78

5.1 Overall architecture of the pipelined GP system. . . . . 87
5.2 Illustration of the operation of the pipelined implementation. . . . . 89
5.3 Layout of the control data SRAM. . . . . 93
5.4 Santa Fe artificial ant trail. . . . . 96
5.5 Simple graphical simulator for the artificial ant. . . . . 98
5.6 Speedup for the artificial ant problem. . . . . 98
5.7 Speedup for the Boolean even-6-parity problem. . . . . 102

6.1 GP performance of the original design for the artificial ant problem. . . . . 106
6.2 Length distribution for standard GP with no fitness. . . . . 107
6.3 Length distribution using truncating crossover without fitness from hardware GP. . 108
6.4 Length distribution using truncating crossover for ant problem. . . . . 109
6.5 Length distribution using truncating crossover for parity problem. . . . . 109
6.6 Length distribution using dual child limiting crossover for ant problem. . . . . 110
6.7 Length distribution using dual child limiting crossover for ant problem. . . . . 111

6.8 Length distribution using dual child limiting crossover for parity problem. . . . . . 111
6.9 Length distribution using single child limiting crossover without fitness. . . . . . . 112
6.10 Length distribution using single child limiting crossover for ant problem. . . . . . . 113
6.11 Length distribution using single child limiting crossover for parity problem. . . . . 113
6.12 Comparative GP behaviour for ant problem. . . . . . . . . . . . . . . . . . . . . . . . 114
6.13 Lengths of programs using truncating crossover for ant problem. . . . . . . . . . . 115
6.14 Lengths of programs using dual child limiting crossover for ant problem. . . . . . . 115
6.15 Lengths of programs using single child crossover for ant problem. . . . . . . . . . 116
6.16 Lengths of programs using truncating crossover for parity problem. . . . . . . . . . 116
6.17 Lengths of programs using dual child limiting crossover for parity problem. . . . . 117
6.18 Lengths of programs using single child limiting crossover for parity problem. . . . 117
6.19 Lengths of programs for ant problem, max length = 16. . . . . . . . . . . . . . . . . 118
6.20 Lengths of programs for ant problem, max length = 8. . . . . . . . . . . . . . . . . . 119
6.21 Length distribution for ant problem, max length = 16. . . . . . . . . . . . . . . . . . 119
6.22 Length distribution for ant problem, max length = 8. . . . . . . . . . . . . . . . . . . 120
6.23 Serial test of a simple LFSR RNG. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
6.24 Serial test for an RNG using 16 parallel LFSRs . . . . . . . . . . . . . . . . . . . . . 127
6.25 Serial test for a 1DCA RNG . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
6.26 Serial test for a 32CA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

7.1 Cost benefit of using hardware GP for the Boolean even-6-parity problem. . . . . . 140
7.2 Cost benefit using hardware GP for a hypothetical problem. . . . . . . . . . . . . . . 140
7.3 Number of runs that can be achieved using the Boolean even-6-parity problem. . . 141
7.4 Number of runs that can be achieved using a hypothetical problem . . . . . . . . . 142

# Acronyms and Terms

This thesis contains a number of acronyms and terms, some of which have specialised meanings. For convenience, they are brought together in this section.

## Acronyms

**ADF** *Automatically Defined Function*

**ADM** *Automatically Defined Macro*

**API** *Application Programming Interface*
A documented programming interface between sets of functions.

**ASIC** *Application Specific Integrated Circuit*
A custom built integrated circuit.

**BNF** *Bakus Naur Form*

**BRAM** *Block Random Access Memory*
Memory that is integrated into a Xilinx FPGA.

**CA** *Cellular Automata*

**CC** *Carry and Control*

**CISC** *Complex Instruction Set Computer*

**CLB** *Configurable Logic Block*

**CMOS** *Complementary Metal Oxide Semiconductor*
A process for realising semiconductor devices.

**CPLD** *Complex Programmable Logic Device*

**CPU** *Central Processing Unit*

**CSP** *Communicating Sequential Processes*
A formalism devised by C.A.Hoare that describes how multiple processes can communicate and cooperate. See [Hoare 85].

**DDR** *Double Data Rate*
Memory technology that uses both the rising and falling edges of a clock to transfer data.

**EABI** *Embedded Application Binary Interface*
A calling convention for the PowerPC architecture. See [IBM and Motorola 95]

**EC** *Evolutionary Computation*

**EDIF** *Electronic Design Interchange Format*
A widely used electronic design interchange format. See [EDIF 02]

**EP** *Evolutionary Programming*

**ES** *Evolutionary Strategies*

**FF** *Flip Flop*

**FPGA** *Field Programmable Gate Array*

**FSM** *Finite State Machine*

**GA** *Genetic Algorithm*

**GAs** *Genetic Algorithms*

**GP** *Genetic Programming*
A means of creating computer programs by applying the principles of evolution.

**HDL** *Hardware Design Language*

**IOB** *Input Output Block*
Logic that interfaces the internal logic to the input/output pins of the device.

**LFSR** *Linear Feedback Shift Register*

**I/O** *Input/Output*

**LUT** *Look up Table*
A type of logic cell within an FPGA that implements a set of logic functions.

**PCI** *Peripheral Component Interconnect*
A high speed interconnection system between a microprocessor and attached devices. Designed by Intel, PCI is designed to be synchronized with the clock speed of the microprocessor.

**PLA** *Programmable Logic Array*

**PMC** *PCI Mezzanine Card*
Industry standard format cards.

**PRNG** *Pseudo Random Number Generator*

**RAM** *Random Access Memory*
Memory that can be randomly addressed and that can be read from and written to.

**RISC** *Reduced Instruction Set Computer*

**RNG** *Random Number Generator*

**SA** *Simulated Annealing*

**SIMD** *Single Instruction Multiple Data*
Instructions that operate on more than one operand at a time.

**ROM** *Read Only Memory*
Memory that allows data to be read, but not written.

**SRAM** *Static Random Access Memory*
Memory that uses a static design which removes the need for refresh circuity and refresh cycles.

**VHDL** *VHSIC Hardware Design Language*
A standard hardware design language

**VHSIC** *Very High Speed Integrated Circuit*

**VLSI** *Very Large Scale Integration*

## Unit Conventions

This thesis uses the IEC recommended prefixes for binary multiples. Using these recommendations KiB indicates $2^{10}$ bytes, MiB indicates $2^{20}$ bytes, and GiB indicates $2^{30}$ bytes. For more information see [NIST 02].

## Typographical Conventions

The following typographical conventions are used in this thesis:

`Typewriter style`
Is used to indicate a computer program listing, code fragment or a statement. Language specific keywords are printed using a **`bold`** typeface
***Slanted bold style***
Is used to indicate a command typed into an interactive shell.

# Chapter 1

# Introduction

It has long been a goal of computer scientists to automatically translate a problem statement directly into a computer program without having to explicitly describe how the program should be created. This has often been referred to as automatic programming. The mechanics of how it might be achieved were hinted at by Alan Turing in his essay “Computing Machinery and Intelligence” [Turing 50]. He suggested that a machine, seeded with random elements, could monitor the results it generated and subsequently modify itself in the light of those results. The idea of adopting automatic computer program generation is also of great interest to the general software engineering community. In the early 1980’s this idea was considered interesting but “somewhat beyond the current frontier of the state of the art” ( [Boehm 81, Chapter 33]).

Evolutionary techniques in general, and *Genetic Programming* (GP) in particular, have brought this goal closer to reality. Several researchers, for example [Friedberg 58, Friedberg 59, Fogel 66, Smith 80, Cramer 85, Hicklin 86, Fujiki 87, Schmidhuber 87], have used evolutionary techniques for this problem, with varying degrees of success, but it was not until the early 1990’s, when John Koza coined the term Genetic Programming [Koza 90b] [Koza 90a], that a robust evolutionary method for automatic program creation became available. Since then, Genetic Programming has grown into a distinct research field with several major conferences and over 2000 published papers, reports and books.

GP systems are generally realised as programs running on general purpose computers. Most of the earliest GP systems were implemented in LISP. More recently, GP systems have been implemented in C, C++, Java and other languages. However, even with modern processors running at over 2 GHz, and the use of massively parallel computers, using GP still requires large amounts of time and computer resources to tackle some of the harder problems.

This research was motivated by the observation that tackling some problems using GP is not practical because of the very long execution time required. One way of reducing the execution time of an algorithm is to implement some, or all, of the algorithm in hardware rather than software. This led to the hypothesis that implementing GP in hardware will reduce the time needed to run the GP algorithm. A reduced run time for GP would then allow the detailed operation of GP to be explored in ways that previously would have required an uneconomic investment in time and equipment. Reducing the running time of GP would also allow GP to be applied to problems that to date have been hard or impossible for GP to solve.

Reducing the running time of the GP algorithm is of interest to two different groups of GP users. The first group are the practitioners who are using GP for solving problems. If the run time of the GP algorithm can be reduced, then more difficult problems could be tackled using GP. Faster execution of the GP algorithm, in particular the fitness evaluation part of the algorithm, also opens up the possibility of using GP in real-time applications, for example signal processing. Secondly, reduced running time would be of benefit to the researcher who is investigating the theoretical operation of GP by, for example, allowing the dynamics of GP to be explored in ways that previously would have required an uneconomic investment in time and equipment.

## 1.1 Contributions

This thesis makes 3 main contributions:

1. It shows that a complete GP system can be designed and implemented in hardware. This is in contrast to all previously reported work involving hardware which implemented only parts of the GP algorithm in hardware, or that only demonstrated the idea using software simulations.

2. It explores the role that the GP algorithm plays in the larger process of problem solving. This is called *Meta-GP*.

3. It gives an empirical analysis of the behavioural aspects of the system, by considering the population dynamics. This shows that the length distribution of GP is important when choosing appropriate operational parameters. From this analysis, an alternative crossover operator

is presented. The new crossover operator – called *single child limiting crossover* – has the effect of reducing the impact of bloat for the two problems which were investigated.

This thesis also makes 4 secondary contributions:

1. A taxonomy of the attributes of GP is presented which allows a GP problem to be categorised with respect to its problem–specific and GP–specific attributes.

2. It shows how a high level language to hardware compilation system allows a software approach to the hardware design of GP.

3. A comparative analysis of different random number generators for hardware platforms is presented.

4. An economic analysis of implementing GP in hardware is given.

## 1.2 Organisation

Following this introductory chapter, Chapter 2 introduces GP and identifies the attributes of GP. A taxonomy of the attributes is then presented. From the data collected during the construction of the taxonomy of attributes, a number of niches can be identified which merit further research, in particular, implementing a complete GP platform in hardware. Chapter 2 also considers GP as part of a problem solving process, called Meta-GP. Because Meta-GP uses the GP algorithm repeatedly in the search for solutions to a problem, the importance of reducing the execution time of the GP algorithm is shown.

Chapter 3 reviews the main hardware and software technologies that are available to implement a GP system in hardware. In particular it looks at *Application Specific Integrated Circuit* (ASIC) and *Field Programmable Gate Array* (FPGA) technologies and reviews work that has used these devices in the general field of evolutionary computing. The languages and tools that are used to program these devices are then examined. Traditional approaches that use hardware design languages are compared to high level languages. Chapter 3 then describes a high level language to hardware compilation system called Handel-C that can be used to translate C-like programs into the configuration data for a FPGA. The features of Handel-C that were of importance to this work are highlighted, together with a comparative analysis of Handel-C and other high level language to hardware compilation systems.

Chapter 4 describes an implementation of GP in hardware that was designed to prove the general concept of using an FPGA for GP. This implementation is deliberately limited in its scope in order to explore the essential characteristics of such a system. Two simple example problems are then shown which illustrate the principles. A quantitative performance comparison is made between the hardware implementation and a traditional software implementation of the same algorithm using a microprocessor emulator. The limitations of this initial design are analysed and a number of alternatives are suggested.

Chapter 5 builds on the results in Chapter 4 and describes a more general solution that supports larger populations by exploiting the on-chip memory of the FPGA and by using external memory. A pipeline is also implemented in this revised design that enables the design to achieve a higher level of throughput. The performance of this design is illustrated using three problems.

Chapter 6 presents a behavioural analysis of two of the experiments from Chapter 5, focusing on the crossover operator. A new crossover operator is presented that appears to reduce the effects of bloat. Chapter 6 also analyses the behaviour of the random number generator used in the design, and compares it to a number of alternative random number generators, including a source of true random numbers. It shows that there are better generators than the generator used in Chapters 4 and 5.

Chapter 7 considers the economics of implementing GP in hardware and suggests some application areas that would be appropriate for such an implementation. The process of using Handel-C to implement GP is reviewed in the light of the experience gained while doing this research.

Chapter 8 summarises the work and presents the main conclusions of the work. Finally, a number of suggestions for future research are discussed and some possible applications, some of which are only made possible by implementing GP in an FPGA, are outlined.

The main body of the thesis is followed by 4 appendixes:

- Appendix A gives a detailed problem description for the experiments in Chapters 4, 5 and 6. It uses the format developed for the GP taxonomy in Chapter 2.

- Appendix B provides details of the data used to construct the taxonomy, giving the problem category, title, author and a subset of the problem specific data.

- Appendix C describes the details of the hardware and software tools used for the experimental work.
- Appendix D gives a detailed set of results from the random number generator analysis given in Chapter 6.

The first implementation of GP in hardware, described in Chapter 4, originally appeared in Genetic Programming and Evolvable Machines, volume 2, number 4 [Martin 01]. The optimised design was described in the proceedings of EuroGP'2002 [Martin 02b], while the results in Chapter 6 appeared in the proceedings of GECCO 2002 [Martin 02c] and [Martin 02a].

# Chapter 2

# Genetic Programming: A Survey and Taxonomy

This chapter surveys the literature in the field of *Genetic Programming* (GP). It begins with a brief historical overview of the precursors to GP and then reviews the general principles of standard GP. The review shows that before GP can find solutions to a problem the user of GP must choose many parameter settings. The individual parameters, or attributes, are reviewed, categorised and presented as a framework for categorising GP. The chapter concludes by suggesting that the GP algorithm is part of a more general process of problem solving, called Meta-GP. A common theme from the review and the idea of Meta-GP is the problem of long running times for GP when implemented in software. The possibility of speeding up GP by implementing GP in hardware is therefore of great interest.

## 2.1 Genetic Programming - A Historical Perspective

Genetic Programming is a stochastic search technique used to find computer programs to solve a given problem. GP is sometimes described as an extension of *Genetic Algorithms* (GAs) which were first proposed by Holland [Holland 75]. In GP the individuals that make up a population are not fixed length(1), limited alphabet, strings but structures that represent programs. To put GP

(1)This is a generalisation used in the GP literature. It is recognised that GAs can and do use variable length representations, for example messy GAs by Goldberg *et al.* [Goldberg 89] and the work by Smith [Smith 80].

into its proper context though it is instructive to look further back. Angeline [Angeline 98] gives a comprehensive historical perspective of GP – what he calls Executable Structures – and Banzhaf *et al.* [Banzhaf 98] also traces the history of GP. The following is a brief history of GP presenting the phylogeny of GP as a timeline.

In 1958 and 1959 Friedberg [Friedberg 58] [Friedberg 59] introduced the idea of a computer that could write programs without being told precisely how. Samuel, in his work on machine learning [Samuel 59], recognised the potential of automatic programming and suggested that "Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort". In 1965 Rechenberg [Rechenberg 65] suggested that evolution could be used to help find solutions to problems. In 1966 Fogel, Owens and Walsh [Fogel 66] described *Evolutionary Programming*, that solved problems using a finite state machine. At around the same time Rechenberg [Rechenberg 73] was developing *Evolutionary Strategies* (ES). In 1975 Holland [Holland 75] (revised in [Holland 92]) introduced GAs which had two important characteristics: the use of fixed length binary strings, and the adoption of crossover as the predominant breeding operator. Smith [Smith 80] developed the fixed length chromosome GA to use variable length chromosomes. Forsyth [Forsyth 81] evolved programs, represented as tree structured Boolean rules, using a GA-like algorithm. Cramer in 1985 [Cramer 85] used both linear and tree representations of the genome to evolve programs in a simple language. Hicklin [Hicklin 86] and Fujiki and Dickinson [Fujiki 87] used GAs to evolve LISP programs and Schmidhuber [Schmidhuber 87] developed a form of GP using PROLOG. Koza in 1989 [Koza 89] used LISP and a tree based representation to solve several problems including sequence induction, linear equations, Boolean problems and others. This work formed the basis for what was termed Genetic Programming in 1990 [Koza 90b], and which is described in detail in [Koza 92].

Figure 2.1 presents the phylogeny of GP as a timeline beginning with the first reported experimental work until the publication of Koza's book [Koza 92] in 1992 which described what has become known as the standard form of GP. The development of GP since 1992 forms part of the review in the next section.

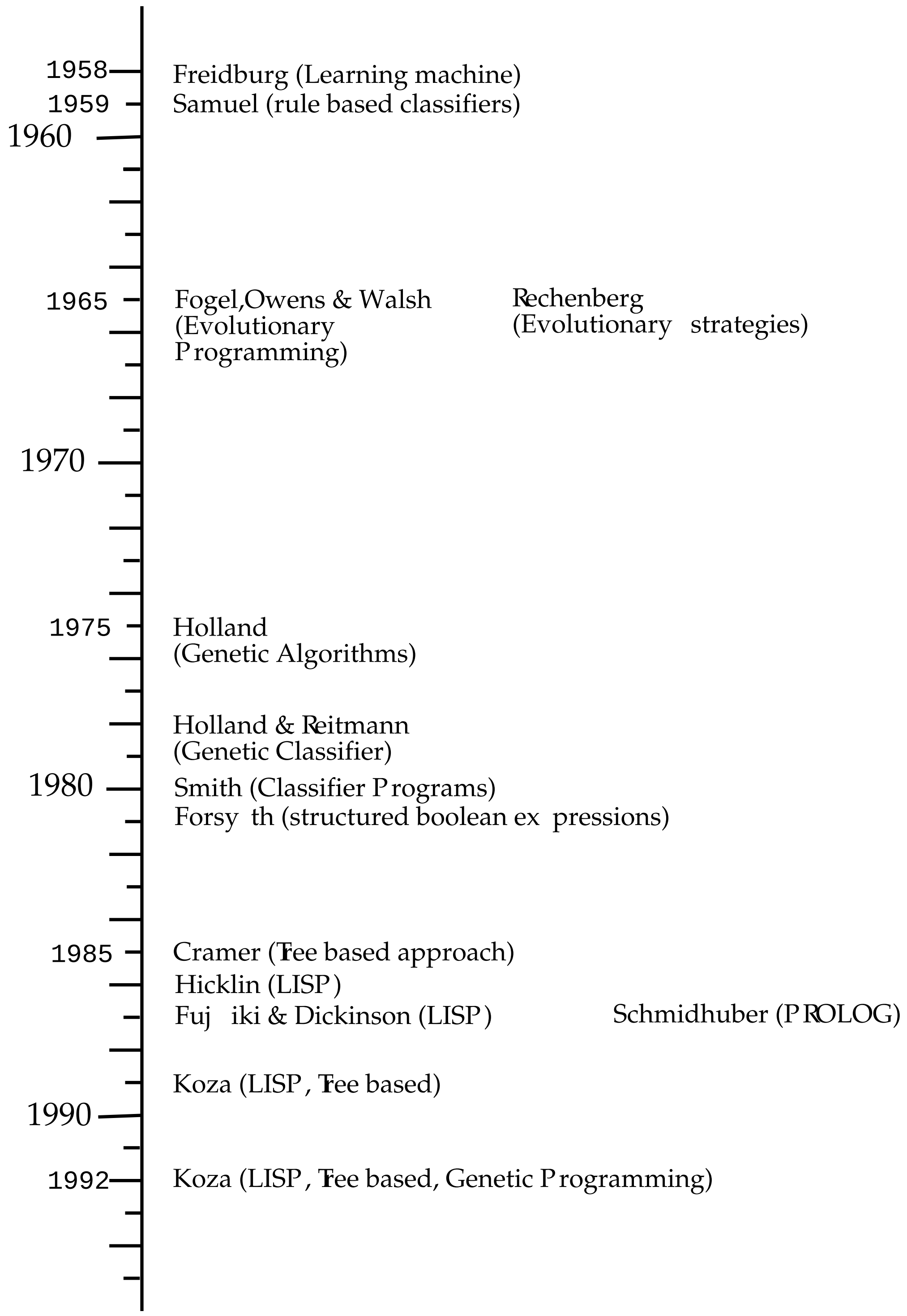

1958
Freidburg (Learning machine)
1959
Samuel (rule based classifiers)
1960
1965
Fogel,Owens & Walsh (Evolutionary Programming)
Rechenberg (Evolutionary strategies)
1970
1975
Holland (Genetic Algorithms)
Holland & Reitmann (Genetic Classifier)
1980
Smith (Classifier Programs)
Forsyth (structured boolean expressions)
1985
Cramer (Tree based approach)
Hicklin (LISP)
Fujiki & Dickinson (LISP)
Schmidhuber (PROLOG)
Koza (LISP, Tree based)
1990
1992
Koza (LISP, Tree based, Genetic Programming)

**Figure 2.1:** The phylogeny of GP as a timeline.

## 2.2 Principles of Genetic Programming

This section reviews the canonical or standard form of GP as described in [Koza 92]. This review provides a basis on which to build a more detailed parameter review later in this chapter. The basic components of GP – the tree representation and the components of the tree – are described first followed by the steps used by GP to solve a problem.

### 2.2.1 Program Representation

A distinguishing feature of standard GP is that the executable structures, or programs, are represented as trees. An example program tree is shown in Figure 2.2. This tree represents the expression $y = (a * (b/2)) + c$, where $a, b$ and $c$ are variables. The nodes labelled $a, b, c$ and 2 are called the leaf nodes, and the nodes labelled +,* and / are called non-leaf nodes.

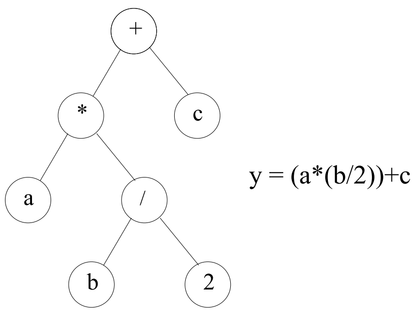


**Figure 2.2:** Example tree for an evolving program in standard GP.

### 2.2.2 Function and Terminal Sets

In standard tree based GP, each program consists of one or more nodes. The nodes are chosen from one of two sets – the function set $\mathcal{F}$ and the terminal set $\mathcal{T}$. The non-leaf nodes are taken from the function set $\mathcal{F} = \{f_1, f_2, ..., f_{n_f}\}$ and have arity (that is can take a number of arguments) of one or greater. The leaf nodes are taken from the terminal set $\mathcal{T} = \{t_1, t_2, ..., t_{n_t}\}$ and have arity of zero.

If the members of $\mathcal{T}$ are considered as functions with arity zero then the total set of nodes is: $\mathcal{C} = \mathcal{F} \cup \mathcal{T}$. The search space is the set of all possible compositions of the members of $\mathcal{C}$. This set must exhibit the two properties of *closure* and *sufficiency*.

Closure requires that each member of $\mathcal{C}$ can accept as its arguments any other member in $\mathcal{C}$. This property is required to guarantee that programs will operate without run time errors, because the members of $\mathcal{C}$ can be used in any arbitrary combination. Closure can be achieved in several ways. Koza [Koza 92] restricts the types of arguments and function return types to compatible types. For example, all floating point types as in the symbolic regression examples or logical types in the Boolean examples. For simple problems with single data types this is sufficient.

Where the problem requires more than one type, a typing mechanism can be employed. Strongly typed approaches such as those described by Montana [Montana 95] and Haynes *et al.* [Haynes 96] impose constraints on the creation of individuals to satisfy the type rules. Clack and Yu [Clack 97b] extended this work to show that expression based parse trees yield more correct programs, and introduced the idea of polymorphism into the data types.

The sufficiency property requires that the set of functions in $\mathcal{C}$ is sufficient to express a program capable of solving the problem under consideration. This is a problem specific property and must be determined before GP will successfully solve a problem. This together with determining a suitable fitness test requires the most effort by a user of GP.

### 2.2.3 The Genetic Programming Algorithm

GP uses four steps to solve a problem:

1. Initial population creation. An initial population of programs is created by randomly selecting nodes from the function set and terminal set.
2. Fitness evaluation. Each program is evaluated to measure how fit it is.
3. Breeding. The current population is then used to form a new population by selecting the better programs and using the following operators to propagate and modify the programs:
   (a) reproduction
   (b) crossover
   (c) mutation.
4. The new population is then re-evaluated.

Steps 3 and 4 are repeated until either a pre-determined number of generations have been processed or an individual meets a pre-determined level of fitness. The GP algorithm is illustrated as a flow chart in Figure 2.3. This cycle differs from the typical GA cycle where mutation is applied after crossover or reproduction.

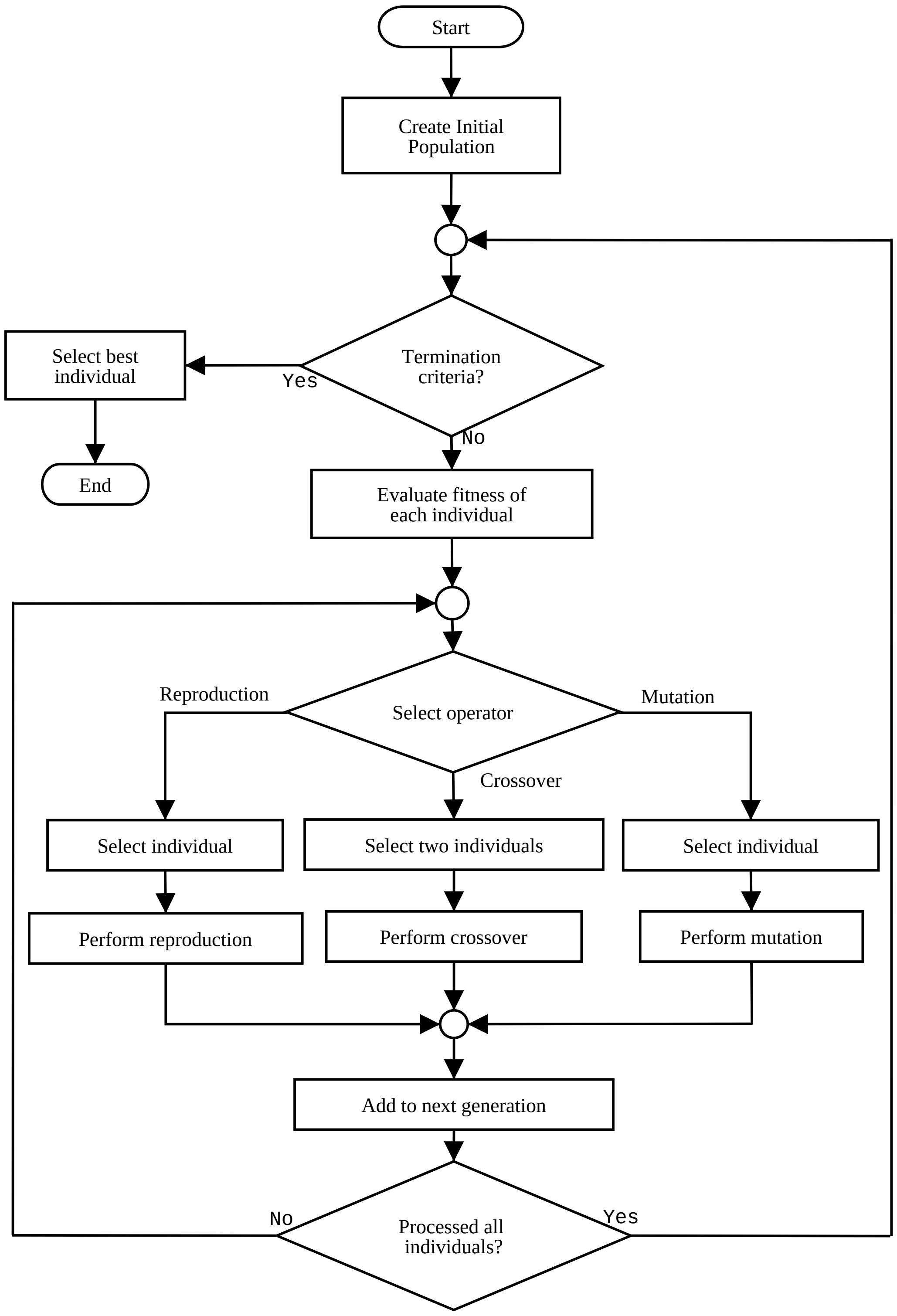


**Figure 2.3:** Flow chart of standard Genetic Programming algorithm.

### Initial population creation

To create the initial population, randomly selected nodes from the function set $\mathcal{F}$ are used to build trees according to the arity of the function. Leaf nodes from $\mathcal{T}$ are inserted according to certain criteria. Two main methods are described by Koza as the *full* and the *grow* methods.

The full method selects nodes from $\mathcal{F}$ until the tree reaches a pre-determined depth then it selects from $\mathcal{T}$. This results in trees with uniform depth.

The grow method differs in that a node is selected from $\mathcal{C}$ if the depth is less than a pre-determined maximum, else a node is selected from $\mathcal{T}$.

A third method combining the full and grow is called *ramped half and half*. Ramped half and half operates by creating an equal number of trees with a depth between 2 and a pre-determined maximum. That is if the maximum depth is 10, then 1/9 will have depth 2, 1/9 depth 3 and so on up to depth 10. Then for each depth, 50% of the trees are created using the full method and 50% using the grow method. This is claimed by Koza to offer a wider variety of shapes and size in the initial population. The difference in performance between the three methods is documented in [Koza 92], with ramped half-and-half yielding higher probabilities of success. Therefore this is the method used by most GP work. Other initialisation strategies have been proposed, including two probabilistic tree creation methods (PCT1 & PCT2) by Luke [Luke 00b], which allow the user to bias the tree creation so that nodes have a guaranteed probability of appearing.

### Fitness evaluation

The programs that have been created during the initial population phase, and later after the breeding phase, are ultimately expected to provide a solution to the problem being solved. The programs therefore need to be evaluated or tested against the problem specification. How well they perform is termed the fitness of the program. Standard GP used LISP which allowed the direct interpretation of the trees as programs. This was possible because LISP makes no distinction between programs and data. Other methods of evaluating the fitness are reviewed in Section 2.4.4.

Evaluating the fitness is highly problem dependent and is usually the most computationally expensive stage in the GP algorithm.

**Breeding the next generation**

When all the programs in the population have been evaluated and their absolute fitness is known, a number of programs are selected to be used to breed the next generation. There are two principal methods of selecting individuals from a generation: *fitness proportionate* and *tournament*. When using the fitness proportionate method an individual is selected with a probability proportionate to its fitness. From [Koza 92], if $f(s_i(t))$ is the non-zero fitness of individual $s_i$ at time $t$, then the probability that individual $s_i$ will be selected from a population of size $M$ for breeding is:

$$\frac{f(s_i(t))}{\sum_{j=1}^{M} f(s_j(t))} \tag{2.1}$$

A refinement on this is rank selection [Baker 87] which reduces the influence of single highly fit individuals.

In tournament selection, $n$ individuals are chosen at random from the population and the individual with the highest fitness is propagated to the next generation. The value of $n$ can be any number greater than one. The winning individual can be left in the donor population, resulting in so called over selection, where it stands a chance of being reselected by subsequent tournaments. Blickle and Thiele [Blickle 95b] provide a comprehensive comparison between different selection schemes for GAs. An interesting feature of their work is that it used GP to derive approximation formulas.

One of three operators are used to construct each individual of the next generation: reproduction, crossover and mutation.

**Reproduction** is the straightforward copying of an individual to the next generation, otherwise known as Darwinian or asexual reproduction. This is also referred to as cloning in some GA work.

**Crossover**, or sexual recombination, is illustrated in Figure 2.4 and consists of taking two individuals $A$ and $B$ and randomly selecting a crossover point in each. The two individuals are then split at these points creating four subtrees $A_1, A_2, B_1, B_2$ and two new individuals $C$ and $D$ are created by combining $A_1$with $B_2$ and $B_1$ with $A_2$. The choice of crossover point is not always uniform. For example, in [Koza 92, page 114] the choice of crossover points is biased so that 90% of crossovers choose internal points (non terminals) and the remaining 10% choose external points (terminals). This is claimed to promote the swapping of larger structures, whereas a uniform crossover point selection would merely swap terminals, effectively causing crossover to degenerate to point mutation.

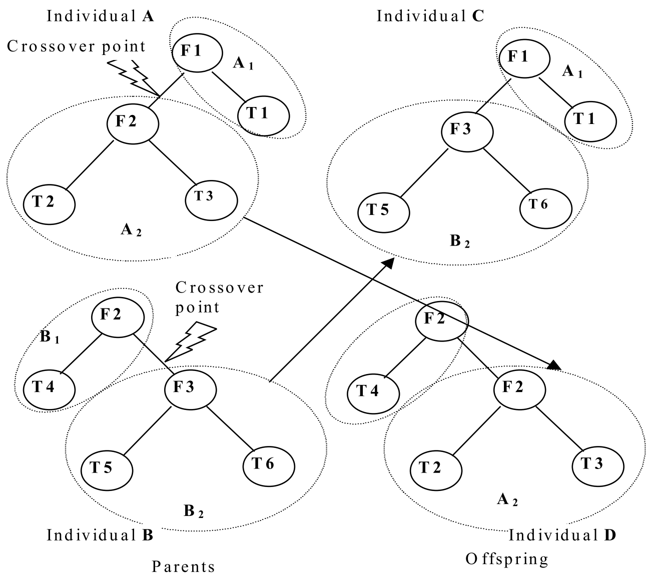


**Figure 2.4:** Crossover operating on two trees.

**Mutation** consists of randomly selecting a mutation point in a tree and substituting a new randomly generated sub tree at that point.

Standard GP processes all the individuals in a population during the breeding phase, creating a new population each time the breeding phase is executed. This strategy is called generational, and a straightforward implementation requires sufficient storage to maintain two complete populations. In some situations providing the storage for the second generation is not possible, so an alternative called the steady-state strategy [Syswerda 89] can be used. In steady-state GP a single population is maintained and a small number of individuals are replaced during each breeding phase.

## 2.3 Attributes of Genetic Programming

From the review of standard GP in the previous section it is evident that the user of GP must specify what primitives are going to be used to build the programs, what types of data the system is going to utilise, as well as run-time parameters such as population size and the number of generations to run the system for. The user also needs to select one or more operators to create new generations. Finally, a fitness measure is required to drive the selection process, together with appropriate stopping criteria for the runs. Further decisions need to be made in the light of experience to optimise the search process, or simply to get the search process to yield up a viable individual. In addition to standard tree based GP there have been many variations on the GP theme, each variation modifying one or more of the standard attributes, and a decision needs to be made whether any of the modifications would be appropriate.

In this section the attributes of GP are organised into four categories. The motivation for this arrangement is that by placing the attributes into groups, a clearer picture of the different types of attributes and their relationships will emerge. Presenting the attributes in this way also allows a more detailed description of a problem to be given in a compact form and this will allow a detailed comparison to be made between problems.

### 2.3.1 Existing Taxonomies

Several categorisations of particular aspects of GP have been made in the past. These fall into one of two broad categories: the topics of research into GP, and the problem types that have been tackled by GP.

#### Research topics

In the research topics category, Langdon [Langdon 98a] proposed a simple taxonomy based on a classification of the research topics that were of interest at the time, identifying the seven categories shown in Table 2.1.

**Table 2.1:** Langdon's taxonomy of GP that identifies the research topics in the published literature.

| |
|---|
| Applications |
| Representation |
| Fitness functions |
| Architecture |
| Search operators |
| Exotic GA techniques |
| Implementations |

#### Problem domains

Koza [Koza 94] gives a summary of GP applications. The grouping of applications suggests a taxonomy of problem types. He suggests the 10 categories listed in Table 2.2.

**Table 2.2:** GP applications suggested by Koza [Koza 94].

| |
|---|
| Art |
| Databases |
| Algorithms |
| Natural language |
| Modules |
| Programming methods |
| Robotic control |
| Neural networks |
| Induction and regression |
| Financial |

Banzhaf *et al.* [Banzhaf 98, (p341)] extends the list in [Koza 94] by further subdividing some categories and adding some new categories. They list 20 different applications areas, listed in Table 2.3.

**Table 2.3:** Application areas for GP from Banzhaf *et al.* [Banzhaf 98, (p341)].

| |
|---|
| Algorithms |
| Art |
| Biotechnology |
| Computer graphics |
| Computing |
| Control (general) |
| Control (process) |
| Control (robots and agents) |
| Control (spacecraft) |
| Data mining |
| Electrical engineering |
| Financial |
| Hybrid systems |
| Image processing |
| Interactive evolution |
| Modelling |
| Natural languages |
| Optimisation |
| Pattern recognition |
| Signal processing |

Langdon [Langdon 98a] lists 7 categories of GP applications, listed in Table 2.4.

**Table 2.4:** GP applications suggested by Langdon [Langdon 98a].

| |
|---|
| Prediction and classification |
| Image and signal processing |
| Design |
| Trading |
| Robots |
| Artificial life |
| Artistic |

## 2.4 A Taxonomy of Genetic Programming Attributes

Previous taxonomies of GP have considered a single attribute – the research topic, problem domain or problem type. However, as already highlighted in Section 2.2, using GP requires the user to consider many different parameters, and there does not appear to be a taxonomy describing the attributes and parameters of GP. This section therefore sets out to categorise the attributes of GP, placing the attributes into taxa.

### 2.4.1 Characteristics of a Good Taxonomy

When constructing a taxonomy, it is useful to consider what characteristics make a good taxonomy so that the taxonomy can be evaluated. Amoroso [Amoroso 94] suggests that a good taxonomy has seven important characteristics with regard to its classification categories ;

1. Mutually exclusive - classifying in one category excludes all others because categories do not overlap,
2. Exhaustive - taken together, the categories include all possibilities,
3. Unambiguous - clear and precise so that classification is not uncertain, regardless of who is classifying,
4. Repeatable - repeated applications result in the same classification, regardless of who is classifying,
5. Accepted - logical and intuitive so that they could become generally approved,
6. Useful - can be used to gain insight into the field of inquiry,
7. Expandable - can be extended to contain new categories.

### 2.4.2 Principal Taxa

A top level framework was initially devised containing four categories. Because GP has a strong relationship to traditional software engineering, in that at its most basic level it is designed to automate the programming activities of humans, the four categories reflect the software development lifecycle often encountered in software engineering;

1. **External**. This category lists the externally observed behavioral attributes of the problem without considering the implementation of the GP system or any solutions found. It is likely that these attributes could be used to describe any problem regardless of how the problem was going to be solved. In traditional software engineering these attributes represent the requirements or the specification of the system.

2. **GP specific**. These attributes are the values of the parameters used to create the GP system, such as function sets, terminal sets, whether memory is required and what representations are used. These attributes represent the design of the system, given the requirements and constraints.

3. **Results.** These attributes are derived from the results of running GP. Amongst other things these provide a picture of the effort required to find solutions. They also give some indication of the problem specific biases and optimisations that were applied to GP to find a solution. These attributes represent the quality of the generated solution .

4. **Implementation**. This category lists the implementation or platform attributes. These describe the physical implementation of the GP system including the computer platform, implementation language and whether the fitness function is internal or external to the GP system. In software engineering these attributes represent the constraints imposed on a design.

An important consequence of this division is that it separates the problem specific attributes from the GP specific attributes which allows cross-problem analysis to be made independently from the analysis of GP specific factors. It may be noted here that during the review of problems for which GP has been used, it became clear that many reports combine the solution of a new problem with the invention of new GP operators, representations and other GP specific features, making analytical comparisons difficult.

Some of the attributes are the standard parameters that are required for any GP system. These are sometimes described in a tableau after the examples found in Koza's books. For convenience, each attribute is given an abbreviated key.

The taxonomy was constructed by carrying out a review of the GP literature, and by collecting data from published reports of using GP for problem solving. The data collected was stored in a relational database which allows the data to be analysed using standard database tools. An example of some analysis is given later in this chapter that demonstrates the utility of this method. The problems were chosen with care to represent a range of GP applications, therefore the problems listed do not include every instance of the many test problems such as the Boolean multiplexers, symbolic regression or the artificial ant problem. Appendix B gives a list of the problems considered and a selection of the attribute data.

### 2.4.3 Externally Observable Attributes

These attributes apply to the problem without considering GP as a means of finding a solution. Indeed, they should be of use when considering any method of finding a solution since these generally relate to the specification of the problem.

#### Problem category (Pcat)

This gives a general categorisation of the problem. This is probably the most subjective attribute with many problems not falling cleanly into a single narrow category. We have already seen three possible categorisations of the problem description ( [Banzhaf 98] [Koza 92] [Langdon 98a]). A categorisation that is too finely divided is likely to make it hard to categorise new problems, because of the subjective and ambiguous nature of this attribute. A finely divided categorisation is also in danger of degenerating into a simple enumeration. Furthermore, in an invited talk at EuroGP'2002, Igno Wegener reminded the GP community that there is a distinction between problem types and instances of problems. Therefore, the number of problem categories has been condensed to 7, including a category that allows hybrid problems to be considered without having to invent new categorisations.

Pcat $\in$ {Art, Algorithms, Control, Design, Regression, Classification, Hybrid}.

**Formal specification of the problem (Fml)**

This attribute describes how the problem is stated. For example, the regression problems can be expressed as a mathematical formula such as the quartic polynomial $y = x^4 + x^3 + x^2 + x$ [Koza 92, p 164]. Other problem statement techniques such as Message Sequence Charts (MSCs) [Martin 00] or Boolean Logic statements have been used. In some problem domains no readily accessible methods of expressing a problem exist, for example when evolving art [Sims 91] or music [Johanson 98]. In these cases the fitness of individuals is judged on a subjective basis. It is interesting to note that despite the use of formal specification languages such as the Z notation [Spivey 92] [Bowen 02], VDM [VDM 96] [VDM 02] or LOTOS [LOTOS 00] [Turner 02] in industry, the use of formal specification methods has not featured in the literature of GP. Many problem statements in the literature are simply English language descriptions of the problem. The precise meaning of these statements is often left for the reader to interpret.

This attribute is important because traditional software engineering has consistently made the case for a clear statement of the requirements of a system. In [Martin 00] it is argued that the fitness function represents the requirements of the problem, and it would be expected that expressing the problem in a formal manner would help in determining the fitness function(s) required.
Fml $\in$ {Equation, English, Logic, Subjective, Message Sequence Chart (MSC), Exemplars}

**Number of fitness cases (Nfit)**

Records how many test cases were chosen. This attribute would be expected to help in identifying problems that require a large processing overhead in evaluating the fitness function.

**Class of Computability or language of GP (Lang)**

Several authors have already shown that for GP to be truly general, the language that GP uses to create programs with must have certain features. In this context, the term language refers to the function set $\mathcal{F}$, terminal set $\mathcal{T}$ and the constructs that bind the functions together, such as sequencing, iteration and conditional execution. In general these features are characterised as making GP Turing complete. The first explicit mention of this feature in GP was made by Teller [Teller 94c]. Yu has investigated a functional programming approach to GP that exhibits many of the characteristics of traditional computer programming languages [Yu 99], and Pringle [Pringle 95] proposed a structured programming approach to using GP. However, many problems can be solved using non-Turing complete languages as demonstrated by the examples from the work in [Koza 89] [Koza 92] [Koza 94].

Before we can use GP to try to solve a problem it is important to have some idea which language is appropriate. Failure to identify the language may result in using a more powerful language than is strictly needed, leading to poor performance in finding a program because of the overhead of executing unwanted program structures such as loops or recursion. The converse case of using a language not powerful enough could lead to the situation of never finding a general solution to a problem.

The selection of an optimal language is also likely to affect the performance of the GP kernel. For example, if a Turing complete language is used, then explicit steps need to be taken to ensure that general recursion is controlled, infinite loops do not compromise the efficiency of the fitness evaluations and in the case of large problems that memory consumption is not excessive. These extra steps involve added complexity and run-time overhead and would therefore increase the effort, both human and machine, required to solve a problem using GP.

From the literature of computer science (eg. [Feynman 96] [Savage 98] [Gruska 97] and others), four broad classifications of computer languages can be discerned:

1. Memoryless computing devices (ML). These include examples such as Boolean functions and expressions, combinatorial digital logic circuits and propositional logic. In the context of GP, common cases such as the continuous functions encountered in the symbolic expression problems can be added.

2. Finite memory computing devices (FM). This includes finite state machines and regular expressions.

3. Unbounded memory devices (U). These include recursive functions, random access machines, and the class of machines known as Turing machines. These can be considered general programs.

4. Other (O). Grouped in this category are non-deterministic programs and probabilistic programs. Recent work on Quantum computing [Spector 99] can be categorised in this family.

From this it is proposed that the bipolar view (turing or non-turing) of GP languages is expanded to consider 4 language types: The set of languages recognised is Lang$\in$\{ML, FM, U, O\}

### 2.4.4 Attributes Specific to Genetic Programming

**Function set ($\mathcal{F}$)**

The function set is decomposed into two attributes: First, the number of functions (NF) in $\mathcal{F}$. Second, the degree of abstraction of the function set is evaluated. The following guidelines are used for this category:

- Low abstraction represents a function set that is close to that of primitive computing machines, for example the arithmetic functions add, subtract, multiply and divide can all be executed by most processors directly, though with varying degrees of efficiency for some complex data types. This also includes all logical functions, most test functions (if-then-else constructs) and examples of machine code as for example in [Nordin 94].
- Medium abstraction functions are those that would require several statements in a high level language.
- High level abstraction is found where the functions perform major portions of processing, or have some major side effect. It is probable that this level would also represent specialised, problem specific functions.

Abs$\in${low, medium, high}

**Terminal set ($\mathcal{T}$)**

This simply records the number of terminals selected for the problem.

**Automatic feature discovery and reuse (ADF).**

Used to indicate whether some form of automatically defined function was used. This is not restricted to the original automatically defined functions, as described in [Koza 92], but includes other methods of capturing frequently used features such as automatically defined macros [Spector 95a] and architecture altering operations [Koza 96a].

**Memory (Mem)**

Some problems require the use of memory, for example to store internal results or state variables. Memory can take on many forms, but commonly found examples are indexed memory as described

by Teller [Teller 94b] as well as more sophisticated data structures such as queues, stacks and lists [Langdon 98a].
Mem∈{none, indexed, structured}

**Evolutionary representation (Rep)**

Standard GP uses a tree representation because it was argued that traditional programs can be represented naturally using such a structure. Indeed, compiler technology often uses such structures for the internal representation of the program. In addition, the LISP language S-expressions can easily be represented as trees. Most examples of GP have used this same basic structure. However several researchers have used alternative representations. Perkis [Perkis 94] showed that a stack representation could be used to evolve programs. Handley [Handley 94b] and Poli [Poli 99] used a directed graph. A linear representation has been used by other researchers, notably Nordin [Nordin 94] in his work on evolving machine code programs for a *Reduced Instruction Set Computer* (RISC). This work was later extended to a *Complex Instruction Set Computer* (CISC) [Nordin 99a]. Miller and Thomson used a method called Cartesian Genetic Programming [Miller 00] in which the program is represented as an indexed graph, encoded as a set of linear string of integers.

Recently hybrid representations have been investigated. Kantschik and Banzhaf used a linear tree representation [Kantschik 01], and a linear graph representation [Kantschik 02].
Rep∈{tree, stack, graph, linear, hybrid}

**Polymorphism and data typing (Typ)**

Simple test problems often use only a single data type. However, real world applications are seldom so regular and need to make use of more than one data type. The treatment of multiple data types, or polymorphism, can be handled by one of several methods. Koza used constrained syntactic structures [Koza 92] that employed problem specific syntactic rules to ensure correct programs were generated. Montana [Montana 95] extended this and described a method of imposing a strong type mechanism on GP. Clack and Yu [Clack 97b] extended this work. A weak method that did not constrain the generation or evolution of the trees is described in [Martin 00] which uses polymorphic data types.

It has been argued that explicit typing can be considered as a form of language bias [Whigham 95] and this has led to alternative approaches that separate the structures used for evolution and the structures used for the programs. The alternatives are reviewed in the next sub section.
Typ∈{none, weak, strong}

**Representation mapping (Map)**

A distinguishing feature of standard GP is that the same trees are used to represent the genotype as well as the program. That is to say, in standard GP the tree structure is used directly by the fitness evaluation to effectively execute the evolved program, as well as being manipulated and modified by the breeding operators. Several indirect mappings have also been used in which the genotype is processed to produce a different phenotype. Banzhaf in [Banzhaf 93] used a standard fixed length linear GAs to construct programs by using editing and repair to produce variable length trees. Holmes and Barclay [Holmes 96] describe the Odin Genetic Programming system which separates the genetic structure from the tree structure of the program. Whigham [Whigham 95] first introduced the idea of using grammars for specifying the structure of the language for the evolved programs. More recently, O'Neill [O'Neill 99a] and O'Neill and Ryan [O'Neill 99b] have developed a technique called Grammatical Evolution in which a variable length binary genome is translated to a program using a *Bakus Naur Form* grammar. Similar work called Gene Expression Programming is reported by Ferreira [Ferreira 01].

One of the major advantages claimed for separating the structures used for evolution and the structures used to represent programs is that the evolutionary mechanism does not need to be concerned with maintaining closure during crossover, thereby making the crossover operator simpler.
Map ∈{Direct, Indirect}

**Operators (Op)**

Crossover, mutation and reproduction are used with many modifications and enumerating them would be both impractical and not tell us very much about the selection of operators. Instead the ratios of the basic operators (mutation (M), reproduction (R) and crossover (C)) are given. In addition, an indication is made if a modified form of an operator is used. For example, the modification proposed by Langdon [Langdon 99] called homologous crossover is classified as a modified operator.

There is still much debate over the usefulness of crossover and/or mutation [GPMail 02]. Koza in [Koza 92] claims that mutation does not play a large part in finding fit individuals and consequently does not use it in most of his experiments. In contrast, studies by Banzhaf *et al.* [Banzhaf 96], Luke and Spector [Luke 97, Luke 98b] show that mutation can be useful in some cases, however they have not discovered any robust heuristics that allow the selection of optimal settings. Finally, Angeline [Angeline 97] puts forward some evidence that crossover may be a form of macromutation and not play any real role in propagating so called building blocks.

#### Creation methods (Crt)

When trees are used for the program representation, this describes the tree creation method. For other representations, including linear representations, the populations are usually initialised using a uniform method.

Crt$\in$ {Grow, Full, Ramped, Probabilistic, Uniform}

#### Seeding (Seed)

Denotes whether the population was seeded or inoculated with known good approximate solutions. [Ferrer 95] [Grant 00] [Langdon 00] and others have shown that seeding a population can both improve the search and lead to generalisations of known good solutions.

#### Population size (M)

Records the size of the population.

#### Number of generations (G)

Records the number of generations the system was run for.

#### Selection method (Sel)

Sel $\in$ {Proportionate, Tournament, Other}

**Generational method (Gen)**

Gen $\in$ {Generational, Steady-state}

### 2.4.5 Attributes Derived From Experimental Results

These are the values of attributes found in the experimental results.

**Program size (Size)**

Typical size (in nodes) of a representative solution.

**Successful outcome (Suc)**

Since the principle aim of using GP is to find solutions to problems, it is not surprising that it is rare to find reports of experiments that have failed to find a solution, though from experience many experimenters know that the number of unsuccessful runs during the development of a GP system is usually high. The number of unsuccessful runs is normally reduced by modifying run time parameters, genetic operators, function sets, or other attributes. Only when the number of runs that produce an acceptable result is reasonable do any results get published. One notable exception is found in Langdon [Langdon 98a, (pp 149-154)] where an attempt to evolve a general purpose program to recognise a Dyck language without using pre-defined stack primitives failed. The dearth of results for unsuccessful runs rather hampers us in the search for a set of acceptable parameters since we only have a part of the data needed. This attribute is therefore an indication of whether a given set of parameters produced a successful run.

**Effort required to find a solution (E)**

Ultimately, any search method is going to be evaluated on the quality of the result and how much effort needs to be expended to discover an acceptable result. Clearly no matter how elegant the search method, if it requires more effort than an alternative method, it will not be favoured. Because of the wider acceptance of the 'no free lunch theorem' [Wolpert 97] that says no one search method will perform better for all problems than any other search method, we need a robust, quantifiable measure of the effectiveness of GP. GP is a stochastic search method, and so it is likely that any meaningful measure of effort will need to consider the probability of finding an acceptable solution as well as elapsed time. Accordingly, several methods of measuring the effort needed by GP to find a solution have been used: a prediction of the number of runs required, the actual number of runs required and the wall clock time.

One method of reporting effort is to predict or measure the number of chromosomes that must be evaluated to find a solution. Koza gives a method of predicting the number of runs required to find a solution with a given probability based on measuring the performance of runs that were successful [Koza 92, page 194]:

$$R(z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil \tag{2.2}$$

where $R(z)$ is the number of runs required to find a solution with probability $z$. In most of the published work $z$ is assumed to be 0.99. $P(M,i)$ is the cumulative probability of having found a solution using a population size of $M$ by generation $i$. This equation is invalid if all runs are successful since $P(M,i) = 1$.

The number of chromosomes (individual programs) $I$ that need to be processed to give a probability $z$ of finding a solution by generation $i$ is:

$$I(M,i,z) = M \cdot R(z) \cdot (i+1) \tag{2.3}$$

This method is useful for comparing results between experiments that use different numbers of generations or different population sizes. It has been pointed out that this method has some weaknesses [Christensen 02] when used with a limited number of runs, and that to obtain robust results, the number of runs used to calculate the probabilities should be increased from the usual 50 by a factor of at least 10.

Miller and Thomson in [Miller 00] argue that the commonly used measure of effort (Equation 2.2) is really a prediction of a future experiment, rather than a measure of an experiment that has already been performed, and they propose a new method called *hit effort*. A hit is recorded for a run of a GP system when the GP system finds a correct solution to the problem. Using this method, an experiment is repeated for a number of runs. Over these runs, if $n$ evaluations are performed, and there are $h$ hits, then the hit effort $e$ is given by:

$$e = \frac{n}{h} \tag{2.4}$$

They show that this measure gave more stable results than Equation 2.2. Clearly this equation is invalid if no hits are recorded. This method is only useful for comparing results between experiments that use the same number of generations and population size.

Wall clock time is a measure of the elapsed time needed for a GP run, measured in seconds, minutes, hours or some other time interval. Most reports of elapsed time emphasise the fact that hard problems take a long time. Examples include Langdon's queue problem [Langdon 98a, p269] taking all day to run on a SUN workstation (speed not specified). Koza and Bennett [Koza 99a] using a 56 node Beowulf-style [Sterling 95] cluster of 533 MHz DEC Alpha machines reported that it took 28.4 hours to complete a run with a population size of 1,120,000. In this case the fitness evaluation took the form of running circuit simulations using a modified version of Spice [Spice 02].

This attribute is recorded as either the predicted number of chromosomes (Predict), actual number of chromosomes (Actual) or None. The numerical value is also recorded.

### 2.4.6 Implementation Details

#### Class of GP platform (Plat)

Almost all GP systems are realised as software running on general purpose computers. A list of some of the available packages is given in Table 2.5, and a comprehensive review at the *Evonet* site [Evonet 02].

**Table 2.5:** Popular software implementations of GP systems

| Name | Language | Comments | Reference |
|---|---|---|---|
| LISP Kernel | LISP | The original code from Koza's first book. | [Koza 92] |
| lilgp | C | A C implementation of the work described in [Koza 92] with additions. | [Zongker 02] |
| Gpc++ | C++ | A strongly Object Oriented based implementation. | [Weinbrenner 02] |
| GP-COM | C++, Tcl, Tk | Component based system with GUI. | [Harris 96] |
| Gpquick | C++ | Simple GP system written in C++ by Andy Singleton. | [Singleton 94] |
| GPDATA | C++ | Extensively modified version of GPquick to support data structures plus much more. | [Langdon 02a] |
| Gpsys | Java | A Java 2 based Genetic Programming system | [Qureshi 00] |
| Gpjpp | Java | A Java implementation of the Gpc++ kernel | |
| SGPC | C | Simple Genetic Programming in C. Based on Koza's LISP code. | [Tacket 93] |
| Vienna | C++ | | [Vienna 02] |
| Gpeist | Smalltalk | | [White 94] |
| pushgp | Java | A GP system that evolves programs in the Push programming language. | [Spector 02] |
| GPSRegress | Mathematica | Package to perform symbolic regression | [Nguyen 94] |
| GPS | C/C++ | Genetic Programming Studio. An implementation of lilgp with a Microsoft Windows GUI interface. | [Campo Novales 02] |
| GPX/Abstractica | | Applies GP to the generation of abstract images. | [Sims 91] |
| YAGPLIC | C | Yet another genetic programming library in C | [Blickle 95a] |

One alternative to using a standard general purpose CPU is to build a custom CPU that contains instructions that have been optimised for the task. Koizumi *et al.* [Koizumi 01] describe a RISC processor designed for GAs. This processor has been designed using *Application Specific Integrated Circuit* (ASIC) technology and implements special instructions that perform some common operations found in GAs, including a single cycle random number generator, instructions for operating on partial words, and some instructions that operate on more than one operand at a time. The latter are called *Single Instruction Multiple Data* (SIMD) operations. They predict a clock speed of 100 MHz using a 0.35 $\mu$m CMOS process. The predicted speed falls short of modern general purpose processors that run at over 2 GHz and so it is questionable if this approach would deliver a useful speedup.

Another approach is the realisation of the evolutionary computation process itself in hardware. A *Very Large Scale Integration* (VLSI) system has been proposed by Yoshida *et al.* [Yoshida 01] which implements a parallel GA machine using two VLSI devices. Examples of previously reported GP systems that use custom hardware in the form of *Field Programmable Gate Array* (FPGA)s are: Heywood [Heywood 00] where the FPGA is used to accelerate the fitness evaluation of GP and Sidhu *et al.* [Sidhu 99]. This is an area that is reviewed in more detail in Chapter 3 and studied in depth in Chapters 4 and 5 of this thesis.
Plat$\in$\{Soft, Hard, Hybrid\}.

### Fitness case implementation (Fimpl)

There is a range of different fitness case implementations reported. These implementations include intrinsic (built-in to the Genetic Programming system), extrinsic (external circuit simulators [Koza 96b], specialised processing devices [Koza 98b] and physical entities such as robots [Langdon 01]) or hybrid systems.
Fimpl$\in$\{Intrinsic, Extrinsic, Hybrid\}.

### Parallelisation of GP (Par)

GP may be run on single machines, but it is well known that GP and other evolutionary computing techniques are highly parallelisable and exploiting this parallelism can result in substantial performance improvements. Cantu-Paz [Cantu-Paz 98] surveyed parallel GA algorithms in depth and proposed four classifications of parallel GAs. A uniform taxonomy of parallel Genetic Algorithms has been proposed by Nowostawski and Poli [Nowostawski 99], which extended the number of classes of parallel GAs to eight (Table 2.6):

Various examples of parallel GP exist, for example the work by Andre and Koza [Andre 96] used a network of Transputers, while Chong and Langdon [Chong 99] explored how the computing resources that are potentially available on the internet could be exploited. Other examples of paral-

**Table 2.6:** A uniform taxonomy of parallel genetic algorithms from Nowostawski [Nowostawski 99].

| |
|---|
| master-slave |
| static subpopulations with migration |
| static overlapping subpopulations without migration |
| massively parallel genetic algorithms |
| dynamic demes |
| parallel steady-state GAs |
| parallel messy GAs |
| hybrid methods |

lel GP include the work done by Koza *et al.* [Koza 00c] which used a thousand standard Pentium PCs, and the use of a massively parallel SIMD machine by Juille and Pollack [Juille 96]. Fernandez and Tomassini have explored the use of parallel GP for several applications [Fernandez 99] [Fernandez 00a] [Fernandez 00b]. This attribute uses the eight categorisations from [Nowostawski 99]. Par $\in$ {MasterSlave, StaticMigrate, Static, Massive, Dynamic, SteadyState, Messy, Hybrid,None}.

## 2.5 A Summary of the Taxonomy

All the attributes discussed in the previous section are summarised in Table 2.7 on the following page which shows the attribute names, the abbreviations used, and the type and range of the data. This table representation of the attributes of GP also lends itself to describing a GP problem in more detail than the standard tableau first introduced by Koza in [Koza 92], and has been used to describe the problems used in the rest of this thesis. The problems are described in Appendix A.

**Table 2.7:** Summary of the attributes of GP and their characteristics.

| Category | Abbreviation | Types and range of values |
|---|---|---|
| **EXTERNAL** | | |
| Problem category | Pcat | From the set Pcat |
| Formal specification | Fml | From the set Fml |
| Number of fitness cases | NFIT | Integer |
| Language | Lang | From the set Lang |
| **GP SPECIFIC** | | |
| Function set size | F | Integer |
| Function set abstraction | Abs | From the set Abs |
| Terminal set size (T) | T | Integer |
| Automatic feature discovery | ADF | True/False |
| Memory | Mem | From the set Mem |
| Program representation | Rep | From the set Rep |
| Polymorphism and data typing | Typ | From the set Typ |
| Representation mapping | Map | From the set Map |
| Crossover operator | X% | Real |
| | Xmod | True/False |
| Mutation operator | M% | Real |
| | Mmod | True/False |
| Reproduction operator | R% | Real |
| | Rmod | True/False |
| Creation method | Crt | From the set Crt |
| Seeding | Seed | True/False |
| Population size | M | Integer |
| Generations | G | Integer |
| Selection method | Sel | From the set Sel |
| Generational method | Gen | From the set Gen |
| **RESULTS** | | |
| Program size | Size | Integer |
| Successful | Sucs | True/False |
| Effort | Effrt | Real |
| Wall clock time | Time | Real |
| **IMPLEMENTATION** | | |
| Class of GP platform | Plat | From the set Plat |
| Fitness function implementation | Fimpl | From the set Fimpl |
| Model of parallelisation | Par | From the set Par |

The attributes have been used to construct a tree based taxonomy, where the four main categories of attributes form four principal sub-divisions. The major attributes are then placed within each subdivision. Where appropriate, the attributes have then been subdivided further into subcategories. The taxonomy is shown in Figure 2.5 on the next page.

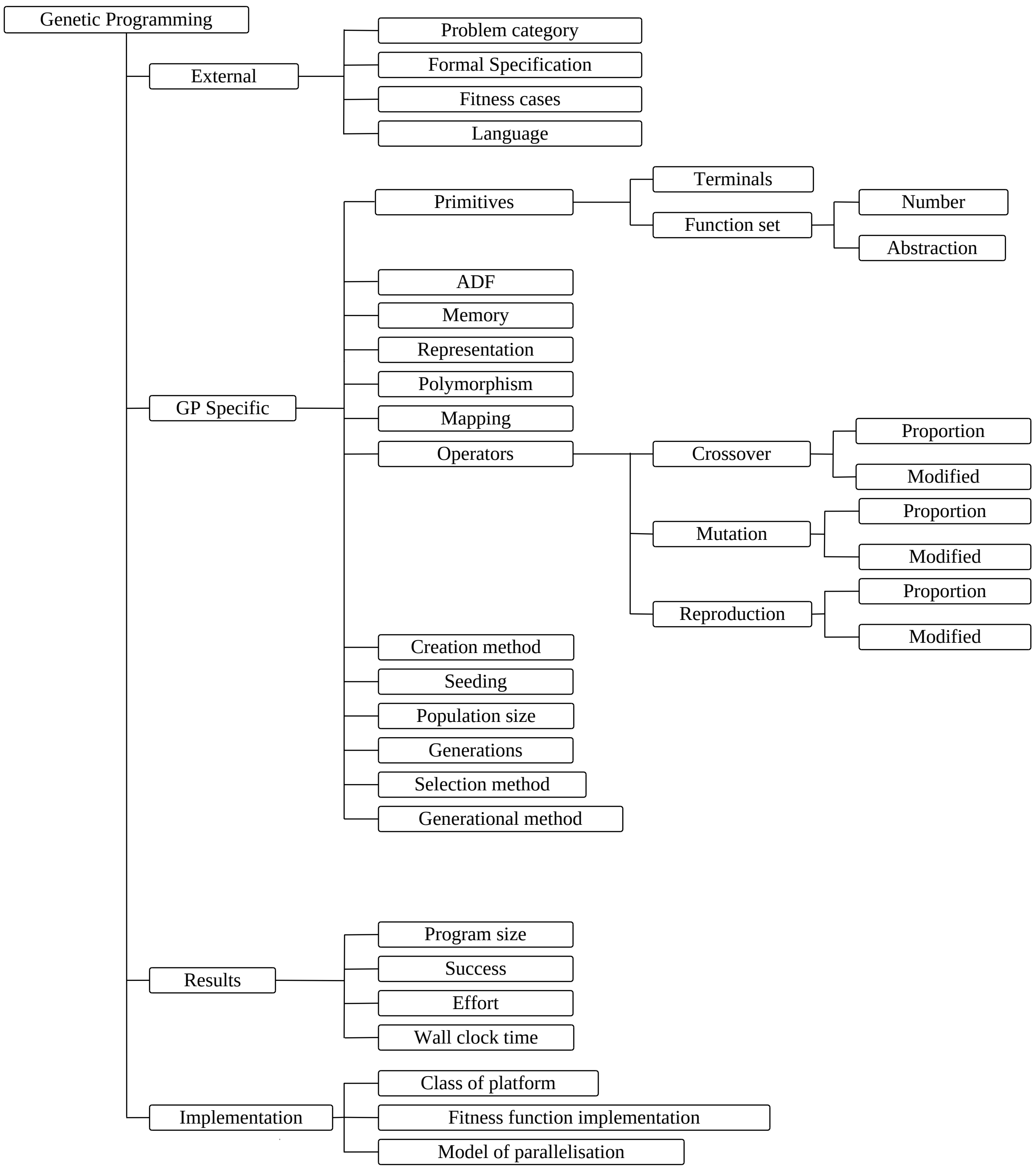


**Figure 2.5:** A tree based taxonomy of the attributes of Genetic Programming

## 2.6 Results and Discussion

### 2.6.1 An Analysis of the Raw Data

This section gives five examples of how the data used to build the taxonomy can be used to obtain quantifiable data on how GP has been used. The raw data has been used to obtain: the distribution of application areas, a measure of the sizes of typical programs using GP, the number of generations used, the split between hardware and software implementations, and the range of population sizes. These were chosen to see how useful these measures were as attributes of a taxonomy.

The distribution of top level problem categories is shown in Figure 2.6. Although there is a bias

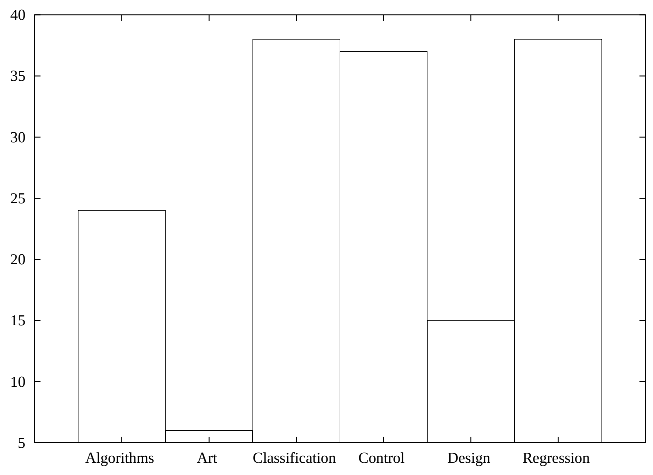


**Figure 2.6:** Distribution of top level problem categories for 158 examples of GP.

towards the classification, control and regression problems, none of the categories dominate. The spread of program sizes is shown in Figure 2.7. One conclusion we can draw from this result is that for the most part GP has been restricted to solving problems that require a modest sized program and is indicative of the obstacles that face GP when trying to scale up to produce large programs.

The number of generations that the problems were run for is plotted in Figure 2.8. This shows that the majority of problems analysed required less than 125 generations.

From 131 examples used to construct the taxonomy only three have used hardware for all or part of the implementation.

The population sizes used in the experiments is shown in Figure 2.9. This shows that almost all problems have used population sizes 50 000 or less.

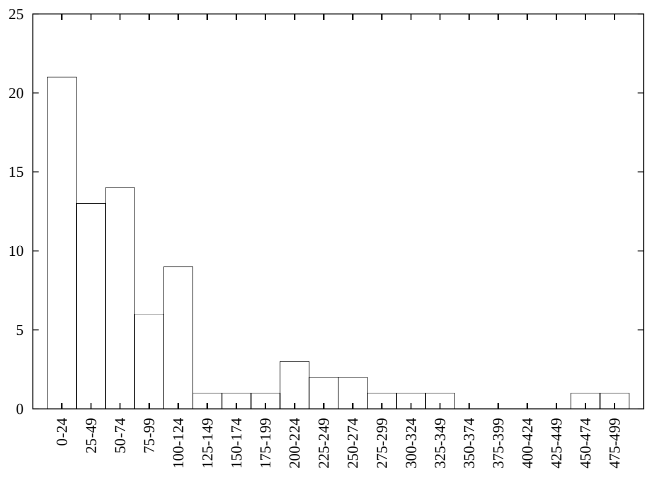


**Figure 2.7:** Spread of program sizes found in 78 Genetic Programming examples

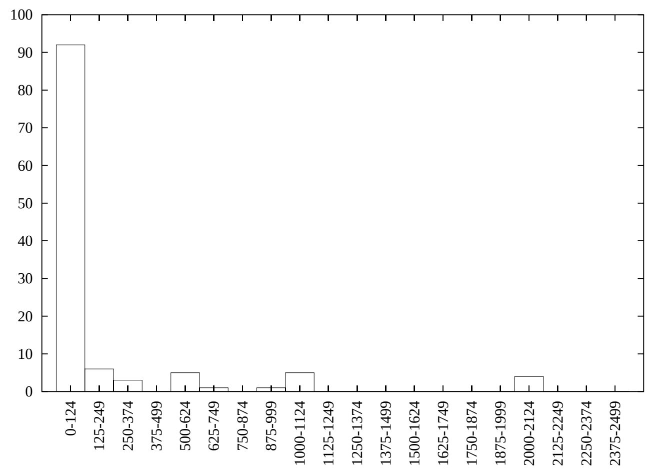


**Figure 2.8:** Number of generations for 120 problems.

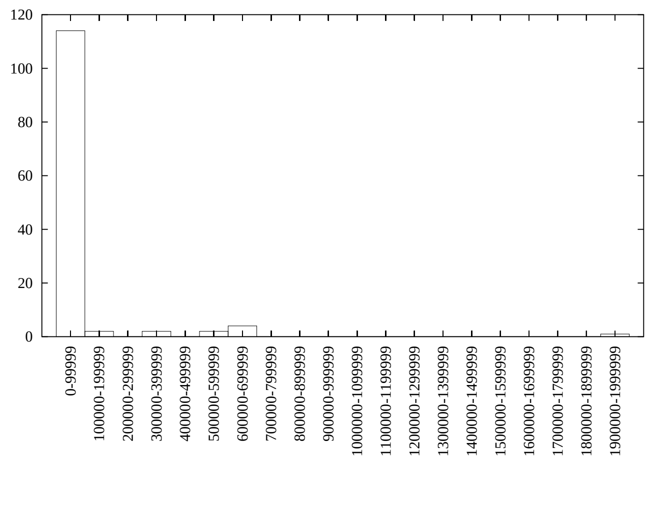


**Figure 2.9:** Population sizes for 125 problems.

However, Figure 2.9 does not show the details of the population sizes for the majority of problems, so Figure 2.10 shows the population size distribution for sizes less than 10 000. This shows that the majority of population sizes used in the reported experiments are less than 1000.

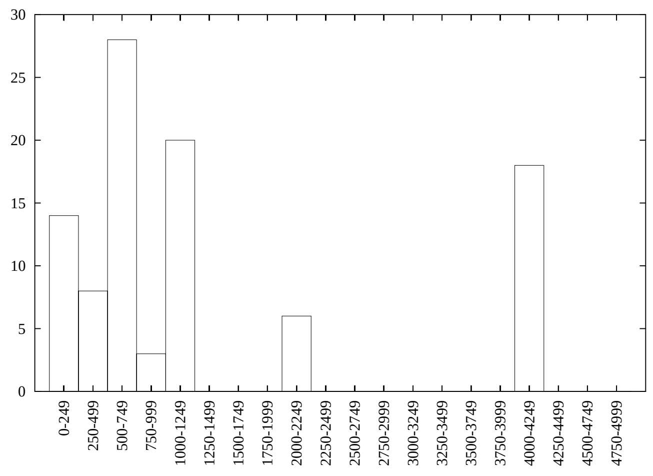


**Figure 2.10:** Population sizes for the 101 problems where the size is less than 10 000.

The examples in this section are by no means exhaustive, but they show the utility of the taxonomy when analysing GP.

### 2.6.2 Conformance to Original Criteria

The taxonomy presented in this chapter has been constructed with the characteristics in Section 2.4.1 in mind. To date, the first four criteria (mutually exclusive, exhaustive, unambiguous and repeatable) have largely been met, in that the number of categories available covers all examples of GP that have been analysed. Number 5 (accepted) is for others to judge. Number 6 (useful) can be judged from the fact that the strong software bias of GP implementations can be quantitatively shown. Number 7 (expandable) is addressed in Section 2.6.4. It is hoped that the taxonomy will prove to be useful in identifying other areas of research.

### 2.6.3 Meta Genetic Programming

Despite the impression given by some of the literature on GP, running an instance of the GP algorithm is only part of a process. Koza [Koza 92, p121] enumerates 5 steps needed to operate GP: 1) determining the set of parameters, 2) determining the set of functions, 3) determining the fitness measure, 4) determining the parameters and variables for controlling the run, and 5) determining the method of designating a result and the criterion for terminating a run. In later work [Koza 94], an additional step is used: 6) determining the programs' architecture. In reality however, it is unusual that using GP would be a linear step by step process. It is likely that for many examples of GP the parameters have been found by trial and error until the system produced acceptable results, or that particular parameters were changed to get a particular result. This fact is not always acknowledged in the reports of running GP experiments, though some authors do explicitly report the fact, for example [Poli 00].

Despite continuing advances in the theory of evolutionary techniques in general, and GP in particular, there is a general lack of robust theoretically based rules for determining parameter values. One exception is the development of general schema theories [Poli 01a] [Poli 01b]. An application of the schema theory shows how the choice of breeding operators and their proportions can be made to meet certain structural goals during a run of GP [McPhee 02]. There is also a limited amount of cross problem analysis of parameter settings available from experimental results. Most studies into parameter selection focus on a single problem or a limited set of problems. While this gives valuable insight into particular aspects of GP, it does not help practitioners who want to apply GP to hitherto untried problems. One helpful set of guidelines has been provided by Banzhaf *et al.* [Banzhaf 98, (pp 334-338)], giving the practitioner a few rules of thumb in choosing function and terminal sets, mutation and crossover rates and allowable depth of the program trees. One of the problems faced by practitioners is that the search space of viable parameters is large and that changing one parameter by a small amount, for example the number of generations to run, will cause other effects come into play, such as extended run time or a complete failure to find a solution.

The process of modifying the parameters during a GP experiment can be seen as a form of Meta-GP. In practice, the results of an experiment are used to help modify one or more of the GP parameters identified in the preceding sections. A flow chart of this meta GP process is shown in Figure 2.11.

The initial choice of function set, terminal set, fitness function and operational parameters will probably be based on previous examples of using GP, and on the specialised domain knowledge of the user. After the first experimental run, the choice of which of the many parameters to change, or whether the function set, terminal set, or fitness function should be modified is driven by observations about how the population and individuals behave as well as the quality of the result obtained from the previous runs. In this way, over the lifetime of running a GP experiment,

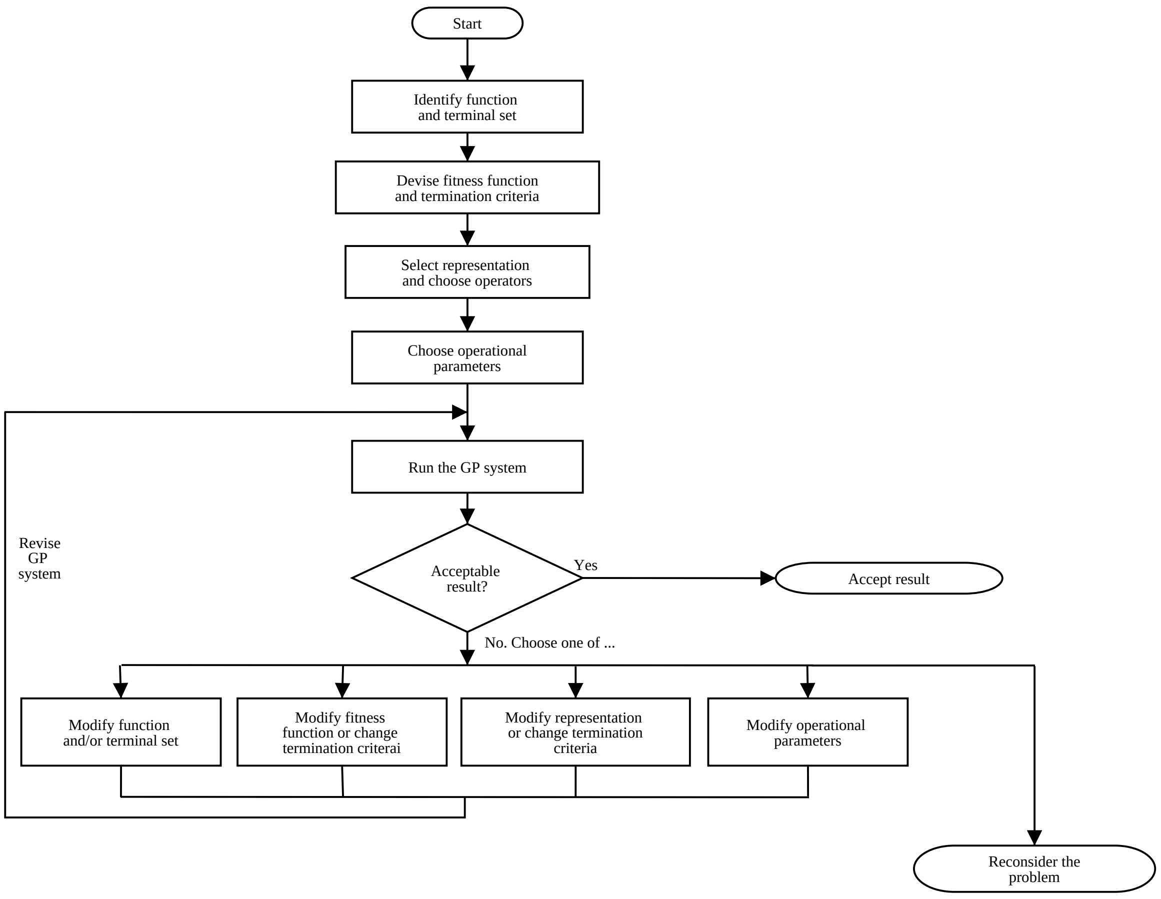

Start
Identify function and terminal set
Devise fitness function and termination criteria
Select representation and choose operators
Choose operational parameters
Run the GP system
Acceptable result?
Yes
Accept result
No. Choose one of ...
Modify function and/or terminal set
Modify fitness function or change termination criterai
Modify representation or change termination criteria
Modify operational parameters
Revise GP system
Reconsider the problem

**Figure 2.11:** Flow chart for the meta GP process.

an evolutionary approach is used initially to find the set of primitives needed, and then to tune the operational parameters.

Generally, Meta-GP takes the form of an iterative process that involves the (human) implementor of a GP system, but several examples where GP has been used to evolve settings of some of the parameters have been published. For example, Kantschik [Kantschik 99] used GP to evolve an improved set of operators, and a precursor to standard GP by Schmidhuber [Schmidhuber 87, (pp 7-13)] used a GP like system that recursively improved its performance.

Of course, the idea of an evolutionary approach to problem solving is not new. Fogel, Owens and Walsh [Fogel 66] suggested that the scientific method is an evolutionary process, and the adoption of evolutionary techniques in software engineering is well established, for example [Stevens 98] and [Somerville 97]. There is a parallel between a mechanical evolutionary system such as GP or ADATE [Olsson 95] and human centred techniques that use an evolutionary approach, including Meta-GP. They both share an iterative invent-test-modify cycle that uses feedback from previous results to refine the search. In the case of GP, promising solutions are identified by their fitness, and the modify phase is carried out by using one or more of the operators commonly found in GP such as reproduction, crossover and mutation. In Meta-GP, the identification of promising solutions is usually the acceptance or rejection of the results of the latest experiment, often by comparing partial results, analysing population dynamics or considering the rate of convergence towards an acceptable solution. The modification phase consists of a change to one or more of the attributes of the system. Despite these similarities, two important differences exist between GP and Meta-GP. Firstly, in the case of GP, a population of individuals is considered together, while in Meta-GP, usually only one individual is considered at a time. The second major difference is that repeated GP runs, given identical starting conditions and a repeatable source of pseudo random numbers, will always behave the same. However, Meta-GP relies on characteristics that have not yet been encoded in a computer program – the creative ingenuity and intuitive leaps of the user to select the initial starting conditions, identify fitness and choose which parameters to change, and so will be less repeatable.

### 2.6.4 Extending the Categories

The taxonomy presented here has attempted to be exhaustive with respect to GP problems, but there are arguably other attributes that could be added. Examples of these include details of the problem specification, such as the number of objectives that the problem has to meet or whether the problem is known to be NP.

Daida *et al*. [Daida 97], [Daida 99] showed that other parameters such as the range over which fitness values are spread, or the random number generator algorithm used, can have a large impact on the performance of GP, and so should also be considered for the taxonomy.

It is recognised that representations and operators are probably the most commonly studied and reported of the GP specific attributes, and that the proposed classification of these attributes is rather limited. The taxonomy would benefit from a further refinement of these attributes.

## 2.7 Summary

Following a historical review of the precursors to GP, this chapter presented a review of standard GP which showed that GP has many parameters and settings that must be chosen if GP is to find an acceptable solution to a problem. From the review it is also clear that there have been many modifications made to standard GP. This prompted a more detailed review of the many attributes associated with GP with a view to grouping related attributes into a taxonomy. One result of constructing the taxonomy is that it provides a formalism to describe a GP problem in more detail than the standard tableau. The data used to construct the taxonomy also allows the discovery of some general features of the problems for which GP has been used. Five examples have been presented showing the spread of applications, the distribution of program sizes, the range of population size, the different numbers of generations used and the very strong bias towards software implementations of GP.

Practical experience has shown that running GP is not simply a case of choosing a few attribute values and setting the GP system running. Using GP is part of a larger process, called Meta-GP. Meta-GP is an iterative and interactive process, so the time needed to execute the core GP algorithm is important. Because there is always a finite amount of time in which to carry out experiments, a problem that takes a long time to run means there is less opportunity to modify a GP system. Speeding up the core GP algorithm should allow more time to find a set of attributes that performs adequately. Therefore, there is a real incentive to reduce the run time of a GP system. Implementing

GP in hardware is a way of tackling the problem of long run times, and one that has to date been rather less well explored than software only GP systems.

# Chapter 3

# FPGAs and Hardware Compilation

The previous chapter highlighted two aspects of using *Genetic Programming* (GP). Firstly, some problems take a long time to execute because the fitness evaluation is computationally expensive. Secondly, running the GP algorithm is part of a process of experimentation and refinement called Meta-GP, possibly requiring many GP runs. For some problems, a long running time for GP is a barrier to exploring and exploiting GP. Therefore, a means of accelerating GP by implementing GP in hardware should help in the exploration and exploitation of GP.

This chapter reviews the hardware and software technologies that are available for implementing GP in hardware. First, the technology of *Field Programmable Gate Array* (FPGA) devices is examined and then the role of FPGAs in evolutionary computing is reviewed. This is followed by a brief overview of high level language to hardware compilation tools. Finally, one particular high level language to hardware compilation system called Handel-C is described in more detail.

## 3.1 Introduction

The survey of GP in Chapter 2 identified two technologies that have been used to implement an evolutionary computing device; a custom *Application Specific Integrated Circuit* (ASIC) and uncommitted reconfigurable FPGAs. Both technologies are sometimes referred to as *Very Large Scale Integration* (VLSI) devices.

### 3.1.1 ASIC Implementations

An ASIC is a custom chip, designed for a particular application. They are often designed using pre-defined libraries of functionality. The work by Koizumi *et al.* [Koizumi 01] described a *Reduced Instruction Set Computer* (RISC) processor designed for *Genetic Algorithm* (GA) that is essentially a general purpose CPU, but with the addition of special instructions designed to speed up the execution of a GA program. The GA program for this device is written using a standard programming language. The program is then compiled using a modified compiler that uses the special instructions. The instructions chosen were those found to be beneficial during the crossover and mutation portions of the GA and the *Random Number Generator* (RNG). They report a three-fold improvement with their optimised instruction set when compared with an equivalent RISC processor without the specialised instructions. However, in GP the time needed to evaluate the fitness of the programs often outweighs the time for crossover and mutation, so it is questionable whether this approach would produce any great improvement if applied to GP. Other instances of VLSI technology for GAs include a proposed VLSI architecture by Turton *et al.* [Turton 94] and a general purpose, problem independent parallel GA in VLSI by Yoshida *et al.* [Yoshida 01].

Current ASIC technology using *Complementary Metal Oxide Semiconductor* (CMOS) processes and standard cell libraries allows designs to be clocked at near GHz rates, so the potential for performance improvement is good. Nevertheless, the economics of using an ASIC implementation need to be examined carefully. Designing an ASIC chip, even using standard libraries, is a specialised task, normally requiring several man-years of effort. The cost of fabricating an ASIC chip is in the order of \$500 000(1) for a prototype run. Therefore, it is seldom economic to use ASIC technologies for small scale runs. It must also be noted that once it has been committed to production, changes to the design are prohibitively expensive. Because of the high initial costs associated with a custom ASIC chip, and the inflexibility of an ASIC once it has been produced, reconfigurable devices such as FPGAs and *Complex Programmable Logic Device* (CPLD)s have become popular as platforms for experimentation and prototyping, though only FPGAs have featured in previously reported work on evolutionary computation.

(1)Obtained using the ASIC cost estimator from Altera, at `http://www.altera.com/products/devices/cost/cst-cost_step1.jsp`, assuming a 474 K gate device using a 0.18 $\mu$m process.

## 3.2 Field Programmable Gate Arrays

FPGAs are a class of programmable hardware devices in which the logic is initially uncommitted. The uncommitted logic is configured by the end user for a particular task. A common architecture is based on an array of *Configurable Logic Block* (CLB)s, *Input Output Block* (IOB)s that connect the logic to the outside world and configurable interconnections that connect the CLBs to each other and the IOBs. In the Xilinx [Xilinx 01b] Virtex device used in this work, each CLB contains two slices, each slice containing two *Logic Cells*. In addition some devices contain on-chip *Random Access Memory* (RAM). A simplified general model of an FPGA is shown in Figure 3.1.

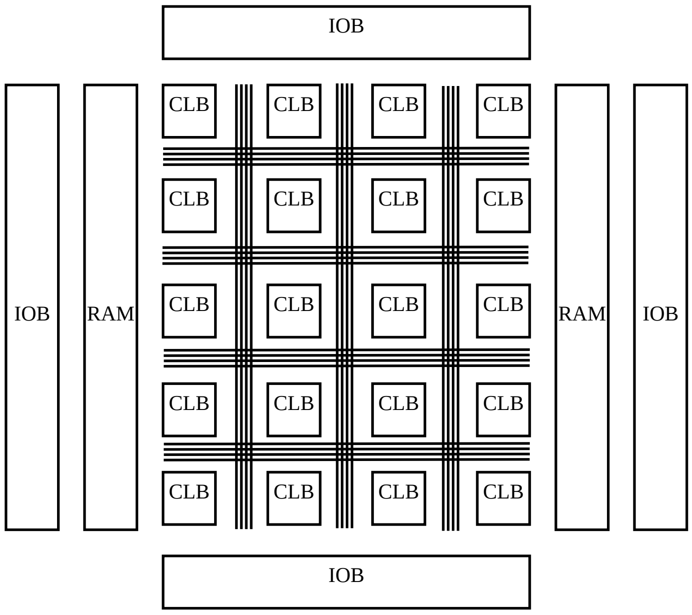


**Figure 3.1:** General model of an FPGA. It consists of an array of *Configurable Logic Blocks* (CLBs), *Input Output blocks* (IOBs) that connect the logic to the outside world and configurable interconnections that connect the CLBs to each other and the IOBs. Some devices also contain on-chip RAM.

Figure 3.2 shows a general model of a Xilinx Virtex Slice containing two Logic Cells. Each Logic Cell consists of a four input function generator implemented as a *Look up Table* (LUT), a storage element or *Flip Flop* (FF) and internal *Carry and Control* (CC) logic. Some devices also contain uncommitted RAM. The chip contains a variety of routing resources for both localised and global chip level routing of signals.

These devices are configured by loading a configuration bit pattern, which in the Virtex device is loaded into static RAM on the chip. Because the RAM is volatile, this has to be done each time the chip is re-powered. For many devices available today, the configuration bit patterns are proprietary and are generated using software tools that take a high level description of the configuration information. Keeping the configuration bit patterns secret has two important consequences. Firstly, it protects the details of the device configuration from being interpreted, and therefore makes it hard to reverse engineer the device. This is important for the commercial adoption of these devices. The second consequence is a limiting one in that only the vendor's specialised tools can be used to configure the device, making it hard to dynamically reconfigure the devices.

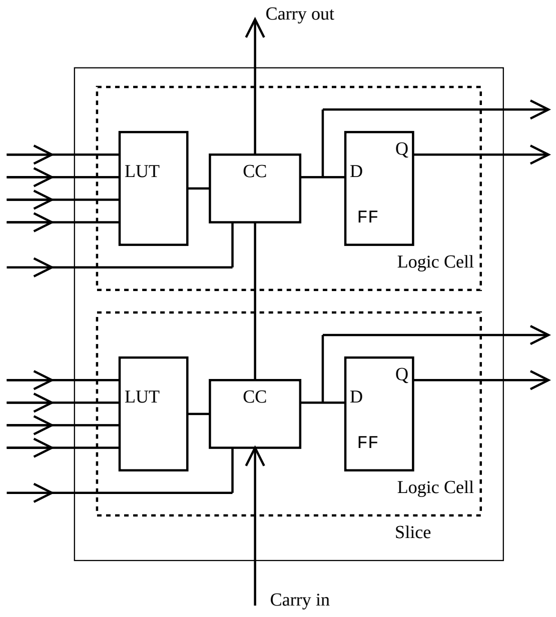


**Figure 3.2:** General model of a Configurable Logic Block or Slice. Each Slice contains two Logic Cells. Each Logic Cell consists of a function generator implemented as a *Look up Table* (LUT) a storage element or *Flip Flop* (FF) and internal *Carry and Control* (CC).

### 3.2.1 Reconfigurability

An early device – the XC6200 series from Xilinx – had an architecture that allowed any arbitrary configuration pattern to be loaded into the device without the possibility of damaging the device. This was of great interest to the evolutionary hardware community because it did not constrain how the devices could be used, and this device was used by many researchers in the field of evolutionary computing. Because the configuration was also accessible to the logic on the FPGA, the FPGA could dynamically reconfigure itself. Unfortunately, the Xilinx XC6200 series did not find commercial use and is no longer in production [Alfke 00]. However, more recent devices have an architecture that supports many more interconnections between the logic cells and so do not allow arbitrary configurations since they could damage the chip. The possibility of damage arises because of the potential for two or more gate outputs to be directly wired together, as illustrated in Figure 3.3. If the output of logic block A is driven to a logic high and the output of logic block B to a logic low, a large current would flow between the output stages via the routing resources. If many of these high current paths are created, then the power dissipated by the device may exceed the safe limits specified by the manufacturer, possibly destroying the device. There is some anecdotal evidence that devices have been spectacularly destroyed by this process[2].

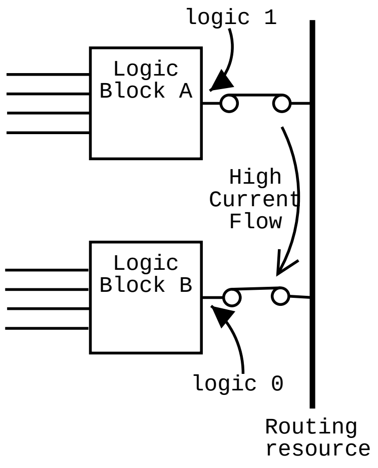


**Figure 3.3:** Consequences of illegal FPGA gate configuration.

Because current FPGAs cannot accept arbitrary configurations there are two consequences for the use of standard FPGA devices for evolutionary computation. Firstly, during the creation of bit patterns for these devices, checks need to be made to eliminate illegal configurations. Secondly, it hampers the possibility of dynamically reconfiguring the device while it is executing an algorithm. To some extent these limitations are being addressed by the device manufacturers. For example, Xilinx provides a Java *Application Programming Interface* (API) called JBits that allows partial reconfiguration of some devices. JBits has been used by Levi and Guccione to evolve stable circuits on standard Xilinx devices [Levi 99]. Hollingworth *et al.* [Hollingworth 00] have also used this technique for intrinsic hardware evolution. Another approach to solving the reconfigurability problem is to synthesise a virtual configurable chip using a standard FPGA that has no restrictions on the configurations [Sekanina 00].

[2]Discussion between the author and an FPGA applications engineer, and postings to comp.fpga.arch

## 3.3 FPGAs and Evolutionary Computing

Chapter 2 cited three examples of using an FPGA to implement part or all of a GP system ( [Heywood 00], [Sidhu 99] and [Koza 97b]). In this section a more detailed review of FPGAs that have featured in the broader field of evolutionary computing is given.

In Evolutionary Computing, FPGAs have previously been used in three main areas: 1) as a means of implementing the fitness functions of Genetic Algorithms or Genetic Programming; 2) as a platform for implementing the Genetic or Evolutionary Algorithm; 3) in relation to evolving hardware using evolutionary techniques. These three strands are surveyed separately. A common theme running through previous work is the use of traditional hardware design tools and languages such as Verilog or *VHSIC Hardware Design Language* (VHDL) to design the logic for the FPGAs.

### 3.3.1 FPGAs for Speeding Up Fitness Evaluations.

In this category only the fitness evaluation is performed by an FPGA. The creation of the initial population and the breeding phases are carried out by a host computer.

Koza *et al.* [Koza 97b] used an FPGA to speed up the evaluation of fitness of a sorting network where the FPGA was used solely to perform the fitness evaluation. The initial population was created by a host computer, and then individuals were downloaded to a pre-programmed FPGA and the FPGA instructed to evaluate the fitness of the individual. Subsequent selection and breeding were again performed by the host computer. The configuration of the FPGA remained unchanged during the experiment. This work used a Xilinx XC6200 device.

Montana *et al.* [Montana 98] developed a hybrid tool called EvolvaWare (see also [BBN Technologies 02]) which was designed to generate VHDL circuits using GP. This tool implemented a number of primitives (functions) on an FPGA. It then used standard GP to evolve programs that used the FPGA hosted primitives. Although this approach showed some promise, they discovered that the hybrid software/hardware system had some problems. Notably, the interface between the software and the hardware was a bottleneck due to the limited number of input/output pins. They also discovered that the FPGA technology available in 1998 would not allow complex primitives to be accommodated on the FPGA. They did not report what FPGA was used for this work.

Yamaguchi *et al.* [Yamaguchi 00] used a Xilinx Virtex XCV1000 FPGA to implement a co-processor for evolutionary computation to solve the iterated prisoners dilemma (IPD) problem. They reported a 200 times performance speedup in processing the IPD functions on the FPGA when compared to a 750 MHz Pentium processor.

Seok *et al.* [Seok 00][3] used a Xilinx XC6200 series device to implement a robot controller. The evolution of the FPGA was performed off line and the FPGAs were embedded in a robot.

Harris [Harris 00] used a GA to evolve a 7-input parity function using an FPGA simulation.

### 3.3.2 Implementing the Logic For Evolution Using FPGAs

In this category the fitness evaluation and breeding, and in some cases the initial population creation, is carried out on the FPGA.

Scott [Scott 94] implemented HGA: A Hardware-based Genetic Algorithm. The design was modelled on an existing software GA by Goldberg called SGA [Smith 91] which allowed them to make meaningful comparisons of the performance between a software and hardware implementation of similar algorithms. The design allowed limited parameterisation of the population size, number of generations PRNG seed and the mutation/crossover proportions. Scott used a three stage coarse grained pipeline in which the selection module, crossover/mutation module and the fitness module were all active at the same time. In addition, the design allowed for the pipeline to be replicated so increasing the potential for speeding up the GA. The prototype was realised using on Xilinx XC4003 FPGA for the PRNG and crossover/mutation module, and three XC4005 devices that housed the fitness function, selection and memory interface and the population sequencer. The prototype was built using wire wrap technology which, because of signal distortion and crosstalk between signals, limited the clock rate to 2 MHz, although the simulation had indicated that 8 MHz should be possible. The performance of the HGA was compared to the SGA running on a workstation with four MIPS R3000 CPUs running at 33 MHz. This produced a speedup of approximately 16 times when the number of clock cycles was measured, but no speed up when the wall clock time was used. However, Scott predicted that if the design was optimised using a PCB layout and that further paralellisations were added that the design could be clocked at between 30 and 40 MHz giving a speedup of two orders of magnitude over the SGA running on the MIPS computer.

Graham and Nelson [Graham 96] implemented a complete GA system using four Xilinx XC4010 FPGAs on the Splash-2 board. Each FPGA was programmed to carry out a different function; selection, crossover, fitness and mutation and finally statistics. Each FPGA passed its results to the next

[3]Additional implementation details obtained directly from the author via email.

forming a pipeline. The performance of their system was compared with a software implementation running on a 125 MHz PA-RISC workstation and they showed an improvement of 4 times.

Kitaura *et al.* [Kitaura 99] implemented a GA machine that was designed to avoid pipeline stalls using a Lucent ATT2C40 FPGA. They report a speedup of 730 times when compared with an equivalent software system.

Tufte and Haddow [Tufte 99] implemented a complete evolutionary hardware system based on a GA – called Complete Hardware Evolution (CHE) – on a Xilinx XC4044XL FPGA that used a pipeline to evolve hardware. They also demonstrate CHE using a robot controller in [Haddow 99].

Choi and Chung [Choi 00] implemented a steady state GA using two Altera EPF10K100A devices.

Perkins *et al.* [Perkins 00] describe a system where a complete GA system was realised on a single Virtex 300 part. Performance was compared with a C implementation, and they report an improvement of over 1000 times, though they do not specify the speed of the CPU used for the C implementation.

Aporntewan and Chongstitvatana [Aporntewan 01] proposed a hardware implementation of a GA using a Xilinx Virtex XCV1000 FPGA that achieved a 1000 time speedup when compared with a 200 MHz SPARC 2 workstation.

Shackleford *et al.* [Shackleford 01] implemented a complete GA system using a Xilinx XCV3200E chip. Their implementation used extensive pipelines and parallel fitness evaluation to get a performance increase of 320 times when compared with the same algorithm running on a 366 MHz Pentium CPU.

Finally in this category, FPGAs have been used to implement parts of a GP system. Sidhu *et al.* [Sidhu 99] presented a novel approach, that used self-reconfiguration of the FPGA, to implement the fitness evaluation and evolution phases of GP. The initial population was configured statically. They used a Xilinx XC6264 device, and compared the performance with lilgp running on a 200 MHz Pentium. They reported a speedup of over 1000 for the Boolean 11 multiplexer problem implemented using lilgp [Zongker 02], though it is questionable whether a direct comparison to lilgp, which uses a different internal representation, gives a true picture of the real speedup.

The system described by Heywood and Zincir-Heywood [Heywood 00] was simulated using traditional FPGA tools. Their proposal used the FPGA for evaluating the individuals and performing mutation and crossover. Initial population creation was done off-line and downloaded to RAM for use by the FPGA.

It is important to note that none of the previously reported work using FPGAs and GP has implemented a complete GP system, that is the initial population creation, fitness evaluation, evolutionary operators and the production of a program at the end of the run.

### 3.3.3 Evolutionary Hardware Using FPGAs

FPGAs have featured regularly as platforms for evolutionary hardware research. Higuchi *et al.* [Higuchi 93] showed the feasibility of hardware evolution using *Hardware Design Language* (HDL) to evolve combinatorial logic circuits. They used GP to evolve the HDL.

Thompson suggested that FPGAs could be used to evolve hardware circuits [Thompson 95]. This was demonstrated using a simulation of the evolution of an FPGA configuration to evolve an oscillator. Thompson [Thompson 96] also demonstrated, for the first time, a physical FPGA being used as the target for evolutionary hardware. His work is interesting for several reasons: Firstly, it used an FPGA – the Xilinx XC6200 – that supported direct reconfiguration of its logic cells. Secondly, his work relied on the asynchronous behaviour of the FPGA to obtain the results, in contrast to much of the current work using FPGAs which is very firmly focussed on the synchronous use of FPGAs. Thirdly, the evolutionary approach discovered an analogue behaviour of the FPGA that resulted in the circuit operating correctly, but only within a limited temperature range. Subsequent work by Thompson and Layzell [Thompson 99] describes the physics of this behaviour. Thompson also demonstrated how a miniature Khepera robot, fitted with a Xilinx XC6216 FPGA, could evolve wall-avoiding behaviour [Thompson 97]. In this case the evolutionary algorithm was executed on an external PC connected using a serial cable.

The work by Fogarty *et al.* [Fogarty 98] described how logic circuits could be evolved directly on a Xilinx XC6000 FPGA without having to place and route a netlist(4) first. Levi and Guccione [Levi 99] described a method of generating the FPGA configuration data for Xilinx XC4000EX and XC4000XL devices that avoids illegal FPGA configurations and ensures the FPGAs are stable.

Kajitani *et al.* [Kajitani 98] described a simulation of an evolvable hardware system using a single LSI chip that contained the GA operations as well as a *Programmable Logic Array* (PLA).

(4)A netlist is a formal description of the gates and their connections.

This work uses traditional GA techniques to evolve the circuits which are executed on the integrated PLAs meaning that the chip can autonomously synthesise hardware circuits.

Jamro and Wiatr [Jamro 01] investigated the use of GP to evolve optimised adders for use in image convolution on FPGAs. Once the adders had been evolved the tool generated optimised VHDL for the required convolver. However, they discovered that *Simulated Annealing* (SA) performed better than GP in this application.

## 3.4 Configuring FPGAs and Hardware Compilation

FPGAs have traditionally been configured by hardware engineers using a *Hardware Design Language* (HDL). The two principal languages being used in 2002 are Verilog [Verilog 01] and VHDL [VHDL 93]. Both languages are similar, and the choice of which one to use is often based on personal preferences, tool availability, or business and commercial issues [Smith 97]. VHDL has been used to implement hardware GAs and GP ( [Scott 94], [Graham 96],, [Perkins 00], [Heywood 00], and [Shackleford 01]). Both languages support a behavioural or algorithmic modelling capability, but the majority of tools cannot translate a behavioural model directly into hardware. Instead, the designer must create a structural model that maps to the hardware. Verilog and VHDL are specialised design techniques that are not immediately accessible to software engineers, who have often been trained using imperative programming languages. Consequently, over the last few years there have been several attempts at translating algorithmic oriented programming languages directly into hardware descriptions. These have taken a number of different approaches to this problem and these are briefly reviewed in this section.

Synopsis [Synopsis 02] produce a product called System Level Design that models the hardware using a set of C++ classes. Frontier Design [A|RT 02] produce a product called A|RT (Algorithm to Register Transfer) which allows the designer to use a C style language to express the algorithm. However, the designer of the C code is forced to use a particular coding style that uses register transfer semantics. CynApps [CynApps 02] produce a tool called Cynthesis in which the designer needs to re-write algorithmic C code to reflect the hardware architectural view.

Another approach is to use a high level language to target part of the system implemented in an FPGA, typically as a co-processor. Clark [Clark 96] used the freely available C compiler lcc [Fraser 95] to compile specific instructions that targeted pre-compiled hardware functions in a co-processor. Callahan and Wawryzynek [Callahan 98] also used a C compiler to target a custom processor built using FPGAs. Wirth [Wirth 98] suggested how parts of a C program could be compiled into programmable hardware.

A third method is to include the algorithm to hardware mapping in the compilation process. This approach does not require substantial changes to the underlying algorithm when compared with the previous approaches. The transmogrifier work by Galloway [Galloway 95] used this approach. Maruyama and Hoshino [Maruyama 00], demonstrated how loops and recursion could be implemented directly. Page [Page 91] [Page 96] recognised the need for a formalism to map between algorithmic constructs and hardware. The early work used *Communicating Sequential Processes* (CSP) and Occam. However, because Occam was regarded as an academic language, the work was extended to use a C like language, called Handel-C, thereby making it more appealing to industrial users. Handel-C is now a commercial product produced by Celoxica [Celoxica Ltd 01e].

Because Handel-C uses a C like language and because it does not force the designer to adopt a structural approach to coding, it is accessible to software engineers. For these reasons Handel-C was used for implementing GP on an FPGA. The next section describes Handel-C in greater detail.

## 3.5 Handel-C

This section is not intended to be a full description of the tool and language, but it does describe the most important features, especially those that influence the design decisions described later in this thesis. For full details of the language and development environment the reader is referred to the user guides and reference material from the manufactures [Celoxica Ltd 01a]. Handel-C has been used in a number of applications including the rapid implementation of a RISC microprocessor core [Sulik 00], the implementation of video algorithms [Page 97] and an SS7 transceiver [Marconi 00]. Other examples of applications can be found in [Celoxica 02].

### 3.5.1 Parallel Hardware Generation

One of the advantages of using hardware is the ability to exploit parallelism directly. This is in contrast to the simulated software parallelism that is found on single *Central Processing Unit* (CPU) computers achieved using time-slicing. Note that although modern processors found in standard PCs and workstations have multiple pipelines that attempt to execute more than one instruction simultaneously, they are only able to process one instruction sequence at a time. Consequently, Handel-C has additional constructs to support the parallelisation of code using the **`par`** statement. For example, the block

```
par {
    a=10;
    b=20;
}
```

generates hardware to assign the value 10 to `a` and 20 to `b` in a single clock cycle. Using this statement, large blocks of functionality can be generated that execute in parallel. Hardware can be replicated using the construct

```
par (i=0;i<10;i++) {
    a[i] = b[i];
}
```

which results in 10 parallel assignment operations. Multiple copies of functions may be created by using the macro language of Handel-C, by using the **`inline`** keyword which instantiates a copy of a function every time it is referenced or by explicitly defining an array of functions as in the following fragment that defines 32 instances of a function:

```
void evaluate[32](unsigned int population)
{
    ...
}
```

### 3.5.2 Efficient Use of FPGA Resources

To make efficient use of the hardware, Handel-C requires the programmer to declare the width of all data, for example,

```
int 5 count;
```

is a signed integer that is 5 bits wide, and so will be able to represent the values $-16 \leq \mathit{count} \leq +15$. Handel-C has a single native Integer data type, but fixed and floating point data types are supported by means of a library.

### 3.5.3 Bit Level Operators

Handel-C provides a number of bit manipulation operators that are not available in ISO-C. The following bit operators are provided:

| | |
|---|---|
| `y = x <- n` | Takes the *n* least significant bits from `x` |
| `y = x \\ n` | Drops the *n* least significant bits from `x` |
| `y = x @ z` | Concatenates the bit patterns that represent `x` and `z` |
| `y = x[5:3]` | Selects bits 3,4 and 5 from `x` |

The **`width`** operator returns the number of bits in an expression.

### 3.5.4 Simple Timing Semantics

According to the Handel-C documentation, the simple rule about timing of statements is that "assignment takes 1 clock cycle, the rest is free". In practice this means that expressions are constructed using combinatorial logic, and data is clocked into a register only when an assignment is performed. For example, Handel-C would generate hardware for the following statement that executed in a single clock cycle.

```
y = ((x*x)+3*x);
```

This feature makes it straightforward to predict the number of clock cycles a section of code will require. However, the more complex the expression, the deeper the logic required to implement the expression. This in turn limits the maximum clock rate at which the design can be run because of the propagation delays associated with deep logic. In practice this means that the designer needs to trade clock cycles against clock rate, and this is typically an iterative process.

### 3.5.5 External Communication

Communication between the hardware and the outside world is performed using interfaces. These may be specified as input or output, and, as with assignment, a write-to or a read-from an interface will take one clock cycle. The language allows the designer to target particular hardware, assign input and output pins, specify the timing of signals, and generally control the low level hardware interfacing details. Macros are available to help target particular devices.

### 3.5.6 Some Restrictions When Using Handel-C and FPGAs

Because Handel-C targets hardware, it imposes some programming restrictions when compared to a traditional ISO-C compiler. These need to be taken into consideration when designing code that can be compiled by Handel-C. Some of these restrictions particularly affect the building of a GP system.

Firstly, there is no stack available, so recursive functions cannot be directly supported by the language. This in turn means that standard tree based GP, which relies heavily on recursion, cannot be implemented without some modification. A solution to this restriction is discussed in Section 4.1.1.

Secondly, the size of memory that can be implemented using standard logic cells on an FPGA is limited, because implementing memory is an inefficient use of FPGA resources. However, some FPGAs have internal RAM that can be used by Handel-C. For example, the Xilinx Virtex and Spartan series support internal memory that Handel-C allows the user to declare as RAM or *Read Only Memory* (ROM). For example the definition

```
ram int 8 mem[128];
```

declares a RAM block of 128 cells, each 8 bits wide, which can be accessed as a standard array.

A limitation of using RAM or ROM is that it cannot be accessed more than once per clock cycle. This restricts the potential for parallel execution of code that accesses RAM or ROM.

Thirdly, expressions are not allowed to have side effects, since this would break the single cycle assignment rule. Therefore code such as

```
a = ++b;
```

is not allowed and needs to be re-written as:

```
b = b + 1;
a = b;
```

### 3.5.7 Targets Supported by Handel-C

Handel-C supports two targets. The first is a simulator target that allows development and testing of code without the need to use any hardware. This is supported by a debugger and other tools. The second target is the synthesis of a netlist for input to place and route(5) tools. This allows the

(5)Place and route is the process of translating a netlist into a hardware layout.

design to be translated into configuration data for particular chips. An overview of the process is shown in Figure 3.4. When compiling the design for a hardware target, Handel-C emits the design in *Electronic Design Interchange Format* (EDIF) format. A cycle count is available from the simulator, and an estimate of gate count is generated by the Handel-C compiler. To get definitive timing information and actual hardware usage, the place and route tools need to be invoked.

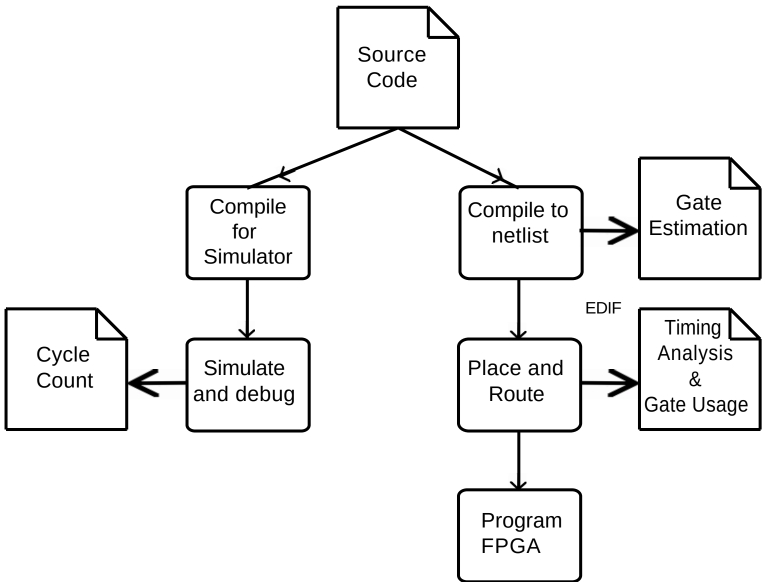


**Figure 3.4:** Overview of the process of translating code into hardware using Handel-C and the critical outputs for analysis of the solution

### 3.5.8 Translating ANSI-C to Handel-C and Code Portability

With care it is possible to re-use the same code for a design implemented in hardware, and a design realised as a traditional software program. The differences in syntax and the various extensions can be made portable using the C pre-processor. For example, the need to supply a width specifier and `par` block statements for Handel-C can be hidden from an ANSI C compiler as follows:

```
#if defined HANDELC
typedef unsigned int 5 Uint_5;
#define PAR par
#else
typedef unsigned int Uint_5;
#define PAR
#endif
...
Uint_5  x;
...
PAR {
    // parallel statements
}
```

The issue of code portability is of interest when it comes to testing new ideas. The time required by Handel-C to compile an algorithm for an FPGA can be measured in tens of minutes to many hours, depending on the complexity of the algorithm and the level of optimisation specified. In contrast, a standard ISO-C compiler on a 1.4 GHz Athlon based PC can recompile an application in the lilgp suite [Zongker 02] in a little over 10 seconds. Clearly, testing an algorithm is much easier and more productive using standard programming techniques. Having proved the algorithm, it is usually advantageous to reuse the algorithm directly in the hardware implementation, thereby reducing the potential for errors in transcription or translation that would occur if the two environments were different. This issue is examined in more detail in Chapter 7 where the economic and software engineering issues are discussed.

## 3.6 Summary

This chapter has surveyed two technologies that have previously been used to implement GAs and GP in hardware. Because ASICs are usually considered to be faster than FPGAs, a custom VLSI implementation using ASIC technology would appear to be highly desirable, but because of the cost implications of using a custom designed VLSI chip, the use of uncommitted reconfigurable devices is a preferred route for experimentation and prototyping.

Because traditional methods of programming FPGAs require a structural approach and because they use specialised languages, the direct use of FPGAs has been inaccessible to software engineers. To remedy this, high level language to hardware design tools have been developed. The use of these tools is appealing to the software engineering community because it allows a traditional imperative programming style to be used to design and implement algorithms in hardware. Handel-C is one of the high level languages available. It provides an algorithmic route to hardware design, allows code to be ported between software and hardware environments and enables the parallelism inherent in hardware to be exploited. Handel-C has an established track record, having been used to implement other algorithms. For these reasons Handel-C was the tool chosen for the implementation of GP in hardware.

# Chapter 4

# Implementing GP in Hardware

This chapter describes the general design decisions taken to implement *Genetic Programming* (GP) in hardware using a *Field Programmable Gate Array* (FPGA) as the platform, and Handel-C as the software environment used to program the FPGA.

## 4.1 A Complete GP System On a Chip

The principal aim of the work described in this chapter was to explore the feasibility of implementing a complete GP system in hardware, that is initial population generation, fitness evaluation, breeding and the delivery of the final result. This is in contrast to all other examples of using FPGAs with Genetic Programming reviewed in Chapter 3, Section 3.3.2, which only implemented the fitness evaluation and possibly some of the other stages of GP. This high level aim guided many of the design decisions. As the aim was to explore the feasibility of using Handel-C and FPGAs, the work focused on the functionality of the GP system. Consequently, it was accepted that there were probably going to be limitations imposed by Handel-C and the use of FPGAs with regard to program and population sizes. It was also recognised that some compromises would be need to be made with regard to the throughput of the system. In the light of the experience gained during this work a number of improvements were identified to increase the program and population sizes and increase the throughput. These improvements are described in Chapter 5.

### 4.1.1 Internal Program Representation

Tree based GP normally uses recursion to traverse the tree, but the lack of a built-in stack when using Handel-C precludes the use of recursion. Although there are well known methods of removing recursion from algorithms (eg. [Sedgewick 84]), a stack of some form is still required to store intermediate results. Tree representations also have a storage overhead because of the need to store the links from a node to its children. Because FPGAs have a limited amount of storage available when compared with a modern computer that has many megabytes of RAM, the use of tree based GP was considered to be too expensive in terms of silicon real estate. However, the survey of GP in Chapter 2, Section 2.4.4, showed that internal representations other than trees have been used for GP. One alternative to the standard tree representation is the linear GP system, examples of which were used by Nordin and Banzhaf [Nordin 95b], Banzhaf *et al.* [Banzhaf 98] and others, in which a linear string of machine code instructions was executed directly by a *Reduced Instruction Set Computer* (RISC) CPU. Subsequent work extended this to *Complex Instruction Set Computer* (CISC) CPUs [Nordin 99a] and [Kuhling 02]. Other examples of linear GP include Atkin and Cohen [Atkin 94], Bhattacharya *et al.* [Bhattacharya 01] and Deschain *et al.* [Deschain 01].

A linear structure was chosen for implementing GP in an FPGA because the representation is simple to implement, and requires very little overhead for representing a program. A further argument for using a linear structure is that it has been shown to be able to solve hard problems. Some examples of the use of a linear structure for hard problems include the evolution of a hand-eye controller for a robot [Langdon 01], data mining [Deschain 00] and real-world design simulation [Deschain 01]. The internal program representation is a linear string of nodes which can be thought of as primitive instructions. The details of the internal representation depend on the word size, number of functions and number of terminals used, and these are dependent on the problem being tackled.

A general purpose load-store register architecture [Patterson 96] was chosen for its simplicity and because it is similar to the work done by Nordin and others using RISC instruction sets, though a load-store register machine is by no means the only machine that could be used. Alternatives to a general purpose register machine include single register or accumulator machines, register (register-memory) and memory-memory machines in which the instructions address memory directly without the use of registers . If the storage concerns can be addressed then stack machines could also be used.

Other possibilities include implementing a high level language virtual machine in hardware, such as a Forth language machine [Moore 70], [Moore 01] and [Koopman Jr. 89], the *p-code* machine [Clark 81] used by the Pascal language, or even a Java virtual machine [Lindholm 99]. GP using the Java virtual machine has been investigated by Lukschandl *et al.* [Lukschandl 98] and Klahold *et al.* [Klahold 98]. The ability to tailor the machine is explored later in this chapter (Section 4.9.4 on page 81) and is exploited in the next chapter when the artificial ant problem is implemented in hardware.

A program consists of an array of instructions and some control information. The programs have a fixed maximum size, which simplifies the storage of individuals. A general layout of an instruction is shown in Figure 4.1.

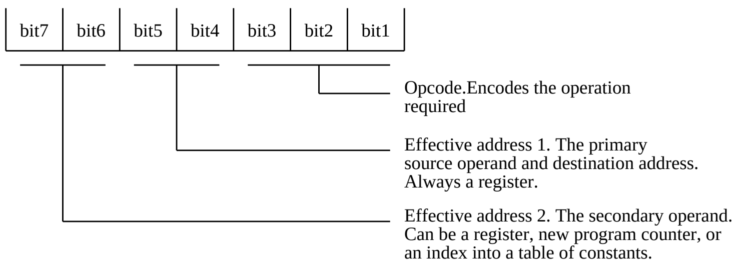

bit7
bit6
bit5
bit4
bit3
bit2
bit1
Opcode.Encodes the operation required
Effective address 1. The primary source operand and destination address. Always a register.
Effective address 2. The secondary operand. Can be a register, new program counter, or an index into a table of constants.

**Figure 4.1:** Layout of an instruction where there are eight possible opcodes and two effective addresses: EA1 and EA2. The details of what the opcodes do and the use of the effective addresses is problem specific.

The layout shows an example in which there are eight possible opcodes. Each opcode can use zero, one or two effective addresses. All instructions are the same length which simplifies the design of the machine. The use of fixed length instructions also makes the crossover operator straightforward to implement when compared with variable length instructions as found in CISC machines [Nordin 99a]. The details of what the opcodes do and the effective addresses is problem specific and the number of functions and effective addresses will vary according to the problem.

The control information for a program consists of the program length and the raw fitness of the program.

## 4.2 Parallelism

When discussing parallelism it is important to distinguish between different forms. Here four types of parallelism are used; *intrinsic*, *geometric*, *algorithmic* and *asynchronous*. These will now be explained.

Firstly, the Handel-C language supports parallelism directly, as already discussed in Chapter 3, enabling efficient implementation of instructions that would normally be executed serially on a standard microprocessor. This, in itself, gives a substantial increase in performance when compared to a standard microprocessor. Since this form of parallelism is built into Handel-C, this is called intrinsic parallelism.

The second use of parallelism is in the implementation of the Genetic Programming algorithm. Chapter 2, Section 2.4.6, introduced parallelism as an attribute of GP. The type of parallelism found in all the examples given is geometric parallelism, where a data set is partitioned into smaller units and the processing is replicated on many processors.

A third form of parallelism – algorithmic parallelism – occurs where many tasks can be pipelined, making fuller use of the available resources. Pipelining is common in hardware design and is found in most modern microprocessors.

Lastly, a form of parallelism called asynchronous or *relaxed parallelism* occurs when two or more tasks communicate on an occasional basis but operate independently, without any explicit synchronisation.

### 4.2.1 Intrinsic Parallelism for a Hardware Implementation

Where possible the design exploits parallel execution of all simple statements. This is done regardless of the phase of GP (creation, fitness evaluation, selection and breeding) since there is no penalty in executing two assignments in parallel. An example of intrinsic parallelism is the initialisation of variables at the beginning of a function.

### 4.2.2 Geometric Parallelism for a Hardware Implementation

The type of geometric parallelism used is master-slave (Section 2.4.6), where the master stores the population and a number of slaves evaluate the fitness of the individuals. Geometric parallelism is a natural fit where the population is a global resource within the FPGA, or closely coupled RAM, and parallel fitness evaluations can be realised by replicating the fitness evaluating hardware. Because the entire system is realised on a single chip, the communication overhead between the master and the slaves (the evaluation functions), which is normally regarded as a bottleneck, is greatly reduced. The time needed for communication between the master and the slaves is effectively the propagation time for a signal from one part of the FPGA to another.

Since it is unlikely that there would be sufficient FPGA resources to be able to evaluate an entire population at once, the population is divided into a number of smaller subsets and each subset is evaluated in parallel. To simplify the logic, both the total population size and the number of individuals in a subset is a power of 2. Parallelisation of the evaluation is implemented by replicating the fitness function. This is achieved by using the **`inline`** keyword in Handel-C, which causes as many copies of the hardware to be generated as required.

### 4.2.3 Algorithmic Parallelism for a Hardware Implementation

Pipelines have not been used in this initial feasibility study, though the opportunity for using them to speed up the design is clear, as shown in some of the previously reported work using hardware to implement GP and GAs, for example, [Scott 94] [Graham 96] [Kitaura 99] and others. The use of algorithmic parallelism is explored in more detail in Chapter 5.

### 4.2.4 Asynchronous Parallelism for a Hardware Implementation

There is one task that is ideally suited to an asynchronous implementation - that of the *Random Number Generator* (RNG). The RNG runs continuously in parallel with everything else, generating a stream of random numbers which are used, as needed, by the rest of the design.

## 4.3 Random Number Generator.

An RNG, or more correctly a *Pseudo Random Number Generator* (PRNG), is used in two of the major steps in GP. Firstly, during initial population creation, to create a diverse population, and secondly, during the breeding phase, to select individuals for breeding and to choose a particular breeding operator from one of crossover, mutation or copy. One of the most common PRNGs is a Linear Congruential generator, often found as part of a computer language run-time library. These use the multiply and divide operators and are discussed in more detail in Section 6.3. When using

Handel-C, the use of the standard multiply and divide instructions is inefficient in terms of silicon because of the deep logic generated for the single cycle combinatorial circuits. Although the problem with Handel-C and the multiply/divide logic may in the future be eased by the appearance of FPGAs with built-in multipliers, because implementing multiply and divide logic using a Xilinx XCV2000 FPGA requires a large amount of logic, the usual linear congruential generators normally found were rejected. Instead, a *Linear Feedback Shift Register* (LFSR) design was used as suggested by the FPGA manufacturers [Altera 01] and [Xilinx 01a]. A word size of 32 was chosen, as this could be implemented efficiently on a standard modern CPU, and so the LFSR could also be implemented using ISO-C. Implementing the LFSR in ISO-C was of interest during the development of the GP algorithm (see Section 4.9.1 on page 79). It is important to choose a good polynomial to ensure that the RNG can generate a maximal sequence of $2^n - 1$ random numbers, while keeping the number of taps to a minimum for efficiency. Schneier [Schneier 96, page 376] gives a list of such polynomials. For a 32 bit word the polynomial $x^{32} + x^7 + x^6 + x^2 + x^0$ was used. The block diagram of the LFSR is shown in Figure 4.2. Only 4 taps are shown since $x^0$ is always 1.

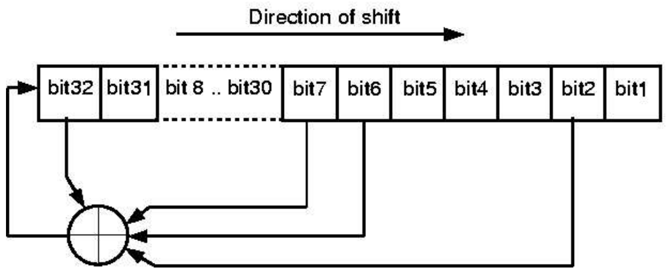


**Figure 4.2:** *Linear Feedback Shift Register* (LFSR) random number generator for the 32 bit polynomial $x^{32}+x^7+x^6+x^2+x^0$. The $\oplus$ symbol is the 4 input logical exclusive OR function (XOR). Only 4 taps are shown since $x^0$ is always 1.

The RNG is designed so that a random number is generated in one cycle. The required number of bits are then read from the 32 bit register, starting at bit 32 to give a random number. For example, if the system has 8 instructions, then 3 bits are needed to encode the instruction. During initial program creation, the random selection of an instruction uses the top 3 bits. Handel-C allows efficient bit operations and the code to select the 3 bits is:

```
unsigned int 3 instruction;
instruction = randReg[31:29];
```

where `randReg` is the shift register variable.

Seeding of the RNG is done by reading a 32 bit port during the initialisation phase. This allows the RNG to be seeded from an external source, such as a time of day clock or other source of noise. It also allows the RNG to be preset to a known seed for producing repeatable results.

After the design had been completed and some experiments had been run (Section 4.8), it was suggested that this RNG is not ideal, because LFSRs are known to perform poorly in the serial test described by Knuth [Knuth 69]. This prompted an analysis of the RNG, given in Chapter 6, Section 6.3, where it is shown that other RNGs can give better results.

## 4.4 Initial Program Creation

Programs are created initially by filling the array of instructions with random integers. The length of the program $l$ is then randomly chosen so that $0 < l \leq L_{\text{max}}$ where $L_{\text{max}}$ is the maximum program length. Because the opcode portion of the instruction can have any value from 0 to $2^{N_{\text{opcode}}} - 1$, where $N_{\text{opcode}}$ is the number of bits for the opcode, the function set must implement instructions for all possible opcodes. That is, there must be $2^{N_{\text{opcode}}}$ functions in $\mathcal{F}$. This decision simplifies the design of the program creation algorithm, and allows the mutation operator to be implemented efficiently. The efficiency is achieved because all bit patterns are treated equally and logic is not needed to handle special cases of missing instructions. The same principle applies to the register portion of the instruction. If there are $N_{\text{ea}}$ bits for the register addresses, there must be $2^{N_{\text{ea}}}$ registers or terminals.

## 4.5 Breeding Policy and Operators

To conserve memory, a steady state breeding policy was used. Tournament selection is used with a tournament size of two. Larger tournament sizes makes little sense with very small populations.

### 4.5.1 Mutation

The mutation operator works by replacing one instruction with a new, randomly generated, instruction. The result is that a mutation can change zero, one or more of the instruction details. This mutation operator is fairly crude and potentially destructive and further work could to be done to evaluate the effect of such a heavy handed method.

### 4.5.2 Crossover

This work maintains a fixed maximum program size and copies segments from one program to another. By exploiting the parallel nature of hardware, the effects of performing block memory copies can be reduced to an acceptable level.

The crossover operator ensures programs do not exceed the maximum program length by selecting crossover points in two individuals at random and exchanging the tail portions up to the maximum program length. Crossovers that result in programs exceeding the maximum length are truncated at the maximum length. This crossover operator was chosen to minimise the amount of logic required and the number of clock cycles needed.

For two programs $a$ and $b$, with lengths $l_a$ and $l_b$, two crossover points $x_a$ and $x_b$ are chosen at random so that $0 \leq x_a < l_a$ and $0 \leq x_b < l_b$. The program size limit is $L_{\max}$. After crossover, the new lengths are $l_{a'} = \min((x_a + l_b - x_b), L_{\max})$ and $l_{b'} = \min((x_b + l_a - x_a), L_{\max})$.

This is illustrated in Figure 4.3 for two programs, where $L_{\max} = 16$, $l_a = 9$, $l_b = 12$, $x_a = 1$ and $x_b = 11$. This shows that after crossover, program $b'$ has been truncated to length $L_{\max}$.

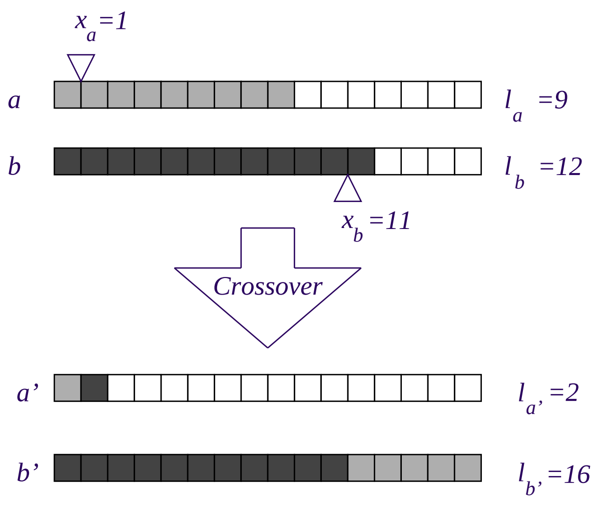


**Figure 4.3:** Truncating crossover operator

The behaviour of this crossover operator is examined in detail, and some alternative crossover operators are investigated in Chapter 6 .

### 4.5.3 Reproduction

By exploiting the parallel nature of the hardware, a copy of an individual of length $l$ requires $l+k$ clock cycles, where $k$ represents the small overhead to set up the copy, currently 3 clock cycles. It would also be possible to use more of the FPGA resources to perform reproduction in constant time by copying all the instructions in one individual in parallel.

### 4.5.4 Calculating Operator Probabilities

The operators in GP are usually selected with fixed probabilities, so an efficient means of obtaining a weighted distribution was required. Using the standard C inequality operators was rejected because of the deep logic that they produce, and the consequent loss of performance. Therefore, an alternative method was devised using bit masks. By using a bit mask and logical operators it is possible to determine if a number lies in a given range. The following example uses an 8 bit unsigned integer to determine if the value $v$ lies between 0 and 31. i.e. the low 5 bits of $v$. This represents 12.5% of the possible values of $v$.

```
bool f(unsigned int v)
{
   unsigned int mask1, v1;
   mask1 = 0xe0;
   v1 = v & mask1;   // v1 is true if v is greater than 31
   return !v1;       // returns true if v < 32
}
```

The operators were selected using the following probabilities. Mutation 12.5%, Crossover 66%, Reproduction 21.5%. These values were arrived at after experimentation.

## 4.6 Performance Comparison Methodology

As already noted, there are potentially four types of parallelism being used in this work. To make any performance comparisons meaningful, the different types of parallelism in operation must be considered when making any comparisons with other implementations of the same algorithm. For this reason the performance comparison is made up of two parts. Firstly, a comparison of the design with a standard microprocessor is made, but without the geometric parallelism. That is, only a single fitness evaluation is made at any one time. Secondly, a comparison is made for different degrees of geometric parallelism.

Comparing the performance of the FPGA system without geometric parallelism to a modern RISC processor is considered reasonable on the grounds that this comparison has been used previously in some of the work reviewed in Chapter 3.

## 4.7 Experimental Setup

To test the feasibility of implementing a GP system in hardware using Handel-C, a number of experiments were devised. This section describes the environment used for the experiments.

There were four aims of running these experiments:

1. To determine whether the system could be implemented using Handel-C and to verify that the design would fit on an FPGA.

2. To determine if a limited GP system could solve the problems chosen.

3. To obtain some indicative performance comparisons between a traditional C implementation and a hardware implementation.

4. To find out whether the design was realisable as hardware and to implement the design in hardware.

To meet the above aims, the problems were run using five different environments. Firstly, as a standard ISO-C application running under Linux. This was to prove the initial program operation and to enable the application to be debugged using standard GNU tools. The program was compiled using gcc v2.95.2 and executed on a 200 MHz AMD K6 PC running Linux.

Secondly, the program was compiled using Handel-C and optimisations were made to the code to increase parallelism, reduce logic depth and minimise the gate count.

Thirdly, the Handel-C implementation was run using the Handel-C simulator. This gave the number of clock cycles needed to execute the program.

Fourthly, the C code was compiled using a cross compiler and executed on an instruction simulator for the Motorola PowerPC architecture. This was performed to obtain a count of instruction and memory cycles needed for a modern processor. The choice of the PowerPC was made on the basis of a readily available simulator for the PowerPC. The PowerPC simulation was performed by using gcc 2.95.2 configured as a PowerPC cross compiler. This version of the program was optimised so as to have a minimal start-up overhead and to avoid using any I/O. It is therefore as close to the FPGA program as possible, allowing a meaningful comparison of performance to be made. The simulator itself was psim [Meissner 01], which is built into the GNU debugger (gdb) from version 5.0 onwards. Psim can also be run as a stand-alone application. Appendix C gives more details of how psim was configured and run.

Fifthly, the output from Handel-C was used to generate a hardware layout for the place and route tools, which gave the maximum clock frequency the design could achieve, as well as an indication of the FPGA resources required.

The design was then transferred to hardware to verify the correct operation of the program.

For the Handel-C simulation and hardware implementation, the code was compiled using Handel-C v3.0 using maximum optimisation. The final FPGA configuration data was produced using Xilinx Design Manager version 3.3i for a Xilinx Virtex XCV2000e-6 chip hosted on a Celoxica RC1000 development board. A block diagram of this board is shown in Figure 4.4 and a photograph of the board showing the Xilinx FPGA is shown in Figure 4.5. This board contains a *Peripheral Component Interconnect* (PCI) bridge that communicates between the RC1000 board and the host computers PCI bus, four banks of *Static Random Access Memory* (SRAM) and a Xilinx FPGA. Logic circuits isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM. The SRAM can be configured as either 2 MiB by 8 bits each, or 512 KiB by 32 bits. For this work, the SRAM was configured as 32 bits wide.

The host computer is responsible for downloading the configuration data to the FPGA. The host can then communicate with the FPGA to control the operation, send data to and read data from the FPGA. A program written to run on the host performed the following operations:

```
begin program host-control-1
  reset RC1000
  configure FPGA
  seed the random number generator in SRAM
  signal FPGA to start
  wait for FPGA to complete
  read results from FPGA using status word
  search for best individual
  output best individual
end
```

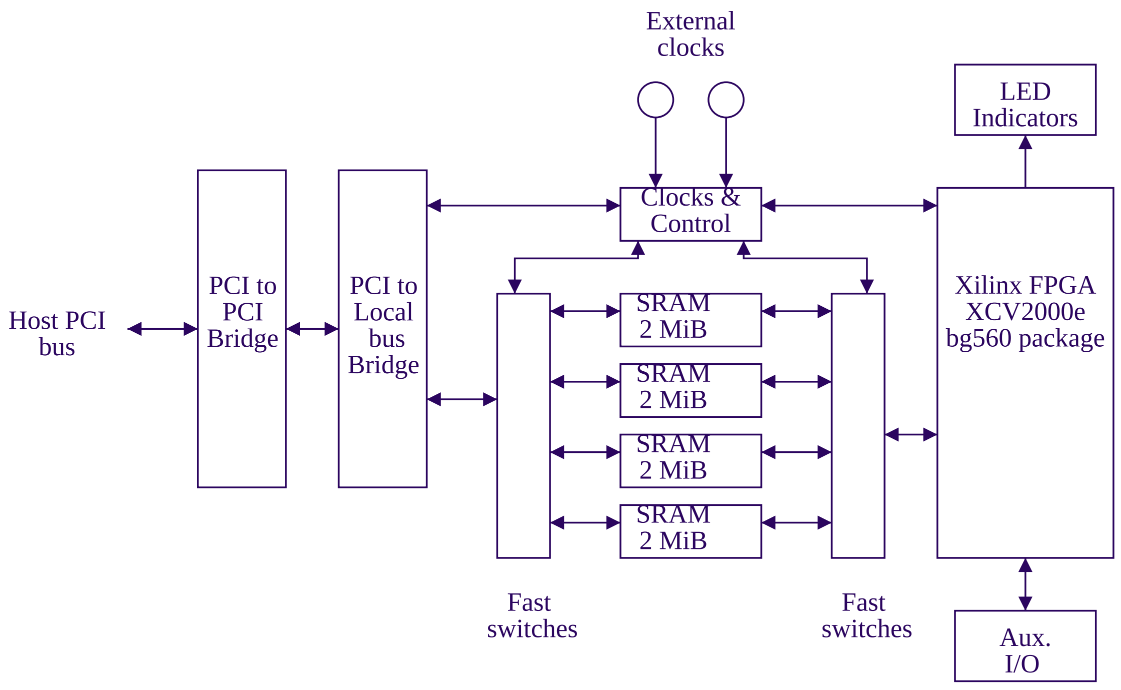


**Figure 4.4:** Block diagram of the Celoxica RC1000 FPGA board. It contains a *Peripheral Component Interconnect* (PCI) bridge that communicates between the RC1000 board and the host computers PCI bus, four banks of *Static Random Access Memory* (SRAM) and a Xilinx FPGA. Logic circuits isolate the FPGA from the SRAM, allowing both the host CPU and the FPGA to access the SRAM. The board also contains two *PCI Mezzanine Card* (PMC) connections to host optional I/O modules.

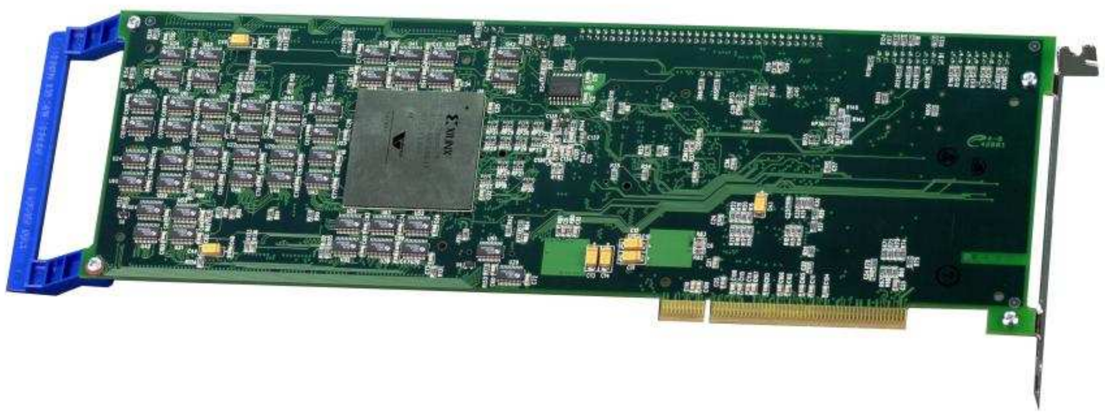


**Figure 4.5:** Photograph of the RC1000 development board. The large device near the centre of the board is the Xilinx FPGA. The SRAM and PCI bridge are located on the other side of the board. Photograph reprinted with permission of Celoxica Ltd.

The FPGA design wrote its output to an 8 bit output port as a sequence of key/data pairs. This data was read by the host PC and saved to a disk file for later analysis. A disassembler was written to decode the output data for analysis.

When measuring the clock counts of both the Handel-C simulation and the PowerPC simulation, the code was modified to run to the maximum number of generations.

### 4.7.1 Estimating PowerPC Clock Cycles

Estimating the number of clock cycles required to execute the PowerPC version of this program is a complex process. Timing is dependent on how well the compiler has arranged the instruction flow to avoid pipeline stalls, accurate branch prediction, how much of the program is in instruction cache and how many external memory reads/writes are required. It also depends how fast the hardware is, especially the memory subsystem. From the Motorola data sheet in the MPC860 Users Manual [Motorola 98], an external load (read) takes 2 cycles when the data is in cache and 3 additional clock cycles when it is not in cache. A write to memory requires 1 cycle. A best case instruction when executed from data cache requires 1 cycle for most common instructions. An estimate of the number of clock cycles required by a program is given by:

$$\text{Clock}_{\text{total}} \simeq (r \times (1 - \text{dhit}) \times 3) + i \tag{4.1}$$

where $r$ is the number of reads from memory, dhit is the percentage of reads satisfied by the data cache. This is estimated to be 60%, based on anecdotal evidence. $i$ is the number of cycles required to execute the program, including pipeline stalls and branch prediction failures. The figure for $i$ also accounts for all writes to memory and all reads from cache.

The instruction and read counts are taken from the output of the instruction simulator. Equation 4.1 assumes that there are no external memory wait states caused by slow memory or bus contention, and that instruction scheduling is optimal.

## 4.8 Experiment Descriptions and Results

Three experiments were devised to prove the general concept of GP in hardware using Handel-C and to start to investigate the behaviour of the GP system when changing the number of parallel fitness evaluations. Two problems were used for the experiments. The limited memory available, without using the *Block Random Access Memory* (BRAM) built into the Xilinx FPGA, means that the problems need to be sufficiently simple to be solved using a small program size and small population size. During program development, the population size and program size were modified until reasonable values were found that allowed the problems to be solved, and allowed the code to be compiled on the workstation using Handel-C. The last point arose because the Handel-C compiler requires substantial computational resources so that arbitrarily large designs cannot be compiled successfully. The figures arrived at were a population size of 16, together with a program size of 8 or 16 depending on the problem chosen. These figures also allowed up to 4 parallel fitness evaluations to be accommodated.

The two problems chosen were a regression problem and a Boolean logic problem. The regression problem uses integer values, since Handel-C does not support a native floating point data type. The regression problem chosen is $x = a + 2b$. The Boolean logic problem is the 2 bit XOR function $x = a \oplus b$.

The problems were realised as a single source file with preprocessor directives controlling problem specific sections.

In both problems the raw fitness was arranged to be zero for a 100% correct program, thereby reducing the amount of logic required to test for fitness.

In both problems, the run was terminated if a 100% correct program was found, or if the maximum number of generations was reached.

### 4.8.1 Regression Problem

The objective of this problem is to find a symbolic expression that fits 4 data points.

#### Description

In common with all GP work, each problem typically requires the selection of appropriate functions. The functions are implemented as opcodes for a problem specific processor. For the regression problem using standard GP, the functions include Addition, Subtraction, Multiplication and Division. In this implementation eight instructions were chosen, requiring three bits. Each instruction can specify up to two registers, and there are four registers available, requiring 2 bits each. Therefore each instruction requires 7 bits of storage.

The instructions for this problem are:

- `add`($R_n$,$R_m$) adds the contents of $R_m$ to the contents of $R_n$ and places the result back into $R_n$.

- `sub`($R_n$,$R_m$) subtracts the value in $R_m$ from the value in $R_n$ and places the result back into $R_n$.
- `shl`($R_n$) shifts the contents of $R_n$ left by one bit, leaving the result in $R_n$.
- `shr`($R_n$) shifts the contents of $R_n$ right by one bit, leaving the result in $R_n$.
- `nop` is a no-operation function. This was included to make the number of instructions a power of 2.
- `halt`($R_n$) causes the evaluation to finish, returning the value in $R_n$.
- `ldim`($R_n$,$K_m$) causes the constant $K_m$ to be placed into $R_n$.
- `jmpifz`($R_n$,$R_m$) tests the value in $R_n$. If the value is zero, then jumps to the location in $R_m$ modulo program size.

Program termination occurs on the following conditions:

1. a halt instruction is encountered;
2. the last instruction in the program is executed;
3. a jmpifz instruction has caused a loop to be created, and a predetermined number of loops have been executed. The counter is set to zero at the start of the program and is incremented for every jmpifz instruction.

The machine that implements these instructions can execute one instruction every two clock cycles, including instruction fetch, decode, operand address evaluation and operand read/write. To speed this up even further it would be possible to build a pipeline, reducing the cycle count to one per instruction, if there were no jmpifz instructions.

Four random constants are made available to each individual. These are created once during the initial construction of individuals.

Most examples of regression in the literature include the multiply and divide functions. Since these two functions generate very deep logic using the default Handel-C operators, these were replaced with single bit shift left and shift right operators, which generate much shallower and therefore faster logic, and have the effect of multiply by two and divide by two instructions respectively. The jump-if-zero opcode was included to allow loops or conditional expressions to appear.

The full set of parameters for the regression problem are given in Table 4.1.

The input values *a* and *b* were placed in registers $R_0$ and $R_1$ before the fitness evaluation. The result *x* was read from register $R_0$ if the program was terminated at the end, or the value in $R_n$ if terminated by a `halt` instruction.

The fitness data was pre-computed once at the start of the program and made available to all copies of the fitness evaluation.

**Table 4.1:** Parameters for the regression problem

| Parameter | Value |
|---|---|
| Population size | 16 |
| Functions | `add`($R_n$, $R_m$), `sub`($R_n$, $R_m$), `shl`($R_n$),`shr`($R_n$), `nop`, `halt`($R_n$), `ldim`($R_n$, $K_m$), `jmpifz`($R_n$, $R_m$) |
| Terminals | 4 registers |
| Word size | 8 bits |
| Max. program size | 8 |
| Generations | 511 |
| Fitness cases | 4 pairs of values of *a* and *b* |
| Raw fitness | The absolute value of the difference between the returned value and the expected value |

**Regression problem results**

The results from the simulator for this problem are given in Table 4.2. The figures for the PowerPC were calculated using the method described in Equation 4.1.

**Table 4.2:** Results of running the regression problem

| Measurement | PowerPC Simulation | Handel-C (Single fitness evaluation) | Handel-C (4 parallel fitness evaluations) |
|---|---|---|---|
| Cycles | 16 612 624 | 351 178 | 188 857 |
| Clock frequency | 200 MHz | 25 MHz | 19 MHz |
| Estimated gates | n/a | 142 443 | 228 624 |
| Number of slices | n/a | 4 250 | 6 800 |
| Percentage of slices used | n/a | 22% | 35% |
| $\text{Speedup}_{\text{cycles}}$ | 1 | 47 | 88 |
| $\text{Speedup}_{\text{time}}$ | 1 | 6 | 8 |
| Elapsed time | 0.083 s | 0.014 s | 0.0099 s |

The estimate of NAND gates is generated by Handel-C as an indication in a vendor independent fashion of the size of the required FPGA. The number of slices used is generated by the place and route tools. The percentage of Slices used is based on the Xilinx XCV2000-BG560-6 chip, which has a total of 9 600 CLBs, arranged as an 80x120 grid. Each CLB contains two slices, giving a total of 19 200 Slices.

The speed-up factors are given for two conditions, the raw cycle counts and the actual time taken to execute the programs. The first is a comparison made in terms of raw clock cycles. This treats the two implementations as though they were operating at the same clock frequency. The second is a comparison between the typical clock rate for the PowerPC and the fastest frequency the FPGA could be clocked as reported by the place and route tools.

The speed-up factor is given by:

$$\mathrm{Speedup}_{\mathrm{cycles}} = \frac{\mathrm{Cycles}_{\mathrm{ppc}}}{\mathrm{Cycles}_{\mathrm{fpga}}} \tag{4.2}$$

and the speed-up factor for time is given by:

$$\mathrm{Speedup}_{\mathrm{time}} = \mathrm{Speedup}_{\mathrm{cycles}} \times \frac{\mathrm{Freq}_{\mathrm{fpga}}}{\mathrm{Freq}_{\mathrm{ppc}}} \tag{4.3}$$

An (annotated) example program from this problem found in generation 16 of one run is:

```
shl(r1)          // r1 = b*2
add(r1,r2)       // nop (all registers = 0 at the start)
add(r0,r1)       // r0 = a + (b*2)
halt(r0)         // Return the result in r0
```

It was found that none of the solutions used the `ldim` instruction and therefore none of the random variables.

The difference in the maximum attainable clock frequency between the single fitness evaluation case and the 4 parallel fitness evaluation case can be explained by the increased number of logic elements required. This in turn requires more routing resources and more delays.

### 4.8.2 XOR Problem

The objective of the XOR problem is to find a program that solves the 2-input Boolean exclusive OR function, using the four 2-input Boolean functions `AND`, `OR`, `NOR` and `NAND`.

**Description**

The XOR function uses the four basic two input logic primitives `AND`, `OR`, `NOR`, `NAND`. Each of these functions takes two registers, $R_n$ and $R_m$. The result is placed into $R_n$. These have been shown to be sufficient to solve the Boolean XOR problem [Koza 92]. Execution is terminated when the last instruction in the program has been executed.

The two inputs $a$ and $b$ were written to registers $R_0$ and $R_1$ before the fitness evaluation, and the result $x$ read from register $R_0$ after the fitness evaluation.

**Table 4.3:** Parameters for the XOR problem

| Parameter | Value |
|---|---|
| Population size | 16 |
| Functions | $\texttt{AND}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{OR}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{NOR}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{NAND}(\texttt{R}_n, \texttt{R}_m)$ |
| Terminals | 4 registers |
| Word size | 1 bit |
| Max program size | 16 |
| Generations | 511 |
| Fitness cases | 4 pairs of values of $a$ and $b$ |
| Raw fitness | The number of fitness cases that failed to yield the expected result. |

The full set of parameters is given in Table 4.3. With only four functions for this problem, each instruction requires six bits.

**XOR problem results**

The XOR problem was executed using the same environments as the regression problem. The results are presented in Table 4.4.

An (annotated) example program from this problem found in generation 86 of one run is:

```
or(r3,r1)        // r3 = b
or(r3,r0)        // r3 = a+b
or(r2,r1)        // nop (since r2 is never used)
nand(r0,r1)      // r0 = $\overline{ab}$
and(r0,r3)       // r0 = $(a+b)\overline{ab}$
```

The final result $(a+b)\overline{ab}$ is equivalent to $(a\overline{b})+(\overline{a}b)$ which is the more familiar logic equation for the exclusive OR function.

**Table 4.4:** Results of running the XOR problem

| Measurement | PowerPC Simulation | Handel-C (Single fitness evaluation) | Handel-C (4 parallel fitness evaluations) |
|---|---|---|---|
| Cycles | 27 785 750 | 715 506 | 384 862 |
| Clock Frequency | 200 MHz | 22 MHz | 18 MHz |
| Estimated Gates | n/a | 89 205 | 142 550 |
| Number of Slices | n/a | 4 630 | 7 434 |
| Percentage of Slices Used | n/a | 24% | 38% |
| $\text{Speedup}_{\text{cycles}}$ | 1 | 38 | 72 |
| $\text{Speedup}_{\text{time}}$ | 1 | 4 | 6 |
| Elapsed time | 0.14 s | 0.032 s | 0.021 s |

### 4.8.3 The Effect of Parallelising the Fitness Evaluation.

To quantify the benefits of using geometric parallelism, the XOR problem was re-implemented, using four different values for the number of parallel fitness evaluations, and run using the Handel-C simulator. The purpose of this experiment was not to successfully evolve programs but rather to explore how much the parallelism affected the performance.

A total population size of 8 was chosen, together with a maximum of 4 nodes per individual. These values appear to be very low but they were chosen to allow the programs to be compiled by Handel-C, since it was found that larger values caused the compilation of the simulation to fail, due to memory exhaustion on the workstation. The number of individuals processed in parallel was modified each time, using the values 1, 2, 4 and 8. Data was collected for the number of cycles to perform the initial population creation, the number of cycles to evaluate the first generation and the number of cycles to perform the breeding operators on the first generation. These are shown in tabular form in Table 4.5.

Figure 4.6 shows the effect on the number of cycles for one fitness evaluation, with different numbers of parallel fitness evaluations. It can be seen from this graph that as the number of parallel fitness evaluations increases, so the benefit tails off. This is due to the constant overhead associated with setting up the fitness evaluations.

The total number of cycles for the problem is shown in Figure 4.7. The program was run for 16 generations. Here the effect of the breeding phase can be seen. The benefit gained from doubling the number of parallel fitness evaluations from four to eight only reduces the cycles required by

**Table 4.5:** Cycle counts and gate estimates for various stages of the GP and different numbers of parallel fitness evaluations. Where N = Number of parallel fitness evaluations. I = Initial population creation (cycles). E = Evaluation of the first generation (cycles). B = Breeding of first generation (cycles). T=Total cycles. G=Gate estimate (NAND gates).

| N | I | E | B | T | G |
|---|---|---|---|---|---|
| 1 | 214 | 324 | 123 | 6517 | 35 666 |
| 2 | 214 | 180 | 123 | 4669 | 43 314 |
| 4 | 214 | 108 | 123 | 3549 | 58 588 |
| 8 | 214 | 60 | 123 | 2877 | 89 136 |

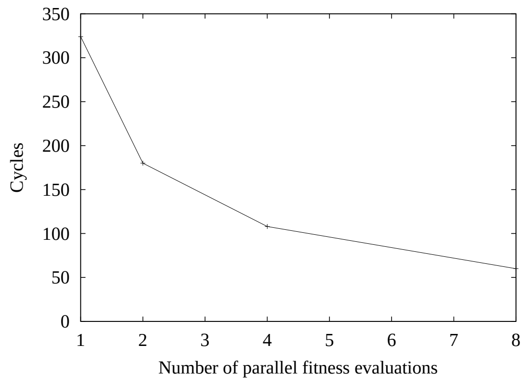


**Figure 4.6:** Number of cycles to evaluate one fitness function evaluation for the population with different numbers of parallel fitness evaluations.

18%. The contribution of the initial population is about 7.5% of the total cycles, when 8 parallel evaluations were performed. This shows clearly that performing fitness and breeding serially does not allow this implementation to exploit parallelism to its best advantage.

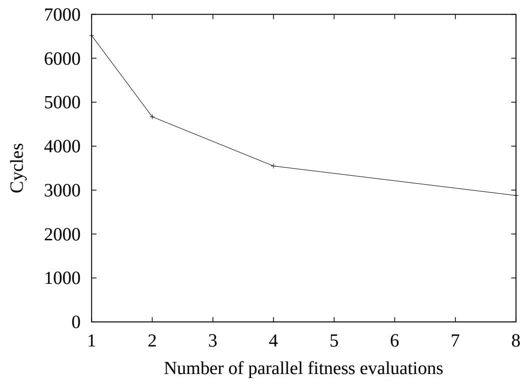


**Figure 4.7:** Total number of cycles for the problem with different numbers of parallel fitness evaluations.

## 4.9 Discussion

### 4.9.1 Consequences of Using a High Level Language

The two problems presented here, though trivial when compared to many problems that have been solved using GP, have proved the general concept of using Handel-C to produce GP systems that can be run on FPGAs. The use of a C like language has some valuable properties. Probably the most significant is that the algorithm can be developed and tested using traditional software tools. This is an important consideration for software engineering, in that there is no need for a software engineer to become proficient in hardware design. This opens up a whole set of possibilities for implementing critical functions in hardware.

However, the issue of productivity needs to be considered here. Compiling the ISO-C implementation using gcc took around 3 seconds to complete, at which time testing could commence. When using Handel-C to compile a simulation, the initial compilation phase took several minutes, and compilation for a host simulation run using Microsoft Visual C++ V6.0 took around 10 minutes. Finally, targeting the FPGA required about 30 minutes for Handel-C to produce the netlist, and several hours for the place and route tools to create the FPGA configuration data. Clearly, using Handel-C for this particular problem needs careful preparation and the judicious use of traditional software tools during the early development phase. It must be stressed that the largest bottleneck is the place and route tools, a problem that any user of FPGA techniques will be familiar with. For reference, all Handel-C and place and route work was performed on a 500 MHz Pentium-II workstation with 384 MiB of RAM running Windows NT4.0. The full capabilities of the FPGA cannot be exploited using such a workstation since the demands on memory are large, evidence suggests that at at least 1 GiB of memory would be required to compile and place/route a design that would fill a Xilinx XCV2000e part (Virtex-E) (see Appendix C).

When using VHDL to design the hardware it is possible to use the FPGA vendor's analysis and simulation tools to get an accurate picture of how the hardware will behave and get an indication of the expected performance. For example, Scott [Scott 94] shows how by using the simulation tools, the design could improved. However, when using Handel-C it is slightly more difficult to get detailed information on how the design will operate without first going through a place and route cycle. Because of the lack of detailed timing data it is harder to tune a design in Handel-C because of the abstraction introduced by Handel-C.

### 4.9.2 The Effect of Increasing Parallelisation of the Fitness Function

The results in Section 4.8.3 show that, using the current implementation and parameter values, the benefits of increasing the number of parallel fitness evaluations falls off above 4. This is due to the breeding phase taking a significant portion of the cycles when compared to the fitness evaluation. This is a direct consequence of the linear representation of the individuals and the unsophisticated crossover operator. It is also a consequence of performing the breed and evaluations in series rather than in parallel. The effect is also amplified because of the simplicity of the problems chosen. Clearly, more work needs to be done in the area of representation and crossover if the benefits of parallelisation are to be fully realised using this implementation.

### 4.9.3 Performance Considerations and Potential Improvements

The work reviewed in Chapter 3 indicated that a performance improvement over a software implementation of two or three orders of magnitude has been achieved by implementing part or all of a GA in hardware. The work described in this chapter has not achieved that level of improvement. This is probably due to the straightforward translation of a serial algorithm into hardware, without considering algorithmic parallelism from the outset, and the limited number of parallel fitness evaluations that could be accommodated.

To achieve maximum performance, algorithmic parallelism or pipelining should be used to perform the selection, breeding and fitness evaluation phases in parallel. In the steady state model of GP with a large population, the system could evaluate a number of individuals, while the breeding of previously evaluated individuals could be carried out in parallel, effectively forming a pipeline.

An estimate of the worst case performance, if the system were implemented using a pipeline, can be made by assuming that:

(a) The fitness evaluation will require the most cycles and will therefore be the slowest stage in the pipeline. This means that we only need to consider how many cycles will be needed for the fitness evaluation. This is shown in Table 4.5 as being reasonable.

(b) Each program is run exactly once per fitness case and that there is no looping or recursion.

(c) All stages are fully pipelined, that is to say that creation, selection, fitness evaluation, random number generation and breeding are all performed in parallel so that a result is produced every $c$ cycles.

(d) All programs are of maximum length.

The number of cycles $C$ required is given by:

$$C = k + \left( \frac{G \times l \times M \times n \times c}{p} \right) \tag{4.4}$$

where $G$ is the number of generations, $l$ is the maximum program length, $M$ is the population size, $n$ is the number of fitness cases, $c$ is the number of clock cycles per instruction or function, $p$ is the number of fitness evaluations performed in parallel, and $k$ is the fixed overhead for startup, general control and generating the final result.

For the XOR problem described in Section 4.8.2, assuming $k$ =500, $G$ =511, $l$ =16, $M$ =16, $n = 4$, $c = 2$, and $p$ =4, this gives a total cycle count of 262 132, a potential improvement of 1.4 times over the results shown in Table 4.4.

Clearly, the implementation of a fully pipelined GP system must be considered for future work.

A further performance boost is possible by increasing the value of $p$. When the population is moved from memory constructed from *Look Up Tables* (LUTs) and Flip Flops to on-chip block select RAM and/or external RAM, it should be possible to accommodate more logic to perform the fitness evaluations and therefore increase $p$ from 4 to a significantly larger value. A value of 32 for $p$ would yield a cycle count of 33 204, which would mean that Speedup$_{\text{time}}$ for the XOR problem would be 75.

### 4.9.4 The Potential of Problem Specifi c Op–Codes

A key difference between this work and that of Nordin and Banzhaf [Nordin 95b], which used a standard microprocessor, is that we are not constrained to a function set that a microprocessor designer sees fit to implement. That is to say, the functions can be designed to have a higher level of abstraction than machine instructions. While the experiments presented in this chapter were restricted to fairly standard microprocessor like opcodes, other problems need not be so restricted.

One example of a problem where the function set is expressed at a high level of abstraction is the Evolution of Emergent Behavior in [Koza 92] page 329. Here the function and terminal set require several steps to be performed. If implemented using a Reduced Instruction Set Computer (RISC) or Complex Instruction Set Computer (CISC) architecture, each step would require several instructions to be executed and therefore require more than one clock cycle to execute. With Handel-C the functions could be encoded efficiently and compactly. An example from the Evolution of Emergent Behaviour work is the implementation of the PICK-UP operator, which picks up food (if any) at the current position if the ant is not already carrying food. Using Handel-C the operator can be written so that it requires one clock cycle:

```
unsigned char grid[32][32];
unsigned int  x,y,carrying_food;

if(!carrying_food && grid[x][y]) {
  par {
    carrying_food = 1;
    grid[x][y] = 0;
  }
}
```

As a comparison, when compiled using gcc, this sequence of statements requires 21 RISC (PowerPC) instructions to be executed when the **`if`** expression evaluates to true.

## 4.10 Summary

This chapter has presented an initial implementation of a GP system written in Handel-C, which can then be used to configure an FPGA. The GP system includes initial population creation, fitness evaluation, selection and breeding operators, and can output the result. To demonstrate the viability of this approach, two simple problems have been solved. The performance of the FPGA implementation is better than the equivalent software implementation without using parallel fitness evaluations. When parallel fitness evaluations were used, the performance increased as well. However, simply translating a serial algorithm into hardware does not exploit the capabilities of the hardware fully. To achieve even better performance the system should make use of pipelining.

To extend the capabilities of this work further, a method of storing the population in external RAM is needed. To accommodate off-chip RAM, which can only be read or written to once per clock cycle, and which has a limited word size, development of an efficient coding scheme will need to be devised and it is likely that a pipelined design would be needed to get the most benefit from using external RAM. The potential for realising even better performance by using a fully pipelined design is clear, as is increasing the number of parallel fitness evaluations. These ideas are explored in the next chapter which uses pipelining to improve throughput and external RAM to store the population.

# Chapter 5

# An Optimised Implementation

The system described in Chapter 4 stored the population directly in on-chip *Look up Table* (LUT) memory. It also used a sequential model of operation. Consequently, it suffered from two major deficiencies. Firstly, because of the inefficiencies of using LUTs to store individuals, the size of the population that could be handled was very small – in the region of 16 individuals. Secondly, the sequential nature of the select-breed-evaluate sequence resulted in a non-linear relationship between the number of parallel evaluations and the speed-up observed. This chapter describes a number of architectural changes that address the issues of population size and throughput.

The chapter begins with a description of the architectural changes. The experimental setup is then presented together with some experimental results that illustrate the effect of the changes. The changes are then discussed and areas for further work are suggested.

## 5.1 System Architecture

### 5.1.1 Extending the Population Size

In the revised architecture, large populations are supported by storing the entire population in off-chip *Static Random Access Memory* (SRAM). The Celoxica RC1000 board has 8 MiB of SRAM arranged as 4 banks of 2 MiB that can be directly addressed by the FPGA. Each bank is configured as 512 Ki 32bit words. One bank is used for storing the results of the run (fitness and lengths of each individual), leaving three banks available for the population. The total population size is determined by the program size chosen and the size of the program nodes. Table 5.1 illustrates the potential range that can be accommodated for a node size of 32 bits.

**Table 5.1:** Possible population sizes when using three 2 MiB memory banks and a word size of 32 bits for different program sizes.

| Max program Length (words) | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|
| Max population size | 98 304 | 49 152 | 24 576 | 12 288 | 6 144 | 3 072 | 1 536 |

No attempt has been made to compress more than one node into a 32 bit word. If even larger populations are needed then this could yield a possible 4 fold increase assuming an 8 bit node. Because access to external memory is a bottleneck, a small working subset of the population as well as the total population fitness values and some control information is kept in on-chip *Block Random Access Memory* (BRAM) on the FPGA where they can be accessed more efficiently by the logic. From the survey in Chapter 2, Figure 2.10 on page 36, it can be seen that these population sizes would allow many GP problems to be implemented using the RC1000 hardware.

External SRAM can only be written to or read from once per clock cycle, so care was taken in the design to ensure that parallel access to memory would not occur. Similarly, the on-chip BRAM must not be accessed more than once per clock cycle so concurrent access to the BRAMs is achieved by partitioning the BRAMs into smaller blocks that can be accessed in parallel, and by using the dual port features of the BRAMs.

The internal program representation using a linear sequence of instructions and a register machine remains unchanged.

### 5.1.2 Replacement Strategy

In conventional steady-state GP, once an individual has been evaluated it replaces the worst individual in the population. In a hardware implementation with parallel fitness evaluations this is expensive to implement since a global search is required. An alternative to this, called survival-driven evolution, has been successfully used by Shackleford *et al.* [Shackleford 01]. In Shackleford's scheme, only offspring that are fitter than the worst of their two parents will survive into the next generation by replacing one of the parents. This removes the need for any global search. A similar approach is used by Yoshida *et al.* [Yoshida 01].

This scheme was adapted to the current work by maintaining a record of the fitness of one of the parents of each individual. After crossover, each new individual is composed of two parts, one from each parent. One parent is designated the *root* parent, and the other the *tail* parent. The fitness of the root parent is used to decide whether a child should replace its parent in the main population after its fitness has been evaluated. If a child has the same fitness or better fitness than its root parent, it will replace its root parent in the main population. In the following example, $x(n)$ is used to indicate an individual $x$ with fitness $n$ and $x*(n)$ indicates a child individual whose root parent is $x$. The lower the value of $n$ the fitter the individual. Consider two individuals $a(10)$ and $b(8)$ which have been selected for breeding. After crossover, two new individuals, $a*$ and $b*$, are created. After fitness evaluation, these have fitness values of $a*(9)$ and $b*(9)$. Because $a*(9)$ is better than its root parent, $a(10)$, it will replace $a(10)$ in the main population. However, $b*(9)$ is not as fit as its root parent, $b(8)$, so it will not replace its root parent in the main population.

### 5.1.3 Using Pipelines to Improve Performance

Implementing algorithmic parallelism, or pipelining, is a frequently used technique in hardware design that reduces the number of clock cycles needed to perform complex operations and has been used in previous hardware implementations of GAs and GP 3.3.2. In particular, Scott [Scott 94] and Shackleford *et al.* [Shackleford 01] used coarse grained pipelines so that the evolutionary steps and the fitness steps were carried out in parallel. The revised design uses two types of pipelines: a coarse grained control pipeline, and fine grained execution pipelines. The coarse grained control pipeline is responsible for executing the fitness evaluation in parallel with the other stages of GP. The four major GP operations are divided among the stages of the pipeline: selection of individuals from the population for breeding, breeding new individuals, fitness evaluation of the individuals and replacement of the new individuals in the population. To control access to the main population in SRAM during the selection phase, which reads individuals from SRAM into BRAM, and writing modified programs back to SRAM, these two operations are combined into one stage so that they execute in sequence. Breeding is closely tied to selection and needs to occur before evaluation can take place so this is combined into the selection phase. Figure 5.1 illustrates the flow of control and the coarse-grained control pipeline.

Stage 1 runs continuously, with the random number generator creating a new random number every clock cycle. The random numbers are available to the rest of the machine with no overhead. Stage 1 also contains the logic for counting cycles. Stage 2 is the main GP machine and consists of

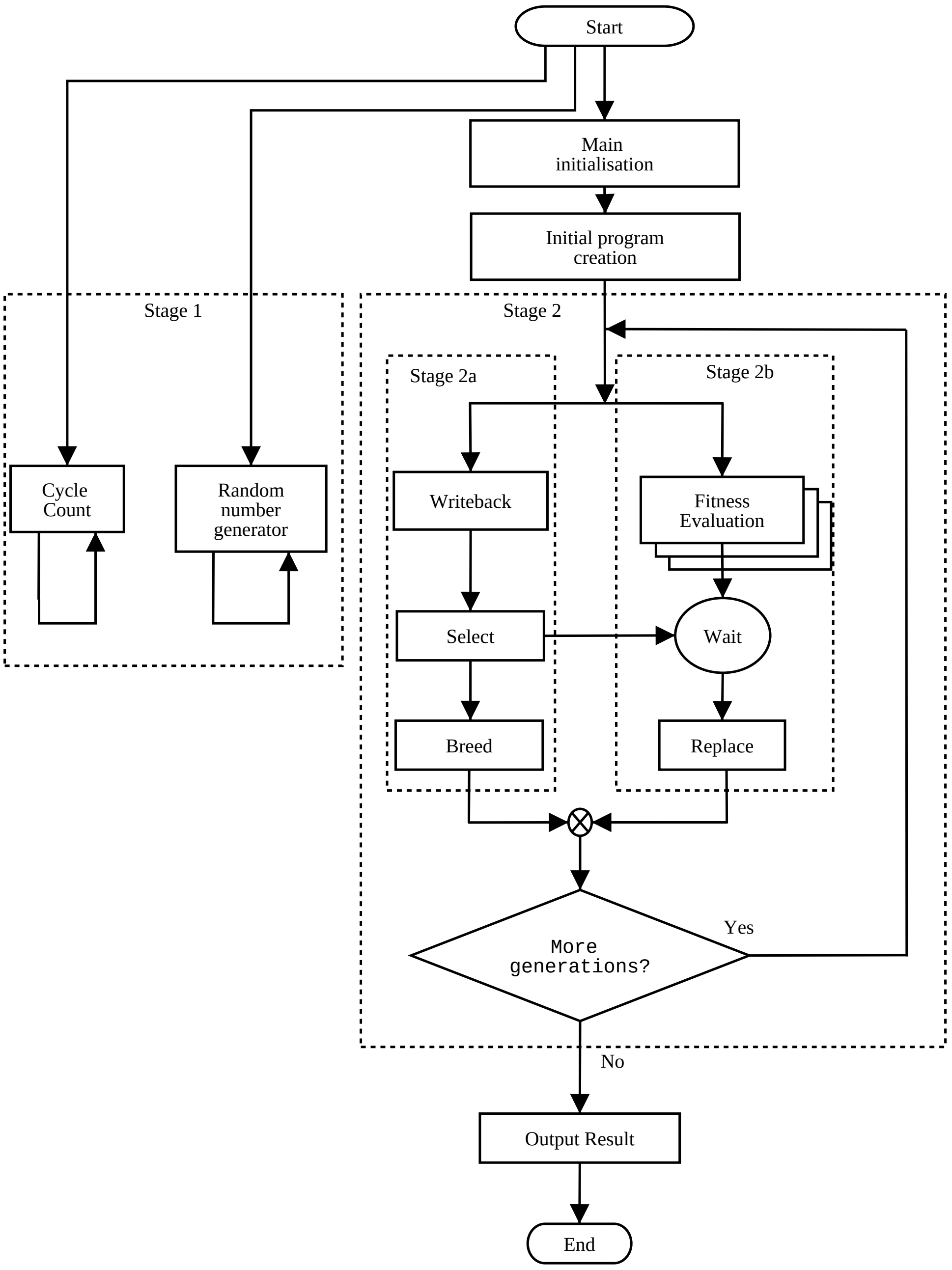


**Figure 5.1:** Overall control flow of the pipelined GP system. Stage 1 runs continuously generating random numbers and counting cycles. Stage 2 is the main GP machine which is itself a pipeline consisting of two stages. The multiple instances of Fitness Evaluation indicates that there may be many copies of this function, all operating in parallel. The pipeline in Stage 2 is formed by feeding the output of Stage 2a into Stage2b.

two sub-stages. Communication between the WriteBack/Select/Breed sub-stage (stage 2a) and the Evaluate/Replace sub-stage (stage 2b) is via a two dimensional array of individuals in block RAM, indexed by a global phase index which is toggled each time stages 2a and 2b complete. The Write-

Back phase updates the main population in SRAM with the result of the preceding Evaluate/Replace phase. The Select phase selects a series of two parents, using tournament selection, and copies the selected individuals from SRAM to the on-chip BRAM. The Breed phase then creates a series of new individuals ready for stage 2b to use. Stage 2b performs parallel evaluations of the individuals and then determines which individuals should be replaced. The individuals identified for replacement are in turn written back to the main population at the start of the next WriteBack/Select/Breed sub-stage. The wait between evaluation and replacement is needed because both selection and replacement require access to the global fitness vector. In practice, the wait only comes into play when the evaluation phase takes less time than WriteBack and Selection which only happens for very simple fitness functions.

The operation of the design is shown in Figure 5.2 where the first six iterations of stages 2a and 2b are shown. The diagram shows the main population, held in SRAM, and its associated control data. It also shows the state of the two sub-sets of the main population held in BRAM. These are labelled BRAM *X* and BRAM *Y*. The illustration shows the case where the design has been configured for two parallel fitness evaluations. If more parallel fitness evaluations are configured, then BRAM *X* and BRAM *Y* will contain more individuals. The population size in this example is 8. The following describes the operation of the design in detail for the first six iterations of the main loop of stage 2 in Figure 5.1, starting after the initial program creation stage. The worst fitness value is 15.

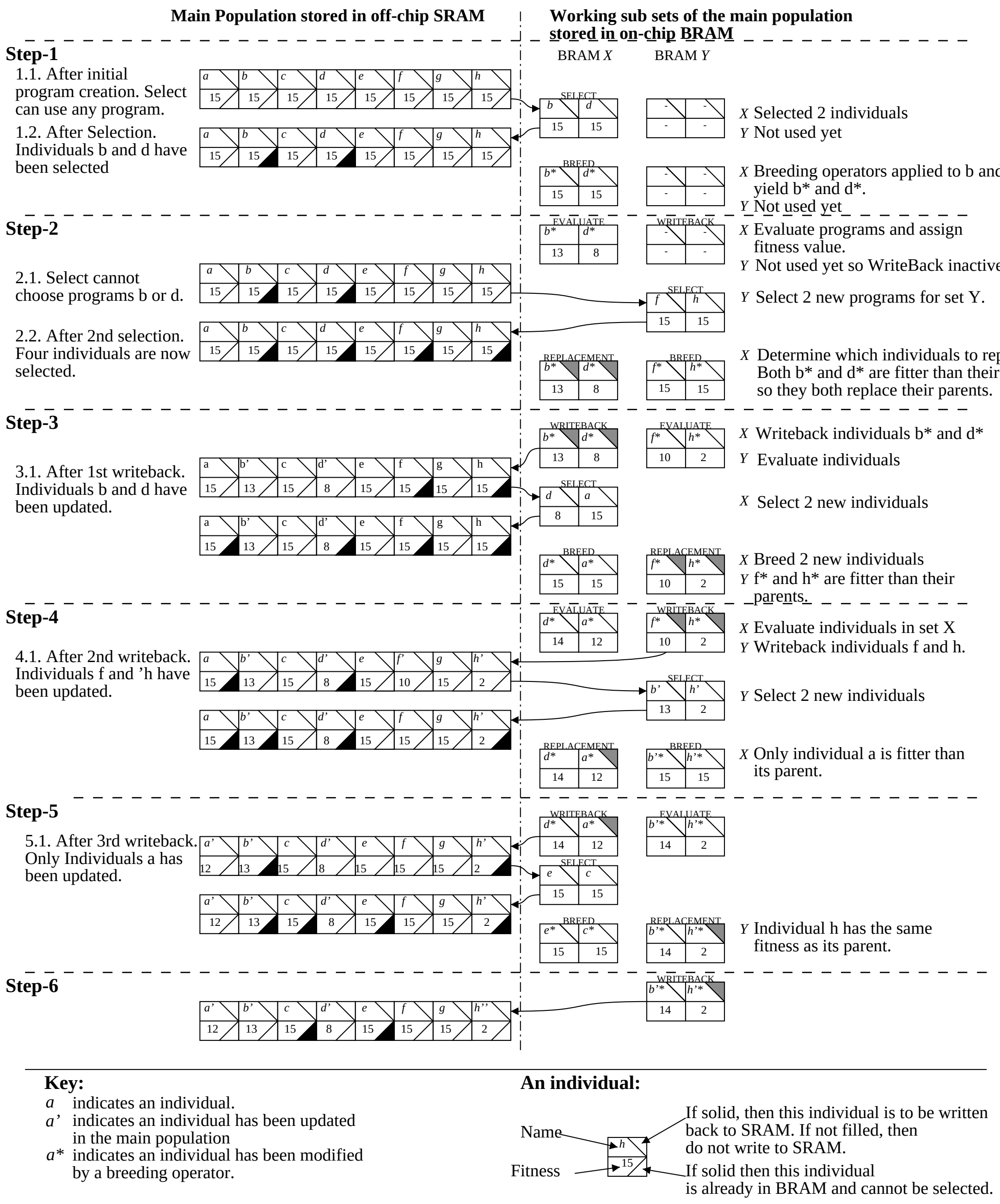

Main Population stored in off-chip SRAM
Working sub sets of the main population stored in on-chip BRAM
BRAM X
BRAM Y
Step-1
1.1. After initial program creation. Select can use any program.
1.2. After Selection. Individuals b and d have been selected
SELECT
BREED
X Selected 2 individuals
Y Not used yet
X Breeding operators applied to b and d to yield b* and d*.
Y Not used yet
Step-2
EVALUATE
WRITEBACK
X Evaluate programs and assign fitness value.
Y Not used yet so WriteBack inactive
2.1. Select cannot choose programs b or d.
Y Select 2 new programs for set Y.
2.2. After 2nd selection. Four individuals are now selected.
REPLACEMENT
X Determine which individuals to replace. Both b* and d* are fitter than their parents so they both replace their parents.
Step-3
3.1. After 1st writeback. Individuals b and d have been updated.
X Writeback individuals b* and d*
Y Evaluate individuals
X Select 2 new individuals
X Breed 2 new individuals
Y f* and h* are fitter than their parents.
Step-4
4.1. After 2nd writeback. Individuals f and ’h have been updated.
X Evaluate individuals in set X
Y Writeback individuals f and h.
Y Select 2 new individuals
X Only individual a is fitter than its parent.
Step-5
5.1. After 3rd writeback. Only Individuals a has been updated.
Y Individual h has the same fitness as its parent.
Step-6
Key:
a indicates an individual.
a’ indicates an individual has been updated in the main population
a* indicates an individual has been modified by a breeding operator.
An individual:
Name
Fitness
If solid, then this individual is to be written back to SRAM. If not filled, then do not write to SRAM.
If solid then this individual is already in BRAM and cannot be selected.

**Figure 5.2:** Illustration of the operation of the pipelined implementation.

**Step1.**

***Stage 2a.*** Two individuals, *b* and *d*, are selected, using tournament selection, and copied from SRAM into BRAM *X*. The control flags are updated to indicate that *b* and *d* are no longer available for selection. This is indicated by the solid block at the bottom right of the individual. After selection, two new individuals are created using the breeding operators. These are denoted *b** and *d**.

**Stage 2b.** At this point, there are no individuals in BRAM *Y*, so nothing will be done in this step.

**Step 2.**

***Stage 2a.*** Two new individuals, *f* and *h*, are chosen for BRAM *Y*. Note that individuals *b* and *d* were not valid candidates for selection. Again, the control flags are updated to indicate that individuals *f* and *h* cannot be selected.

***Stage 2b.*** Individuals *b** and *d** in BRAM *X* are evaluated. When the evaluations are complete, the individuals have new fitness values. These fitness values are used to determine which, if any, of the individuals are better than their parents, and so can be written back to the main population. The replacement strategy is described in Section 5.1.2. If an individual has the same fitness or is fitter than its parent, then it is marked as being ready to be written back to SRAM. This is shown by the filled block at the top right of the individual.

**Step 3.**

***Stage 2a.*** Individuals marked for being written to SRAM in BRAM *X* ( *b** and *d**) are copied back to SRAM. This updates the individual programs in SRAM and updates their fitness values. It also clears their control flags so that they are now available for selection. When the WriteBack has completed, two new individuals (*d* and *a*) are selected and copied to BRAM *X*. Note that both of the individuals just written back ( *b* and *d*) were valid candidates for this selection.

***Stage 2b.*** The individuals in BRAM *Y* are being evaluated, and their replacement status is also updated. Both individuals *f** and *h** are better then their parents, so they will be written back to SRAM.

**Step 4.**

***Stage 2a.*** Individuals *b'* and *h'* are selected and copied to BRAM *Y*. These are used to create two new individuals *b'** and *h'**. The notation *b'* and *h'* indicates that these individuals have already been modified once in a previous step.

***Stage 2b.*** Individuals *d** and *a** in BRAM *X* are evaluated and their replacement status is marked. Note, in this step, individual *d** has a worse fitness value than its parent, *d*, so will not be written back to the main population in SRAM. However, individual *a** is better than its parent, *a*, so is scheduled to be written back.

**Step 5.**

***Stage 2a.*** Individual *a** is written back to the main population and the control flags for individuals *a'* and *d'* are cleared. Two more individuals, *e* and *c*, are chosen and copied to BRAM *X*.

***Stage 2b.*** Individuals *b'** and *h'** in BRAM *Y* are evaluated. Individual *h'** has the same fitness as its parent and is therefore scheduled to be written to SRAM in the next step.

**Step 6 to *n*.**

The above sequence continues until the required number of generations have been completed. Execution finishes by writing any valid individuals from BRAM to SRAM before outputting the results of the GP run.

A finer grained level of pipelining is implemented in the fitness evaluation function. To reduce logic depth, and hence increase the clock frequency, it is often advantageous to split a complex expression into more but simpler expressions. This usually requires more clock cycles, but by pipelining the operations an effective single cycle throughput can be achieved. In this design, the function read, and decode is pipelined with the function evaluation, though the effectiveness of this is problem specific.

### 5.1.4 Measuring Performance

To compare the performance of different implementations, a way of measuring the number of cycles used by the FPGA is needed. One possibility is to use the DK1 simulator, but in large designs with long running times this can take many hours of running which is often impractical. An alternative is to include a cycle counter in the design which can be read by external programs. The internal cycle counter runs in parallel with the rest of the hardware, incrementing a counter once per clock cycle. This approach could be extended to providing fine-grained measurement of the cycles required by the individual phases, which would be valuable for evaluating the detailed performance of the design.

### 5.1.5 Obtaining the Results From a Run

Once the GP machine has finished a run, the best program needs to be communicated to the outside world. The individual programs are already in SRAM, so they can be read directly by the host. The program fitness and lengths are copied from BRAM to SRAM when the GP machine has finished so they can also be read by the host. In addition, the cycle count(s), and other parameters are made available to the host via SRAM.

## 5.2 Experimental Setup

The method of communication between the RC1000 board and the host control program was modified from that described in Chapter 4. The programs were held in the on-board SRAM so they could be accessed directly by the host control program. Once the run had completed, the FPGA wrote the run details to one of the SRAM banks so that the host control program could also gain access to these. Access to the on-board SRAM was achieved using DMA via the PCI bus. Consequently the host control program needs to be more sophisticated. The following is the pseudo-code for the revised host control program.

```
begin program host-control-2
    reset RC1000
    configure FPGA
    seed the random number generator
    signal FPGA to start
    wait for FPGA to complete
    DMA GP control data to host
    DMA GP population to host
    search for best individual
    output best individual
end
```

The control data contains sufficient information for the host control program to identify the problem and the run-time parameters, so that it can present the results of the run without requiring *a priori* knowledge of the problem being run. The control data contains the problem type as one of an enumerated set of values, the population size, the number of generations run for, the number of parallel fitness evaluations, the maximum program length, the population size, the number of cycles the main loop ran for as well as the fitness and program lengths for all programs in the final population. The control data is arranged in three pages. The page size is the based on the population size which makes the address generation within the FPGA simple and efficient . The layout of the control data area is shown in Figure 5.3.

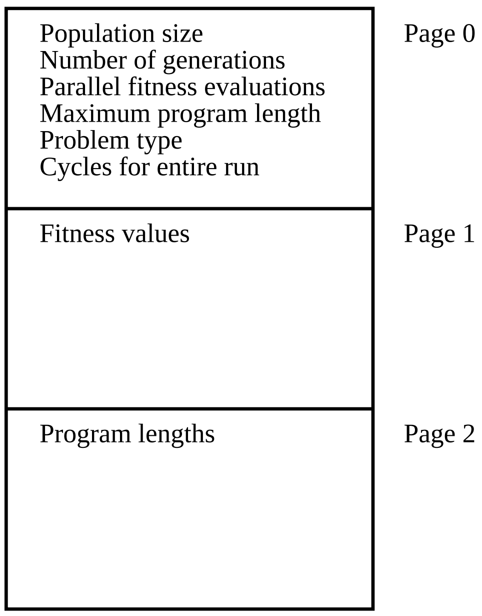


**Figure 5.3:** Layout of the control data SRAM. This is used to communicate the results of a run from the FPGA to a host based control program.

## 5.3 Experiment Descriptions and Results

### 5.3.1 Exclusive Or Problem

This is the same problem already described in Chapter 4, Section 4.8.2 on page 75

#### XOR problem results

The results from running the XOR problem are given in Table 5.2.

**Table 5.2:** Results of running the XOR problem. The results are the average of 10 runs for each configuration, each run using a different random seed.

| Measurement | PowerPC | Handel-C | | | |
|---|---|---|---|---|---|
| Parallel fitness evaluations | n/a | 1 | 2 | 4 | 8 |
| Cycles | 13 723 187 | 74 819 | 73 232 | 72 184 | 81 767 |
| Clock Frequency | 200 MHz | 52 MHz | 48 MHz | 42 MHz | 37 MHz |
| Number of Slices | n/a | 1238 | 1247 | 1725 | 2801 |
| Speedup$_{\text{cycles}}$ | 1 | 183 | 187 | 190 | 167 |
| $Spe$edup$_{\text{time}}$ | 1 | 47 | 44 | 39 | 31 |
| Elapsed time | 68 ms | 1.44 ms | 1.52 ms | 1.71 ms | 2.21 ms |

Comparing these results first with the results in Section 4.8.2, which achieved a Speedup$_{\text{time}}$ of 6 using 4 parallel evaluations, it can be seen that splitting the algorithm into two sub-stages gives a useful increase in performance. However, the surprising result is that it takes longer to run the XOR problem when more evaluations are performed in parallel, in particular with 8 parallel evaluations. Detailed investigation showed that this was a side effect of the selection method. During selection, the number of individuals selected from the main population is the number of parallel fitness evaluations wanted, and these are selected at random from the population, but only those individuals that are not currently being evaluated by the Evaluate/Replace sub-stage are valid candidates. When the number of individuals required is half the population size, many more attempts must be made by the selection phase to find valid individuals. This explains why when the number of parallel evaluations is 8, the run time is greater than when only two individuals are being selected.

The frequencies in Table 5.2 for the Handel-C implementations is as reported by the place and route tools, and takes into account the delays introduced by the combinatorial logic and the delays introduced by the routing resources used on the FPGA. A lot of effort was spent to reduce the logic and routing delays in the design, with the result that this design runs substantially faster that the previous design which could only reach 18 MHz.

Equation 4.4 on page 81 predicts the worst case number of cycles required for a problem using a pipeline. Using the figures from Table 5.2 for 4 parallel fitness evaluations and assuming the startup overhead has increased to 5000 and the number of clock cycles per instruction $c = 1$, Equation 4.4 predicts that the XOR problem would require 135 816 cycles, which is 1.8 times the number reported in the experiment. This result shows that the pipeline is still effective. The difference in predicted versus actual is explained by the fact that the prediction assumes all programs to be of maximum length.

### 5.3.2 Artifi cial Ant Problem

The objective of this problem is to evolve an ant-like program that can traverse a discontinuous trail of food laid out on a toroidal grid. The ant has a limited set of moves it can execute and a limited view of its surroundings.

#### Description

The motivation for choosing this problem for a hardware implementation is two fold: Firstly it is a hard problem for GP to solve [Langdon 98b], and secondly it demonstrates that a custom hardware design can efficiently encode the function and terminal set as native 'instructions'. That is to say one of the attractions of using an FPGA is that custom instructions not normally found in production CPUs can easily be constructed.

This popular test problem was originally described by Jefferson *et al.* [Jefferson 90] and in the context of GP was described by Koza in 1990 [Koza 90a]. It involves finding a program for an ant-like machine that enables it to navigate its way round a trail of food on a toroidal grid of cells within a fixed number of time steps. The function set $\mathcal{F} = \{\texttt{IF\_FOOD}, \texttt{PROGN2}\}$ where `IF_FOOD` is a two argument function that looks at the cell ahead and if it contains food evaluate the first terminal, otherwise evaluate the second terminal. `PROGN2` evaluates its first and second terminals in sequence. The terminal set $\mathcal{T} = \{\texttt{LEFT}, \texttt{RIGHT}, \texttt{MOVE}, \texttt{NOP}\}$, where `LEFT` and `RIGHT` change the direction the ant is facing, `MOVE` moves the ant one space forwards to a new cell, and if the new cell contains food, the food is eaten. `NOP` is a no-operation terminal and has no effect on the ant. The `NOP` terminal is included to make the number of terminals a power of 2, which simplifies the hardware logic. Each time `LEFT`, `RIGHT` or `MOVE` is executed, the ant consumes one time step. The run stops when either all the time steps have been used, or the ant has eaten all the food.

All the results in this section use the Santa Fe trail [Jefferson 90] shown in Figure 5.4, which has 89 pellets of food on a 32x32 grid. All the food pellets are present at the start of each run. The Santa Fe trail is considered to be harder than the alternative John Muir trail [Koza 90a].

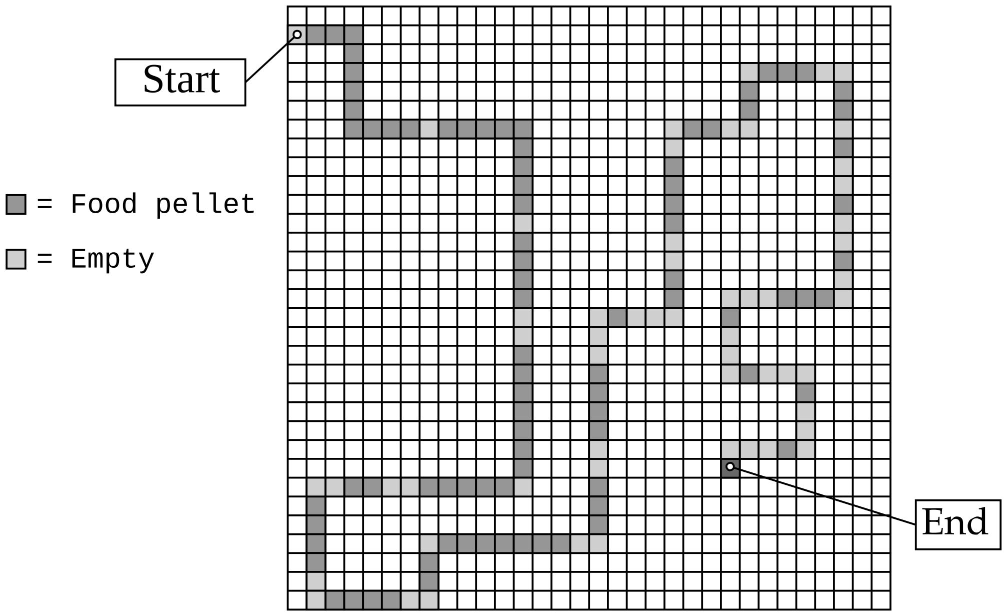


**Figure 5.4:** Santa Fe artificial ant trail. The trail is denoted by the grey squares. The darker squares contain food pellets.

The population size is 1024, the maximum program length is 31 and all experiments were run for 32 generations. The maximum number of time steps for the ant to complete its trail is 600[(1)]. The original work by Koza [Koza 92] quoted the number of timesteps as 400, but it has subsequently been shown that the number is actually 600[(2)]. The figure of 600 has been used in most of the reported experimental work using the artificial ant problem, for example, [Maxwell III 94], [Langdon 98b], [Miller 00] and others.

The full set of parameters is given in Table 5.3.

### Artificial ant problem results

The artificial ant problem was executed using the same environments as the regression problem and the results are presented in Table 5.4.

(1)This differs from the results that were published in [Martin 02b], which used 1024 timesteps.

(2)From a private email discussion with Bill Langdon.

**Table 5.3:** Parameters for the artificial ant problem

| Parameter | Value |
|---|---|
| Population Size | 1024 |
| Functions | $\texttt{IF\_FOOD}(\texttt{T}_a, \texttt{T}_b)$, $\texttt{PROGN}(\texttt{T}_a, \texttt{T}_b)$ |
| Terminals | `MOVE`, `LEFT`, `RIGHT`, `NOP` |
| Max Program Size | 32 |
| Generations | 32 |
| Fitness Cases | One fitness case. The program was run until 600 timesteps had elapsed or the ant had consumed all the food. |
| Raw Fitness | The number of pieces of food not eaten in the time available. |

**Table 5.4:** Results of running the artificial ant problem

| Measurement | PowerPC | Handel-C | | | | |
|---|---|---|---|---|---|---|
| Parallel fitness evaluations | n/a | 2 | 4 | 8 | 16 | 32 |
| Cycles | 2.26e9 | 33.21e6 | 18.08e6 | 10.25e6 | 5.87e6 | 3.28e6 |
| Clock Frequency | 200 MHz | 40 MHz | 38 MHz | 36 MHz | 33 MHz | 31 MHz |
| Number of Slices | n/a | 1 835 | 2 636 | 4 840 | 7 429 | 14 908 |
| $\text{Speedup}_{\text{cycles}}$ | 1 | 69 | 125 | 220 | 386 | 691 |
| $\text{Speedup}_{\text{time}}$ | 1 | 13 | 23 | 39 | 63 | 107 |
| Elapsed time | 11.3 s | 0.83 s | 0.47 s | 0.28 s | 0.17_s | 0.11 s |

To verify that the correct trajectory had been followed by the ant, over 150 example correct programs were examined using a graphical ant simulator. A screen-shot of the simulator is shown in Figure 5.5. It was written using the ncurses screen handling package. All the programs examined followed the trail correctly.

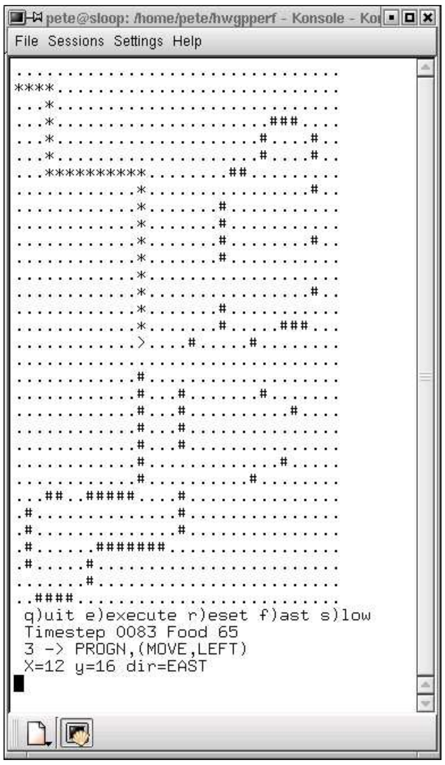
A 100% correct ant found during one run is:

```
IF_FOOD(MOVE,NOP)
IF_FOOD(LEFT,LEFT)
IF_FOOD(NOP,LEFT)
PROGN(MOVE,LEFT)
IF_FOOD(MOVE,RIGHT)
```

This program used 558 timesteps to complete the trail, and was used to display the behaviour in Figure 5.5.

Figure 5.6 shows the speedup results for the artificial ant problem, and gives both the $\text{Speedup}_{\text{cycles}}$ and $\text{Speedup}_{\text{time}}$.

These results show, that for the artificial ant problem, increasing the number of parallel fitness evaluations increases the $\text{Speedup}_{\text{cycles}}$ factor almost linearly, but because the routing delay on the


pete@sloop: /home/pete/hwgpperf - Konsole - Ko
File Sessions Settings Help

```
................................
****............................
...*............................
...*.....................###....
...*....................#....#..
...*....................#....#..
...**********........##.........
............*................#..
............*.......#...........
............*.......#...........
............*.......#........#..
............*.......#...........
............*...................
............*................#..
............*.......#...........
............*.......#.....###...
............>....#.....#........
................................
............#...................
............#...#.......#.......
............#...#..........#....
............#...#...............
............#...#...............
............#.............#.....
............#..........#........
...##..#####....#...............
.#..............#...............
.#..............#...............
.#......#######.................
.#.....#........................
.......#........................
..####..........................
 q)uit e)execute r)eset f)ast s)low
 Timestep 0083 Food 65
 3 -> PROGN,(MOVE,LEFT)
 X=12 y=16 dir=EAST
```

**Figure 5.5:** Simple graphical simulator for the artificial ant. The ant is at position x=12,y=16 and has eaten 24 of the 89 pellets of food. The ant is currently facing east and has consumed 83 of the possible 600 timesteps so far.

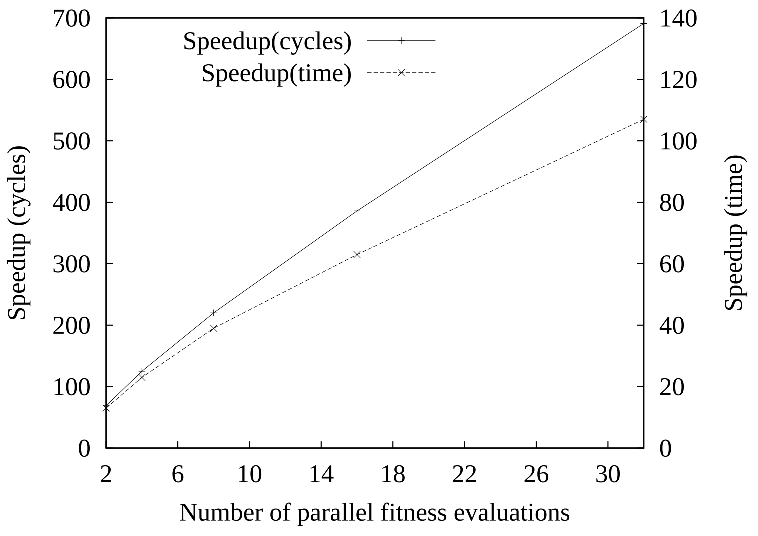


**Figure 5.6:** The speedup factors for the number of cycles (Speedup$_{\text{cycles}}$) and the time (Speedup$_{\text{time}}$) for the artificial ant problem. The Speedup$_{\text{cycles}}$ shows a nearly linear increase as the number of parallel fitness evaluations is increased, while the Speedup$_{\text{time}}$ shows a non-linear relationship due to the increased routing delays.

FPGA increases with larger designs, the maximum clock frequency decreases, offsetting some of the gains made by increasing the parallelism.

The number of slices used for 32 parallel evaluations is nearly 80% of the total available on the chip. This is effectively the limit for the XCV2000E FPGA. It is worth remarking that a PC with 750 MiB of RAM was required to run the Handel-C compiler and the place and route tools before this design could be implemented, and a PC with a 1.4 GHz Athlon CPU required nearly four hours to complete the place and route. This is in contrast to a modest 500 MHz Pentium machine capable of compiling and running lilgp and other popular GP packages.

Using Equation 4.4 for the artificial ant problem requires an estimate of the number of fitness cases. Although the ant problem specifies a single fitness case, the program is run repeatedly until either all the food is consumed, or all the timesteps have been used up. Using the shortest viable program length of 4 to consume all 600 timesteps gives an estimate of 150 for $n$. Applying Equation 4.4 for 32 parallel fitness evaluations, and given that $c = 4$, the predicted worst case number of cycles is $9.8 \times 10^6$. The measured value $4.2 \times 10^6$ is better than this worst case prediction.

#### 5.3.2.1 Comparison of hardware GP to using a standard PC

The results shown so far have compared the hardware implementation of the algorithm with a software implementation of the same algorithm. Although this shows the speedup that the hardware implementation is capable of it has only compared a relatively slow processor (200 MHz) to the hardware. Using a slow processor does not help a practitioner to decide whether a hardware or software implementation is appropriate, especially with modern processors running at over 2 GHz.

In addition, the results so far have only compared the performance of the same algorithm when run on the hardware platform and a software platform. While this comparison gives an accurate measure of the effect of using hardware, it does not give an indication of how useful the hardware GP is compared to a standard tree based GP system.

This section addresses the above limitations by comparing the performance of the hardware with the same algorithm running on a 1.4 GHz Athlon processor, and by comparing the hardware GP to a standard tree based GP system, again running on a 1.4 GHz Athlon PC. The package used for the tree based GP was lilgp. This package was chosen because of its popularity within the GP community (online survey carried out in 1998 [Martin 98, pp 4.8]). The artificial ant problem was used for these experiments. For the lilgp implementation, the standard lilgp settings were used.

Table 5.5 shows the results of running these experiments.

**Table 5.5:** Comparison of the performance of the artificial ant running in hardware to the artificial ant running on a 1.4 GHz processor, using both the hardware algorithm and lilgp. The hardware implementation used 32 parallel fitness evaluations.

| Platform | Clock Speed | Time (seconds) |
|---|---|---|
| Hardware GP | 31 MHz | 0.11 |
| Athlon | 1.4 GHz | 0.94 |
| Athlon lilgp 1.1 | 1.4 GHz | 7.94 |

It can be seen from this table that the hardware implementation is approximately 8.5 times faster than the same algorithm running on a 1.4 GHz processor. When compared with the standard lilgp program running on the same processor, the hardware implementation is over 70 times faster. This large speedup can be attributed to a number of factors. Firstly, lilgp uses a tree based GP representation and so imposes an overhead when traversing the tree structure and invoking the functions which are accessed using function pointers. Secondly, because of the large data set used in lilgp (approximately 3 MBytes), the effect of the data cache on the Athlon processor is reduced. Thirdly, in lilgp the programs lengths are not tightly constrained, so the programs that are being evaluated will tend to be longer than the hardware GP algorithm.

### 5.3.3 Boolean Even-6-Parity Problem

The objective of this problem is to evolve a function that takes *n* Boolean inputs and returns 1 if an even number of inputs evaluate to 1, 0 otherwise.

#### Description

This problem was chosen because it has been well studied and represents another popular test problem. It is a more arduous problem than the artificial ant problem since it requires a larger population, longer program lengths and is run for more generations. However, the fitness function is simpler, so more parallel fitness function evaluations can be accommodated on the FPGA than the ant problem.

For this problem the population size is 2048, the maximum program length is 256 and all experiments were run for 64 generations. The full set of parameters is given in Table 5.6. The function set for this problem included the `NOT` and `XOR` functions because it has been shown that for Boolean-*n*-parity problems the standard function set (`AND, OR, NAND, NOR` from [Koza 92]) standard GP was unable to find solutions when $n \geq 6$ [Koza 92]. However, in [Koza 92] it was found that if ADFs were used, the `XOR` function was created by the ADFs which allowed solutions to be found.

**Table 5.6:** Parameters for the Boolean even-6-parity problem

| Parameter | Value |
|---|---|
| Population Size | 2048 |
| Functions | $\texttt{AND}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{OR}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{NOT}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{XOR}(\texttt{R}_n, \texttt{R}_m)$ |
| Terminals | 16 registers |
| Max Program Size | 256 |
| Generations | 64 |
| Fitness Cases | All $2^6 = 64$ test cases |
| Raw Fitness | The number of fitness cases that were incorrect. |

### Boolean even-6-parity results

The Boolean even-6-parity problem was executed using the same environments as the regression problem and the results are presented in Table 5.7.

**Table 5.7:** Results of running the Boolean even-6-parity problem

| Measurement | PowerPC | Handel-C | | | | | |
|---|---|---|---|---|---|---|---|
| Parallel fitness evaluations | n/a | 2 | 4 | 8 | 16 | 32 | 64 |
| Cycles | 90.8e9 | 1.2e9 | 617e6 | 341e6 | 173e6 | 86e6 | 40e6 |
| Clock Frequency | 200 MHz | 48 MHz | 46 MHz | 44 MHz | 41 MHz | 38 MHz | 34 MHz |
| Number of Slices | n/a | 1 418 | 1 654 | 2 161 | 6 396 | 10 124 | 19 051 |
| Speedup$_{\text{cycles}}$ | 1 | 75 | 147 | 266 | 524 | 1055 | 2270 |
| Speedup$_{\text{time}}$ | 1 | 18 | 34 | 88 | 163 | 281 | 419 |
| Elapsed time | 454 s | 25 s | 13.4 s | 7.75 s | 4.22 s | 2.26 s | 1.18 s |

Figure 5.7 shows the speedup results for the Boolean even-6-parity problem, and gives both the Speedup$_{\text{cycles}}$ and Speedup$_{\text{time}}$. As for the artificial ant problem, these results show that for the Boolean even-6-parity problem, increasing the number of parallel fitness evaluations increases the Speedup$_{\text{cycles}}$ factor almost linearly, but because the routing delay on the FPGA increases with larger designs, the maximum clock frequency decreases, offsetting some of the gains made by increasing the parallelism. The number of slices used for 64 parallel evaluations is nearly 100% of the total available on the chip. With 64 parallel evaluations, the design took 14 hours to place and route using a 1.4 GHz Athlon workstation with 1 GiB of RAM.

Using Equation 4.4 for 64 parallel fitness evaluations and given that $c = 2$, the predicted worst case number of cycles is $67 \times 10^6$. The measured value of $40 \times 10^6$ is better than the worst case predicted value.

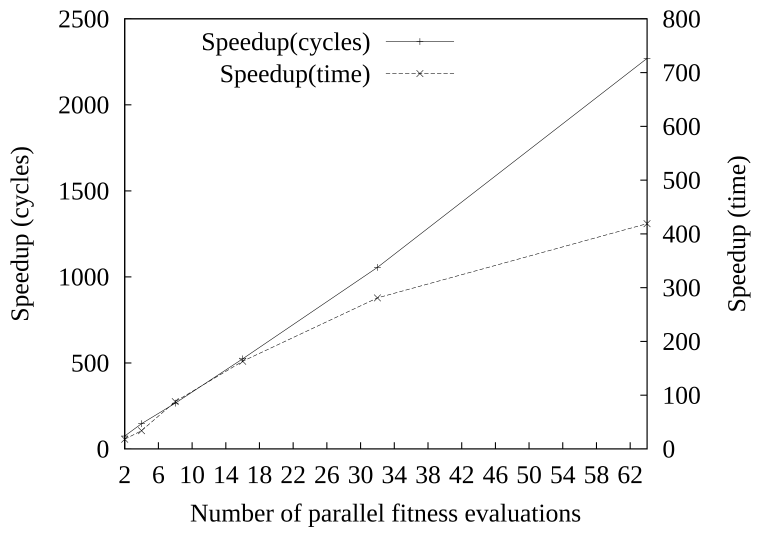


**Figure 5.7:** The speedup factors for the number of cycles (Speedup$_{\text{cycles}}$) and the time (Speedup$_{\text{time}}$) for the Boolean even-6-parity problem. The Speedup$_{\text{cycles}}$ shows a nearly linear increase as the number of parallel fitness evaluations is increased, while the Speedup$_{\text{time}}$ shows a non-linear relationship due to the increased routing delays.

## 5.4 Discussion

### 5.4.1 Effect of Implementing Pipelines and Increasing Parallelism

A direct comparison between the XOR problem in Chapter 4 and the XOR problem in this chapter (sect. 5.3.1) shows that using pipelines provides a useful speedup, but because the fitness evaluation is much shorter than the WriteBack/Select/Breed sub-stage, there is no benefit to increasing the number of parallel fitness evaluations. It is also clear that for small populations there is a limit to the number of parallel fitness evaluations that can be accommodated. However, the situation is reversed in the artificial ant problem and the even-6-parity problem because the time needed for fitness evaluation is much larger than the WriteBack/Select/Breed phase. Clearly, for other problems where the fitness function takes a long time it would be worth devising efficient pipelines for the fitness evaluation functions.

The speedups achieved for the Boolean even-6-parity problem are better than those achieved for the artificial ant problem. This is mainly because the functions in the function set $\mathcal{F}$ for the Boolean even-6-parity problem are simpler than the artificial ant problem, requiring fewer steps to be executed for each instruction. The intrinsic parallelism of Handel-C also allowed the functions to be implemented very efficiently. A second reason for the improved throughput is that the Boolean even-6-parity problem has a longer mean program length than the artificial ant problem. Stallings [Stallings 00] provides an equation that gives the speedup factor of a pipeline with $k$ stages:

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{k+(n-1)} \tag{5.1}$$

where $\tau$ is the maximum stage delay, $k$ is the number of stages in an instruction pipeline and $n$ is the number of instructions executed.

From Equation 5.1 it can be seen that the longer the instruction stream, the better the speedup. For the artificial ant problem, the mean program length was 16 and the pipeline had two stages. This gives a speedup factor for the pipeline of 1.88, while for the Boolean even-6-parity problem the mean program length was 128. Again the pipeline had two stages. This gives a pipeline speedup factor of 1.98.

Equation 4.4 on page 81 was used to predict the worst case number of cycles that would be needed to execute the programs. In all cases, the measured results were less than the predicted worst case results, confirming that the pipeline implementation was effective and that the fitness evaluation for all the problems was the dominant thread of execution.

### 5.4.2 Comparison With a Popular Software GP system.

While this may appear to be an unfair comparison, a lot of work in GP is centered around exploring the detailed operation of GP, which often requires hundreds or thousands of runs with minor parameter changes, and performance is still likely to be an issue even with processors running at 1 GHz and beyond. This comparison was made to see if using a hardware implementation would be of benefit to researchers. The results show that where a fixed problem type needs to be run many times, a hardware implementation using many parallel fitness evaluations could reduce the time required for extended runs . In particular, when compared to lilgp running on a 1.4 GHz Athlon processor, the hardware implementation of the artificial ant problem was 70 times faster. This result shows that despite the availability of fast processors that a hardware GP system would be useful for some hard problems.

## 5.5 Summary

Moving the population storage to off-chip SRAM enabled the design to solve problems, such as the artificial ant and Boolean even-6-parity problems, which require larger population sizes. Pipelining the breed, selection and evaluation phases gives a performance boost to problems that have short evaluation time requirements like the XOR problem. For problems like the artificial ant problem and the Boolean even-6-parity problem that require extended fitness evaluation times, the benefits of using a pipeline are even greater, giving a nearly linear increase in throughput as the number of parallel fitness evaluations is increased.

# Chapter 6

# Behavioural Analysis

Chapters 4 and 5 described two hardware implementations of GP. In both cases the work concentrated on the implementation issues and improving the clock speed of the implementation, but put to one side the performance of the system with respect to its ability to solve GP problems. Now that the raw throughput issues have been addressed it is time to look at how well the hardware implementation performs, in particular the biases introduced by the crossover operator and the *Random Number Generator* (RNG).

This chapter begins with an empirical analysis of the crossover operator. Three crossover operators that limit the lengths of programs are used; the truncating operator used in the work described previously, a standard limiting operator that constrains the lengths of both offspring and a new limiting operator that only constrains the length of one offspring. The latter has some interesting properties that suggest this operator has the effect of limiting code growth in the presence of fitness.

In Chapter 4 it was suggested that the RNG should be looked at more closely. The behaviour of the *Linear Feedback Shift Register* (LFSR) RNG is analysed and compared to six alternative RNGs. The results show that an alternative RNG allows the hardware GP system to produce more correct programs.

## 6.1 Analysis of the Crossover Operator

The hardware design uses a linear program representation with a fixed maximum size. Choosing a fixed maximum size made the storage of programs in on-chip RAM and off-chip RAM efficient and simple to implement. Consequently a method of limiting the program size during crossover was needed. The first implementation, described in the preceding chapters, used a *truncating crossover* operator. This is compared to a second method of limiting lengths, called the *limiting crossover* operator.

The analysis is performed by running a number of experiments and by comparing the results. Two different implementations of GP were used for the experiments. First, a simple program that simulated the effects of GP crossover was used to show the expected program length distributions in the absence of fitness using a linear representation. This is referred to as the *GP simulator*. Second, the hardware implementation described in Chapter 5 was used to obtain results both with and without fitness. The test problems for all the experiments where fitness is used were the artificial ant and the even-6-parity problems described in Chapter 5. In the case of the experiments that do not use fitness, all programs are assigned the same fitness value. This means that all individuals have the same probability of being selected for the breeding phase.

### 6.1.1 Behavioural Analysis

The measurement of overall GP behavior is frequently limited to plotting the average population fitness vs. generation. This is shown for the artificial ant problem, using the hardware implementation described in the previous chapter, in Figure 6.1. The graph shows the average of 500 runs. This will be used as a baseline when looking at changes to the original design.

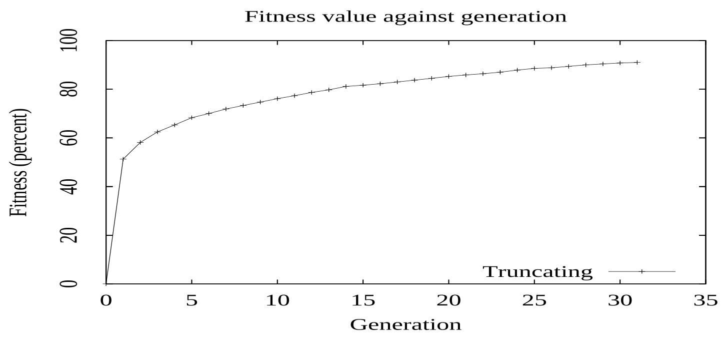


**Figure 6.1:** GP performance of the original design for the artificial ant problem.

However, when looking for the reasons to explain why a feature of an operator or representation has an effect, raw performance gives us a very restricted view of what is happening, and more analytical methods are needed. One such method is to consider one or more aspects of the internal population dynamics during a run. Recently a lot of work has been done to develop exact schema theories for Genetic Programming [Poli 01a] [Poli 01b], which, among other things, give us a description of the expected changes in the program length distribution during a GP run. The asymptotic distribution of program lengths is important to us because it is a way of comparing the sampling behavior (search bias) of different crossover operators and replacement strategies.

Starting with the standard GP model using standard subtree crossover, uniform initial length distribution and ignoring the effects of fitness, Figure 6.2 shows the expected length distribution for generations 0,1,10 and 31. The population was initialised using a uniform random distribution of lengths between 1 and 31 inclusive. In this case there is no maximum program size. This agrees with the results in [Poli 01b] where the distribution asymptotically converges to a discrete Gamma distribution. The mean program length at generation 31 is 16. Note, only program lengths 1 – 32 are shown in the graph, but the tail continues towards infinity.

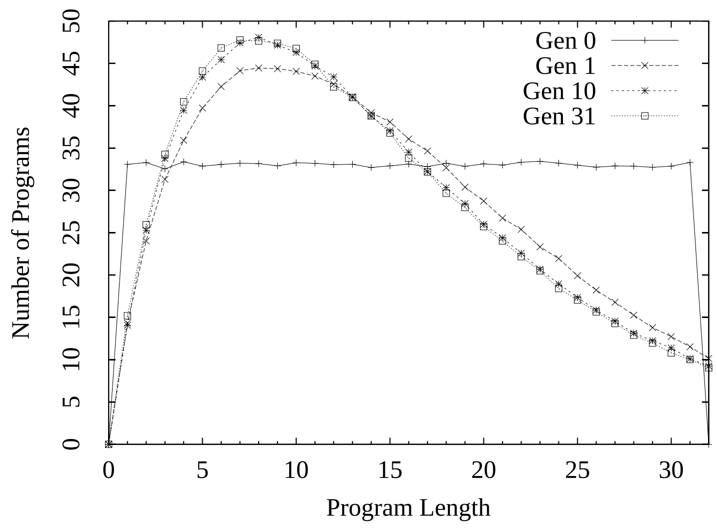


**Figure 6.2:** Program length distribution for standard GP crossover operating on linear strings , using a global replacement strategy, non-steady state and no fitness, using the GP simulator. Average of 500 runs.

### 6.1.2 Truncating Crossover Operator

This crossover operator (described in Chapter 4, Section 4.5.2) ensures programs do not exceed the maximum program length by selecting crossover points in two individuals at random and exchanging the tail portions up to the maximum program length. Crossovers that result in programs exceeding the maximum length are truncated at the maximum length.

The behavior of the hardware implementation using the truncating crossover operator, without fitness, is shown in Figure 6.3.

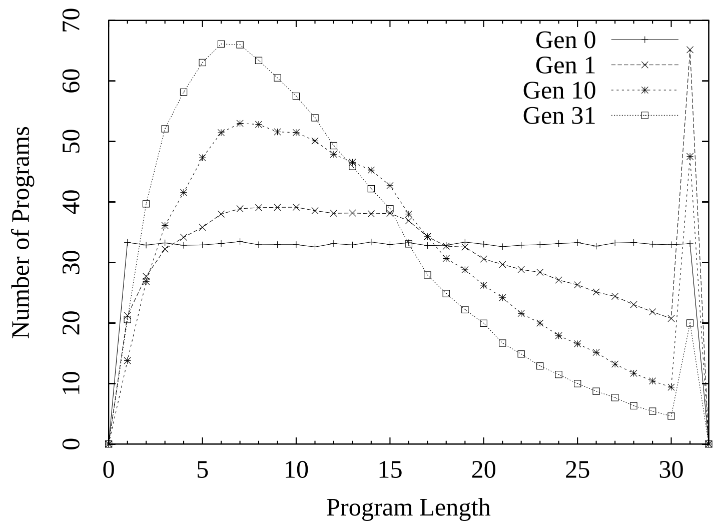


**Figure 6.3:** Program length distribution using truncating crossover using a linear program representation without fitness. From the hardware implementation of the artificial ant problem. Average of 500 runs.

A feature of this result is that there is initially a large peak at the maximum program size of 31, but in subsequent generations the distribution tends to resemble a Gamma distribution like the one in 6.2. However, it is important to note that it not the same Gamma distribution, because the mean program length tends to decrease with this crossover operator. The reason is that with the truncation the amount of genetic material removed from the parents when creating the offspring may be bigger than the amount of genetic material replacing it. At generation 31 the mean program length is 12.5, down from the initial mean length of 16.

When fitness is used, the length distribution for the artificial ant problem changes as shown in Figure 6.4, but it still retains some of the features of a Gamma distribution. The striking feature is the large peak at the maximum program length limit which represents 13% of the total population at generation 31. At generation 31, the mean program length is 16.5, up from the initial mean value of 16.

The corresponding result from the Boolean even-6-parity problem is shown in Figure 6.5, which also shows a large peak at the maximum program length of 255. At generation 63, the mean program length is 144, up from the initial mean value of 128.

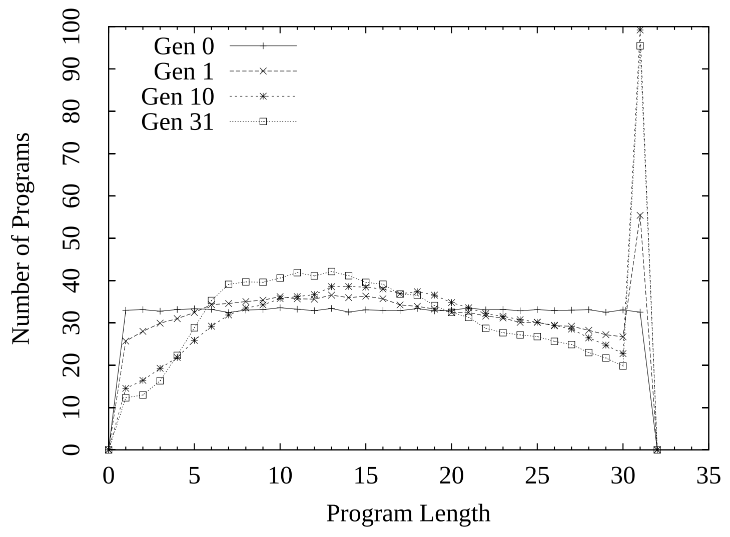


**Figure 6.4:** Program length distribution using truncating crossover using a linear program representation with fitness. From the hardware implementation of the artificial ant problem.

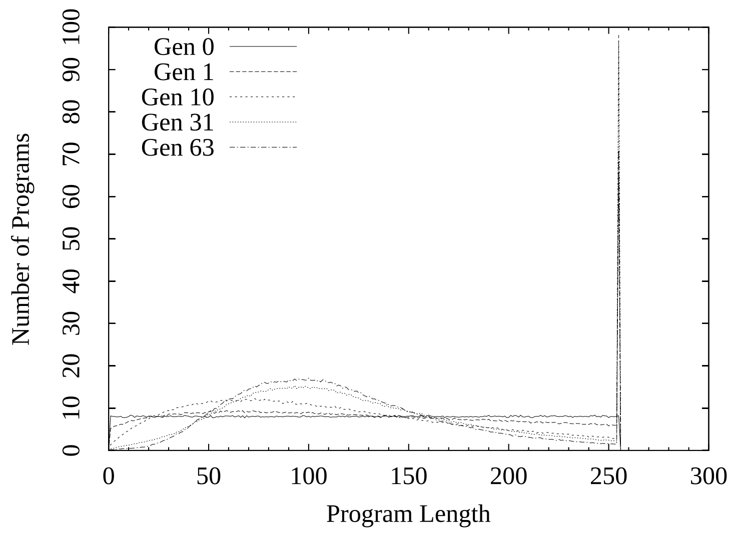


**Figure 6.5:** Program length distribution using truncating crossover using a linear program representation with fitness. From the hardware implementation of the Boolean even-6-parity problem. Average of 500 runs.

### 6.1.3 Limiting Crossover Operator

An alternative method of ensuring that programs do not exceed the fixed limit is to repeatedly choose crossover points until the lengths of both child programs are below the program size limit $L_{\max}$. For two programs $a$ and $b$, with lengths $l_a$ and $l_b$, two crossover points $x_a$ and $x_b$ are chosen so that $0 \leq x_a < l_a$ and $0 \leq x_b < l_b$. After crossover, the new lengths are simply $l'_a = x_a + l_b - x_b$ and $l'_b = x_b + l_a - x_a$. If $l'_a > L_{\max}$ or $l'_b > L_{\max}$ the selection of $x_a$ and $x_b$ is repeated until $l'_a \leq L_{\max}$ AND

$l'_b \leq L_{\max}$. Because both child programs are required to be shorter than $L_{\max}$ this operator is termed the *dual-child limiting* crossover operator.

This is the approach taken in lilgp (versions 1.02 and 1.1) when the `keep_trying` parameter is enabled [Zongker 96] to limit the tree depth and the total number of nodes in a program tree during crossover.

When this method of limiting the program length was implemented in the hardware version, but without fitness, we obtained the distribution shown in Figure 6.6. The mean program lengths for all generations is 16.

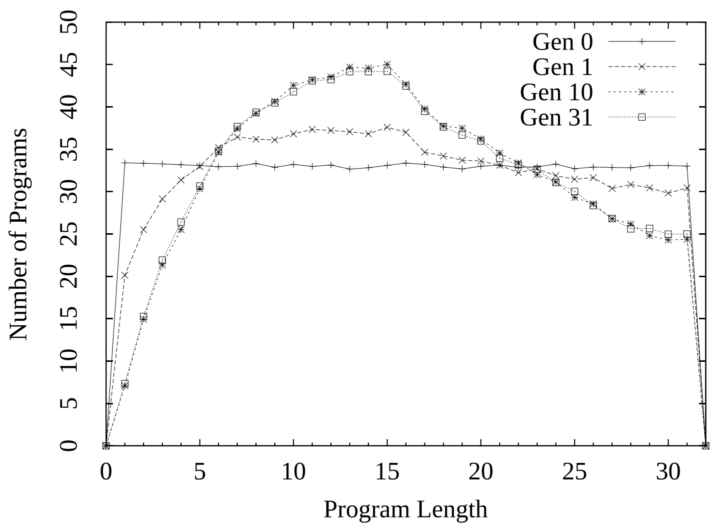


**Figure 6.6:** Program length distribution using limiting crossover without fitness, from the hardware implementation of the artificial ant problem. Average of 500 runs.

When fitness is enabled using the dual-child variant for the artificial ant problem, there is a bias in favor of longer programs, as shown in Figure 6.7. A feature of the results for the artificial ant problem is the sharp rise in program lengths for generations 10 and 31 above length 15, and the peak after length 15. This is likely to be due to the distribution of fitness in the program search space and can be seen as a form of what is commonly termed bloat. Bloat is the term given to the tendency of GP programs to grow in length without any increase in their fitness. This tendency is well documented in the GP literature [Koza 92] [Blickle 94] [Nordin 95a] [Soule 98] [Luke 00a] [Langdon 97]. At generation 31, the mean program length has increased to 20.3, up from the initial mean length of 16.

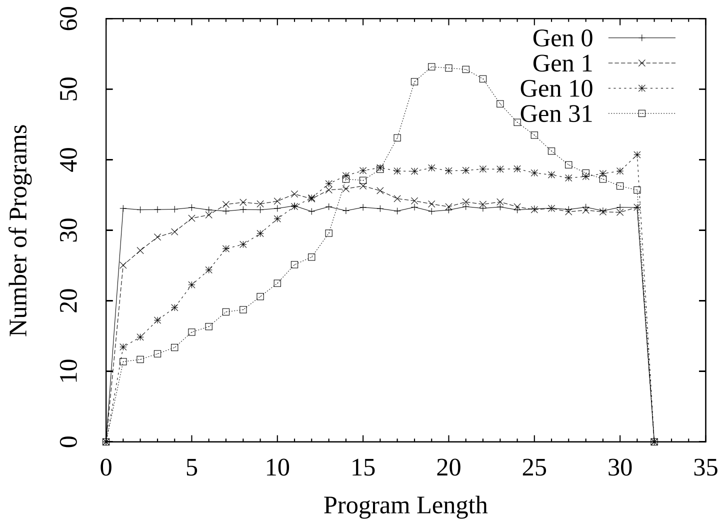


**Figure 6.7:** Program length distribution using limiting crossover with fitness and the dual-child variant. From the hardware implementation of the artificial ant problem. Average of 500 runs.

The results for the Boolean even-6-parity problem are shown in Figure 6.8, which also exhibits a similar peak of program lengths. At generation 63, the mean program length has increased to 162, up from the initial mean length of 128.

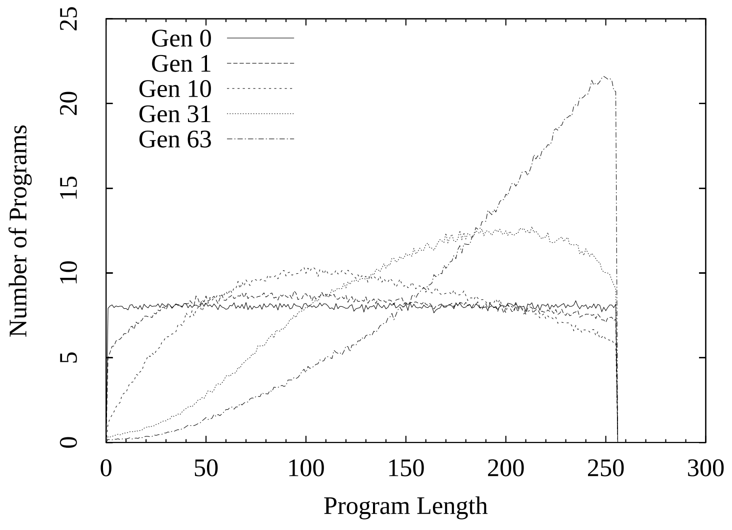


**Figure 6.8:** Program length distribution using limiting crossover with fitness and the dual-child variant. From the hardware implementation of the even-6-parity problem. Average of 500 runs.

### 6.1.4 Modified Limiting Crossover Operator

The above crossover operator (dual-child limiting) requires both of the child programs to have lengths less than $L_{\max}$. A variation of this operator is to require only one of the child programs to be shorter than $L_{\max}$. That is, the selection of $x_a$ and $x_b$ is repeated until $l'_a \leq L_{\max}$ OR $l'_b \leq L_{\max}$. If one of the child programs is larger than the maximum, it is simply discarded and the parent substituted in its place. This is termed the *single-child limiting* crossover operator. When the hardware implementation was modified to incorporate the single-child limiting operator, the result without fitness is shown in Figure 6.9 was obtained. A feature of this result is that the mean program length moves towards smaller values. At generation 31, the mean program length is 11.5, down from the initial mean length of 16.

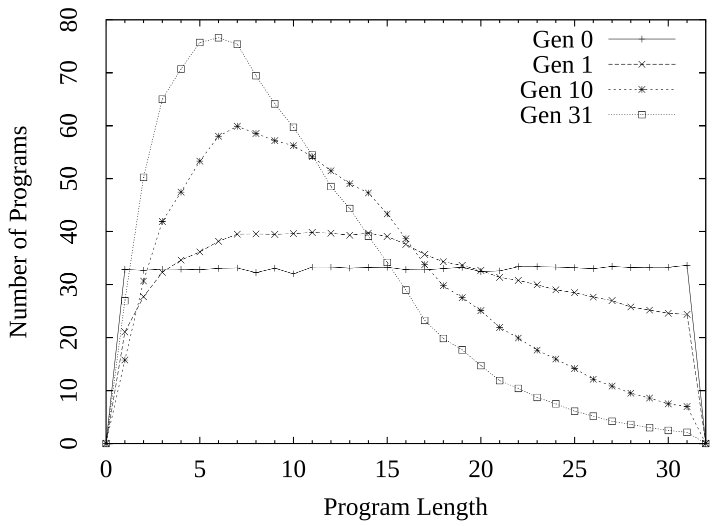


**Figure 6.9:** Program length distribution using limiting crossover without fitness and the single-child variant. From the hardware implementation of the artificial ant problem. Average of 500 runs.

However, when the program length distribution using the single-child variant was plotted for the artificial ant problem, shown in Figure 6.10, the length distribution peaks closer to the mean of $L_{\max}$. At generation 31 the mean program length is 17.2. This behavior is interesting since it appears to have avoided the phenomenon commonly known as bloat. The corresponding plot for the Boolean even-6-parity problem is shown in Figure 6.11. Again there is a peak of lengths near the mean of $L_{\max}$. At generation 63, the mean program length is 140, up from the initial mean length of 128.

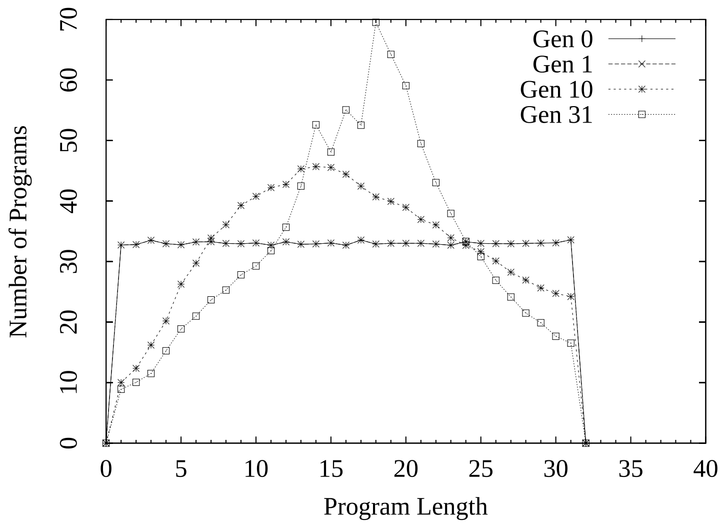


**Figure 6.10:** Program length distribution using limiting crossover with fitness and the single-child variant. From the hardware implementation of the artificial ant problem. Average of 500 runs.

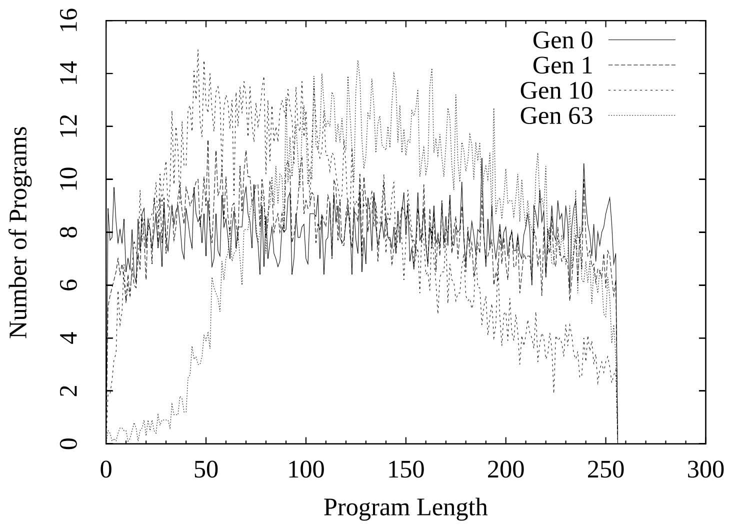


**Figure 6.11:** Program length distribution using limiting crossover with fitness and the single-child variant. From the hardware implementation of the even-6-parity problem. Average of 500 runs.

In both examples of the single child limiting crossover with fitness, the plots (Figures 6.10 and 6.11) are less smooth than the dual child limiting crossover results (Figures 6.7 and 6.8). The plots for the single child limiting crossover suggest some form of instability, suggesting an area for future research.

The effect of using the limiting crossover operator with and without the single-child variant on the mean population fitness using the artificial ant problem is shown in Figure 6.12 together with the original mean population fitness. This graph shows that all three crossover implementations have a similar rate of improvement, with the limiting crossover operator with single-child variant maybe performing slightly better on the artificial ant problem.

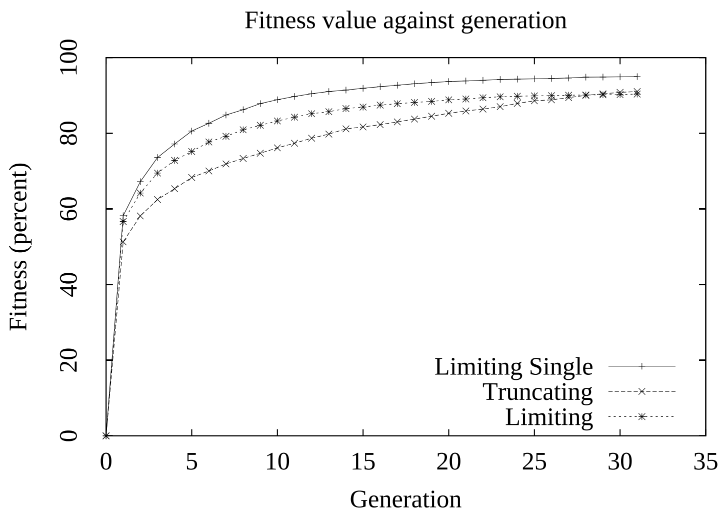


**Figure 6.12:** Comparative mean population fitness of the hardware implementation for the artificial ant problem using truncating crossover, dual child limiting crossover and single child limiting crossover. Average of 500 runs.

Finally, the distribution of 100% correct program lengths was measured for truncating and both limiting crossovers. The hardware implementation was run 500 times, and if a 100% correct program was generated, the length was recorded. These are shown in Figures 6.13, 6.14 and 6.15 respectively for the artificial ant problem.

These plots show that truncating crossover has allowed GP to find more 100% correct programs than the limiting crossover using the dual-child variant. However, when using the single-child variant, limiting crossover found the most 100% correct programs. It is interesting to note that the results shown in Figure 6.12 do not show this difference in the outcome, highlighting the weakness of using the standard measure of performance.

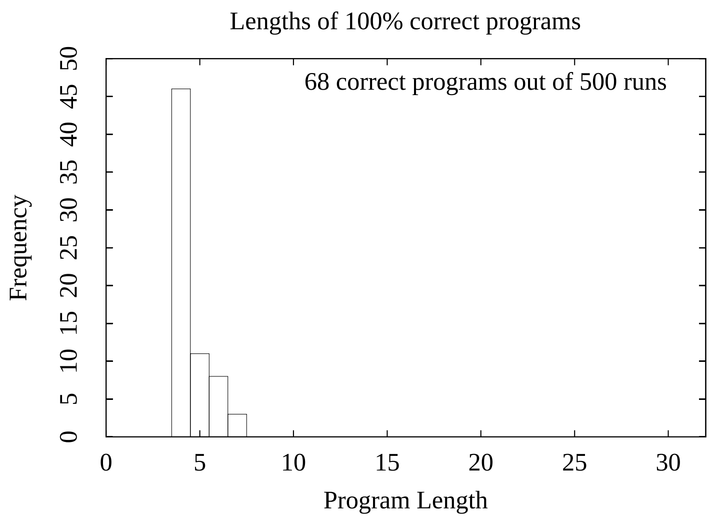


**Figure 6.13:** Distribution of lengths of 100% correct programs using the truncating crossover operator for the artificial ant problem over 500 runs.

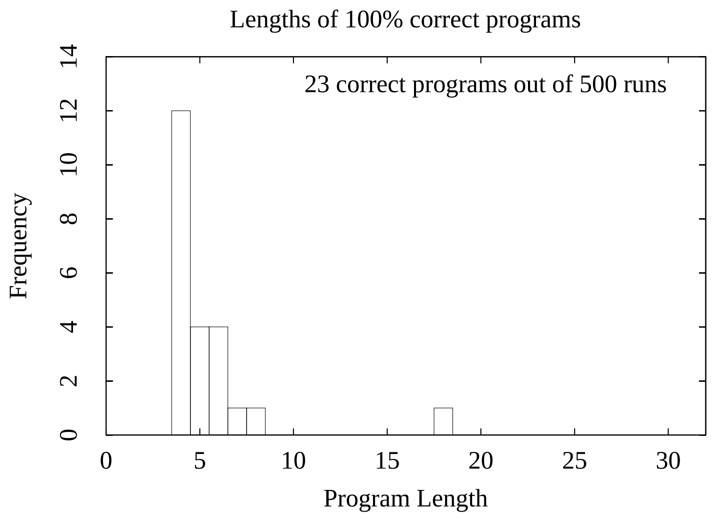


**Figure 6.14:** Distribution of lengths of 100% correct programs using the dual-child variant limiting crossover operator for the artificial ant problem over 500 runs.

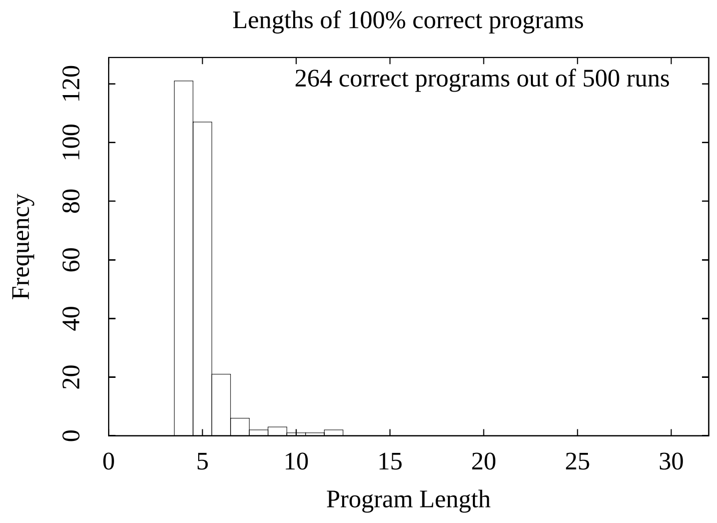


**Figure 6.15:** Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator for the artificial ant problem over 500 runs.

The corresponding program plots for the Boolean even-6-parity problem are shown in Figures 6.16, 6.17 and 6.18 for the truncate, dual-child limiting and the single-child limiting crossovers respectively. Again, from these results the truncating crossover has produced more 100% correct programs than the limiting crossover using the dual child variant, and the single child variant has produced the most 100% correct programs, though the difference is not as marked as for the artificial ant problem. This is possibly due to the fact that the parameters chosen for the Boolean even-6-parity problem were more favourable to finding a solution than the artificial ant problem.

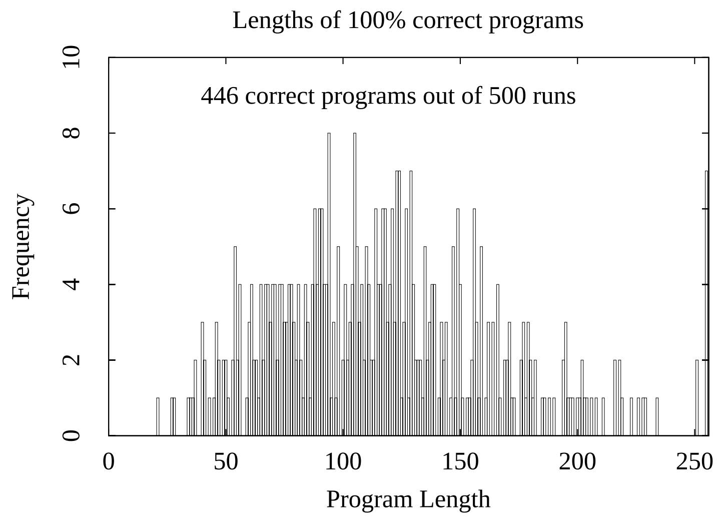


**Figure 6.16:** Distribution of lengths of 100% correct programs using the truncating crossover operator for the Boolean even-6-parity problem over 500 runs.

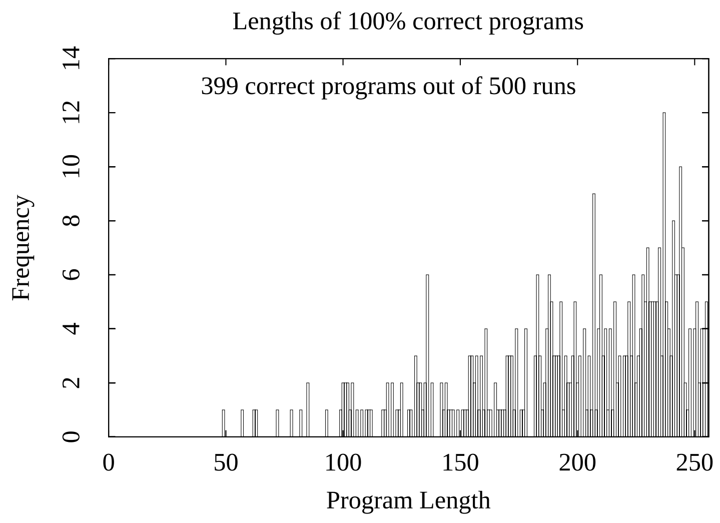


**Figure 6.17:** Distribution of lengths of 100% correct programs using the dual-child variant limiting crossover operator for the even-6-parity problem over 500 runs.

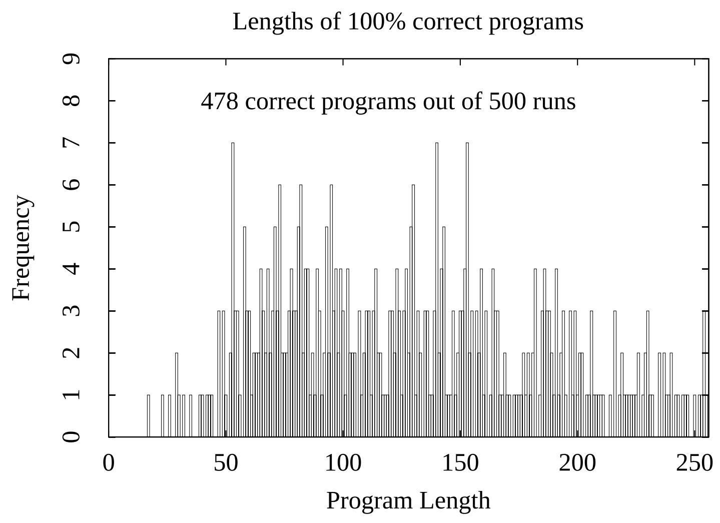


**Figure 6.18:** Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator for the even-6-parity problem over 500 runs.

### 6.1.5 Tuning the Maximum Length Parameter

The results shown in Figures 6.13, 6.14 and 6.15 suggest that for the artificial ant problem implemented in hardware, programs of length 4 or 5 are most likely to be correct, though this can only be confirmed by either exhaustively testing every possible program or by randomly sampling all possible programs. It was then observed that the peak program length in Figure 6.10 was larger than length 4. It was then conjectured that if the maximum program length was changed so that the peak was moved so that it was closer to the most frequently successful program, that GP would find even more successful programs. The experiments using the artificial ant problem using the single child limiting crossover were repeated but using maximum lengths of 16 and 8. The results of running the hardware implementation with these modified lengths are shown in Figures 6.19 and 6.20 for maximum lengths 16 and 8 respectively.

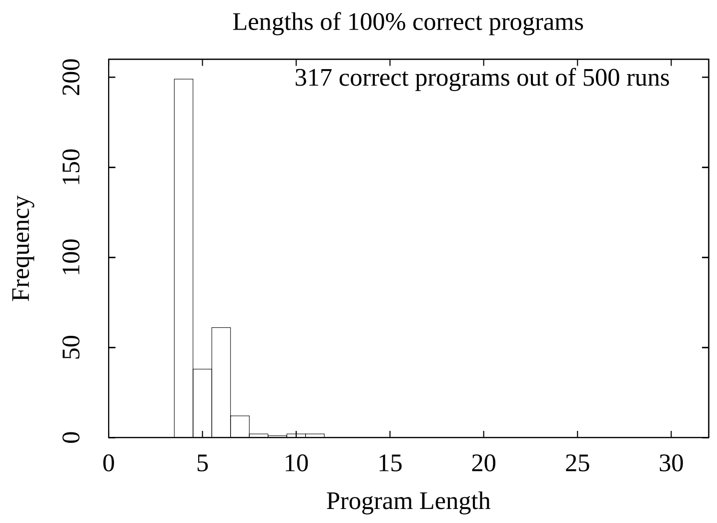


**Figure 6.19:** Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator and a length limit of 16 for the artificial ant problem over 500 runs.

This confirmed the idea that by limiting the program lengths that GP is allowed to create, that GP could create more 100% correct programs. The corresponding program length distributions are shown in Figures 6.21 and 6.22. Again these show that the program length distribution peaks near the peak of the successful programs.

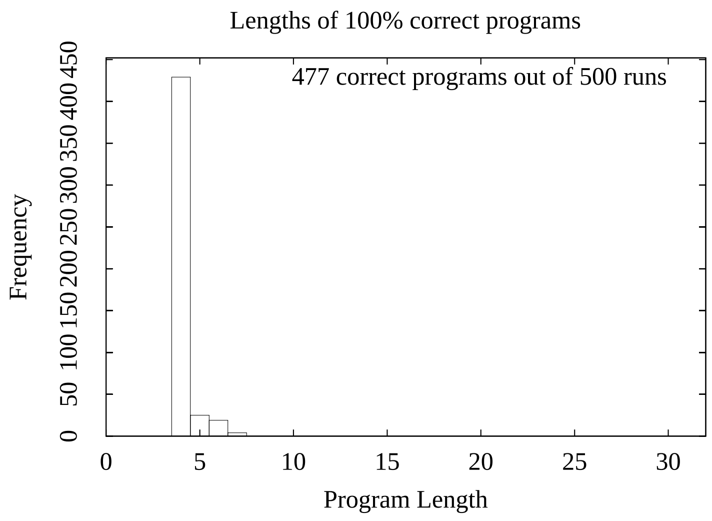


**Figure 6.20:** Distribution of lengths of 100% correct programs using the the single-child variant limiting crossover operator and a length limit of 8 for the artificial ant problem over 500 runs.

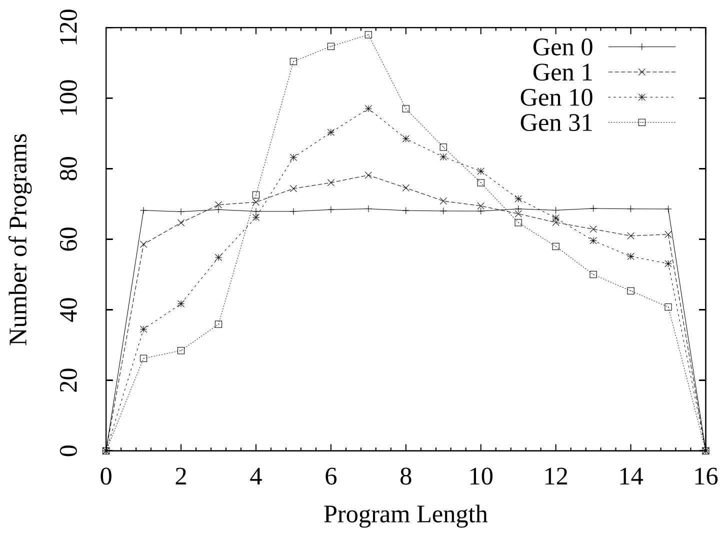


**Figure 6.21:** Program length distribution using limiting crossover with fitness and the single-child variant. Maximum length limited to 16. From the hardware implementation of the artificial ant problem over 500 runs.

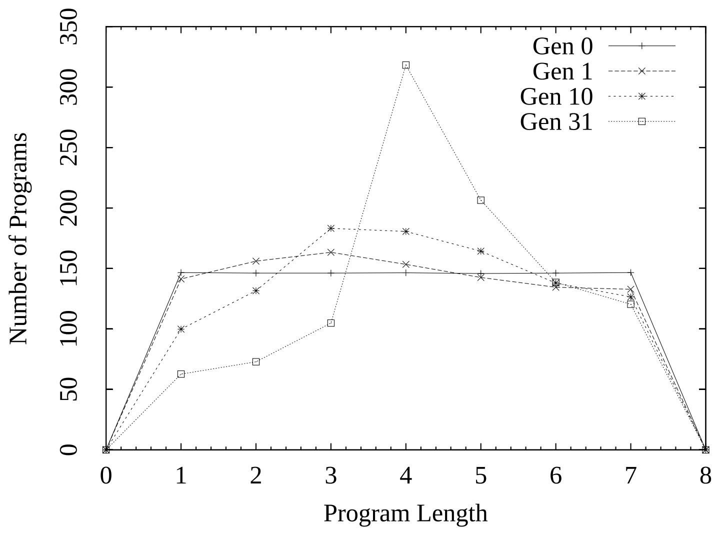


**Figure 6.22:** Program length distribution using limiting crossover with fitness and the single-child variant. Maximum length limited to 8. From the hardware implementation of the artificial ant problem over 500 runs.

## 6.2 Discussion

The differences between the dual-child and single-child variants can be explained by considering first the dual-child case. Starting with a uniform distribution of program lengths, the average program length is given by $L_{\text{avg}} = \frac{L_{\text{max}}}{2}$ and the average crossover point is $\frac{L_{\text{avg}}}{2}$. Every crossover produces two offspring, the average length of which is $\frac{L_{\text{max}}}{2}$, with one small and one large program produced. When one of the offspring exceeds $L_{\text{max}}$ both crossover points are re-selected until both programs satisfy the length constraint. The result is that the average program length using this crossover will remain $\frac{L_{\text{max}}}{2}$. However, in the single-child case, only one child needs to meet the length constraint. With one long and one short offspring, the short offspring will be more likely to satisfy the constraint and so be retained for propagation. Because the shorter program is preferred, the mean program length will tend to continually decrease. In summary, in the absence of fitness, the single-child variant selects programs that are on average smaller than $\frac{L_{\text{max}}}{2}$.

In the presence of fitness we believe that this pressure to decrease the mean program length competes with the well documented tendency of GP programs to grow in the presence of fitness in a variety of problems. The result is that when using the single length constraint and an upper bound on the program length, the program length distribution does not have a strong bias to longer lengths and is more likely to remain in an equilibrium.

A side effect of using the single child variant is that when a long program is rejected, a copy of the parent is propagated to the next generation. This means that even if crossover is used as the only operator, a proportion of reproduction will be present. The effect of this bias has not been investigated.

A practical penalty of the dual child limiting crossover approach is that multiple passes may be required to obtain two crossover points that satisfy the length constraints. Depending on the implementation this could have an impact on the time needed to complete a GP run. In practice, for most problems the time required for crossover in a standard GP system is much smaller that the time for evaluating programs, and so will only extend the time required by a small factor. In the hardware implementation, crossover is performed in parallel with evaluation, so there will be no impact for most problems, where fitness evaluation takes longer than selection and breeding. For the artificial ant problem and the Boolean even-6-parity problem implemented in hardware, the limiting crossover operators did not have any effect on the overall behavior of the design, both the clock speed and number of clock cycles remained the same as the truncating crossover implementation. It is worth noting that the single-child limiting crossover will need zero iterations to find a legal offspring, so this will have a smaller effect on the overall performance in a serial implementation of the algorithm.

The effect of adjusting the program length limit so that the peak in the length distribution is closer to the peak of correct program lengths suggests that allowing programs to be unlimited in length may be detrimental to using GP effectively because of the time needed for carrying out unnecessary processing.

The speedup that hardware GP had over a software implementation meant that it was feasible to perform 500 runs of the experiments in a reasonable time. Using the elapsed times from Table 5.7 on page 101, and ignoring the overhead of the scripts used to run the experiments, 500 runs of the Boolean even-6-parity problem took $1.18 \times 500 = 590$ s, or nearly 10 minutes. Using a software implementation on a processor running at 1 GHz would have required $454/5 \times 500 = 45\,400$ s or nearly 12 hours.

## 6.3 Analysis of the Random Number Generator

This section analyses the *Random Number Generator* (RNG), or more correctly the *Pseudo Random Number Generator* (PRNG), chosen for the work in Chapters 4, 5 and Section 6.1 of this chapter. The behaviour of the system is then compared using several other RNGs. This section starts with a survey of RNGs and PRNGs that have previously been used to generate random or pseudo-random sequences in hardware.

### 6.3.1 Previous Work on Random Number Generator Quality

The effect of RNG quality on the performance of GA and GP has been investigated in [Meysenburg 97a], [Meysenburg 97b], [Meysenburg 99b], [Meysenburg 99a], [Meysenburg 02], [Daida 97], [Daida 99] and [Cantú-Paz 02]. The general conclusions from these investigations are that the RNG used can have an effect on the performance of GAs and GP. However, the effect is often statistically insignificant. Furthermore, the effect is not always correlated with the quality of the RNG, as measured using standard statistical tests.

### 6.3.2 Previous Work on Using Random Number Generators in Hardware

There are three dominant methods of generating random or pseudo-random sequences in hardware: LFSR or Tauseworth generators, generators based on *Cellular Automata* (CA), and generators that exploit a physical noisy phenomenon such as the noise from an electrical circuit, radioactive decay or atmospheric radio noise.

#### LFSR generators

These were used by Maruyama *et al.* [Maruyama 99]. In their paper they referred to the generator as a m-sequence, or maximal sequence. This means that the generator of length $n$ generates $2^n - 1$ numbers. Graham [Graham 96] implemented a single cycle LFSR.

An interesting hybrid was used by Tommiska and Vuori [Tommiska 96] where three coupled LFSRs were used to provide a random sequence. An interesting feature of this work is that the RNG was combined with a source of noise. The amplified noise from a diode was fed into an analogue to digital converter, and the resulting digital values were used to seed the RNG, and also added to the LFSR at intervals to perturb its state.

The manufacturers of FPGAs provide example designs of LFSRs to be used as random sequence generators. For example Xilinx [Xilinx 01a], and Altera [Altera 01] provide *Hardware Design Language* (HDL) code for LFSRs.

### CA generators

Aporntewan [Aporntewan 01] used a one dimensional 2-state CA. Scott [Scott 94] used a CA that consisted of 16 cells with 2 different rules as described by Wolfram [Wolfram 84]. Shackleford *et al.* [Shackleford 01] implemented a CA based on the work by Wolfram [Wolfram 86]. Yoshida *et al.* [Yoshida 01] also used a CA system combining two CAs that used two different rules.

### Other generators

L'Ecuyer in [L'Ecuyer 88] and [L'Ecuyer 90] showed that a lagged Fibonacci generator gives better results than other generators. However, Chu and Jones [Chu 99] showed that when implemented in an FPGA, a lagged Fibonacci generator requires a large amount of FPGA resources and that LFSRs are much more space efficient.

As already noted, Tommiska and Vuori [Tommiska 96] used the noise from a diode to provide a source of true randomness to seed the LFSR generator. The generator was also perturbed at intervals using the same random noise source. It would appear that digital circuits, clocked at a regular frequency, would themselves make poor random number generators, but the physics of semiconductor devices means that there is always a low-level of noise introduced into digital semiconductor circuits. This can be exploited to produce random sequences, as demonstrated by Fairfield *et al.* [Fairfield 84], by exploiting the frequency instabilities inherent in LSI devices. However, they report that the generator is slow – producing random integers at a rate of around 3 bytes per second. Thermal noise is another frequently used method of generating random sequences and was used by Bright and Turton [Bright 00].

Other examples of generators that have used natural sources of noise include the radioactive beta decay of radioactive isotopes such as Krypton-85 [Fourmilab 02] and the visually interesting patterns generated by Lava lamps [Lavarand 02].

Because generators that use natural noise are often slow, they have the drawback that they are unable to generate random numbers at a high enough rate to be used as the primary source in an evolutionary algorithm, so instead they are often used to seed and/or perturb the standard PRNGs. They also often require specialised hardware to implement them.

## 6.3.3 Experimental Setup

The performance of the various RNGs in this section was evaluated using three methods: using the Diehard test suite, using the artificial ant problem from Chapter 5 and using Handel-C to implement the generators.

### Diehard test suite

The Diehard test suite maintained by Marsaglia [Marsaglia 01] is a de-facto standard for evaluating RNGs and was used to gauge the general performance of the RNG. The suite consists of up to 15 tests that are modeled on applications of random numbers. All the RNGs considered in this paper were implemented in ISO-C and were submitted to all 15 tests. The test method for Diehard is similar to that described in Meysenburg and Foster [Meysenburg 99a]. Each RNG was used to generate a binary file of about 10 MiB. Each Diehard test produces one or more $p$-values(1). A $p$-value can be considered good, bad, or suspect. Meysenburg used a scheme by Johnson [Johnson 96] which assigns a score to a $p$-value as follows. If $p \geq 0.998$ then it is classified as bad. If $0.95 \leq p < 0.998$ then it is classified as suspect. All other $p$-values are classified as good. Using this classification, every bad $p$-value scores 4, every suspect $p$-value scores 2 and good $p$-values score zero. For each RNG, the scores for each test were summed, and the total for each RNG is the sum of all the test scores for that RNG. Using this scheme, high scores indicate a poor RNG and low scores indicate a good RNG. The results for each test are given in Appendix D.

### Incorporation of RNGs into hardware GP

Each RNG was then incorporated into the hardware GP system described in Chapter 5 using the artificial ant problem with the Santa Fe trail. The problem was run 500 times, and the number of correct programs that appeared was recorded. This is used as a measure of how well the RNG performs in a GP system. In all cases, the population size is 1024, the maximum program length is 31 and all experiments were run for 31 generations. The ant was allocated 600 timesteps. The probability

(1) $p$-values are given by $p = F(X)$, where $F$ is the assumed distribution of the sample random variable $X$ — often normal. The $p$-value should be uniform on $[0,1)$ if the input fi le contains truly independent random bits.

of selecting crossover was 66%, mutation 12.5% and reproduction 21.5%. The crossover operator used the truncating method of limiting the maximum program length, as described in Section 6.1.2.

**RNG implementation using Handel-C**

Each RNG was also implemented as a stand alone application for an FPGA using Handel-C, and the number of slices used and the maximum attainable clock frequency was recorded. This gives a measure of the hardware resources needed to implement the RNG, and also an indication of the logic depth required.

## 6.4 Random Number Generator Implementations

This section considers several different RNGs including the generators commonly used in hardware implementations.

### 6.4.1 LFSR RNG

The details of the LFSR generator have already been given in Section 4.3 on page 63. The obvious weakness of this type of RNG is that sequential values fail the serial test described by Knuth [Knuth 69]. At any time step $t$ there are two equally likely values for time $t+1$. If for an LFSR of length $n$ at time $t$ the value is $v$, then at time $t+1$ the value will be $v/2$ or $v/2+2^{n-1}$. All the serial tests in this section were performed by generating a file of random numbers using each of the generators. The file created was about 11 MiB in size (i.e. about 2.8 million 32 bit numbers). The results for the LFSR RNG are shown in Figure 6.23 where pairs of values $v_t$ and $v_{t+1}$ are plotted.

It can be seen that for any value $v_t$ there are only two possible values of $v_{t+1}$. Though the random number generator runs in parallel with the main GP machine, it is possible to access sequential values when creating an initial program, or when choosing crossover points. There is then a possibility of introducing a potentially degrading bias by using such an RNG.

### 6.4.2 Multiple LFSRs

One method of obtaining better serial test results when using an LFSR is to allow the LFSR to run for a number of cycles before reading another number. That is, to produce $n$ random bits from an

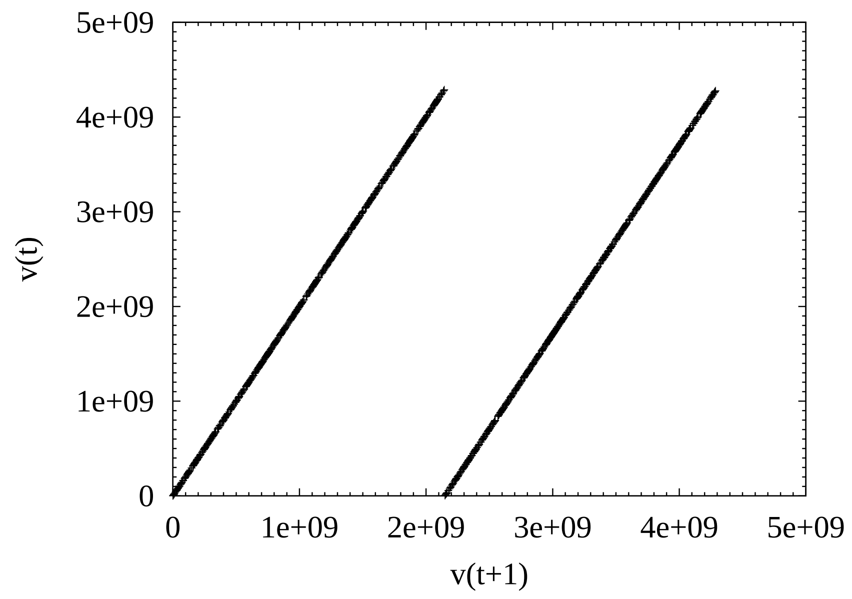


**Figure 6.23:** Serial test of a simple LFSR RNG.

LFSR of length *m* (where $n \leq m$), the LFSR should be allowed to run for *n* cycles between reading the bits. Since this would limit the rate at which random numbers could be generated in the present design it is not explored any further. However, an equivalent result can be obtained by implementing *n* LFSRs of length *m* and using a single bit from each LFSR at each time step. This can also be done using a single long LFSR of $n \times m$ bits, [Stiliadis 96] effectively implementing *n* parallel LFSRs. However, implementing a long shift register in a Xilinx Virtex FPGA is not efficient because the look up tables can implement a 16 bit shift register very easily, but longer shift registers require more extensive routing resources.

The effect of using a better RNG was investigated by implementing 32 16 bit LFSR machines that run in parallel, and initialising each LFSR to a different value. Bit32 from each LFSR is used to construct a 32 bit random number. The serial test result is shown in Figure 6.24, which shows the serial test result for 32 LFSRs is better than the single LFSR. This generator is referred to as the 32LFSR.

### 6.4.3 Cellular Automata RNG

Another popular RNG for hardware implementations is based on Cellular Automata (CA). A one-dimensional (1D) CA consists of a string of cells. Each cell has two neighbours - left and right, or in some literature west and east respectively. At each time step, the value of any cell *c* is given

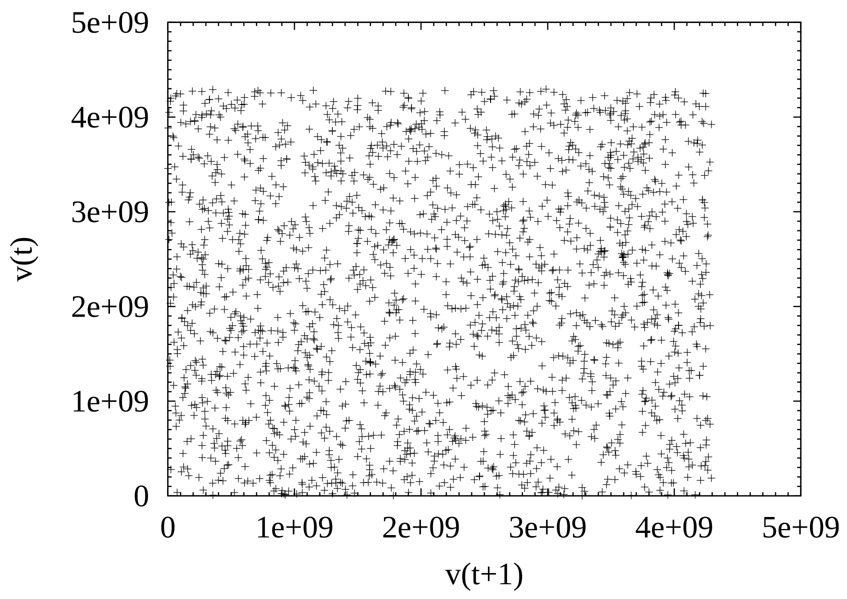


**Figure 6.24:** Serial test for an RNG using 16 parallel LFSRs

by a rule. For this implementation, rule 30 is used, which states that for any cell $c$ at time $t$, $c_{t+1} = ((west_t + c_t) \oplus east_t)$, where $\oplus$ denotes the exclusive OR function. In practice the CA is implemented using a single 32 bit word, and for cell 0, its right-hand neighbour is cell 31, and similarly for cell 31 its left hand neighbour is cell 0. Figure 6.25 shows the result of running this RNG using the serial test. As in the simple LFSR RNG there is a distinct pattern to the numbers, but for most values of $v_t$ there are several possible values for $v_{t+1}$. This generator is referred to as 1DCA.

### 6.4.4 Multiple CA generators

As in the case of the LFSR RNG, if several CAs are combined, it would be expected that the results would be much better than the single 1DCA. For this test, 32 CAs were implemented, and by taking one bit from each CA, a 32 bit random number can be generated. The result for the serial test appears to be much more random, as shown in Figure 6.26. Each CA is initialised with a different pattern. This generator is referred to as the 32CA

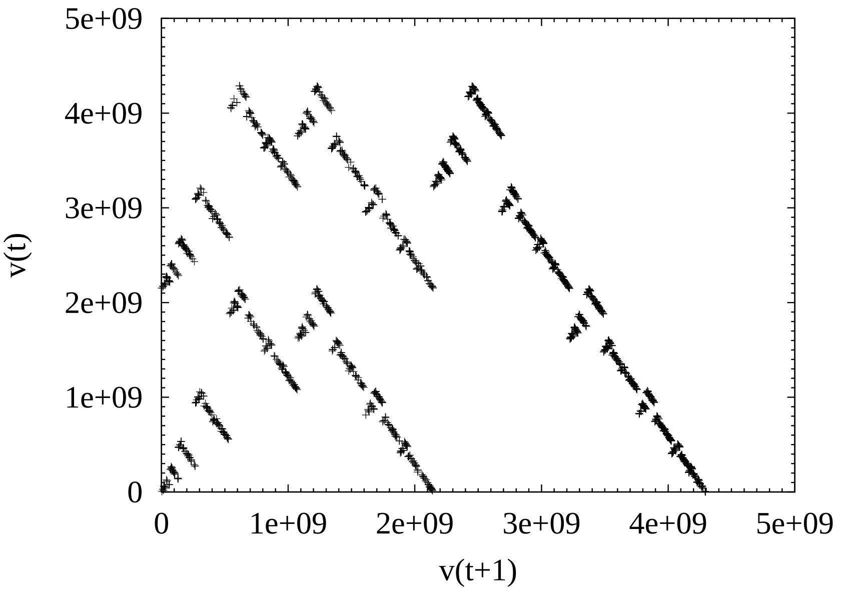


**Figure 6.25:** Serial test for a 1DCA RNG

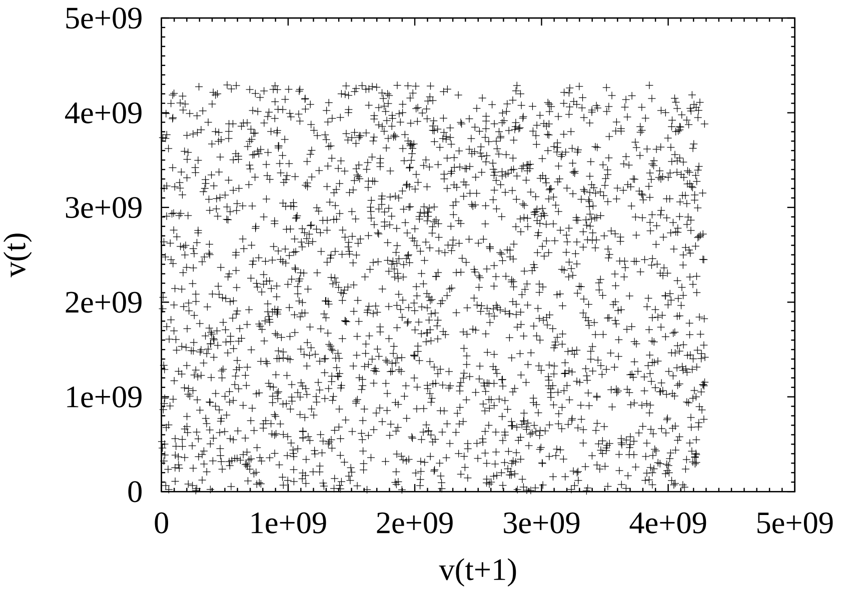


**Figure 6.26:** Serial test for a 32CA

### 6.4.5 Standard C RNGs

Another frequently used RNG is the *Linear Congruential* generator that is often found in implementations of the standard C library, following the illustration of an example `rand()` function in [Kernighan 88]. The general equation for these is:

$$I_{j+1} = (aI_j + c) \bmod m \tag{6.1}$$

where $a, c$ and $m$ are constants chosen to produce a maximal length RNG. However, as pointed out by many authors (eg: [Press 86]) these generators are not good. A factor against implementing such a generator in hardware is that it requires one addition, one multiplication, and one modulus operator, which in Handel-C would consume a large amount of silicon and because of the deep logic produced when using 32 bit words, would be slow. An alternative given by [Press 86] avoids the modulus operator, and is called the Even Quicker Generator (EQG). It is claimed that this is about as good as any 32 bit linear congruential generator. Its equation is:

$$I_{j+1} = aI_j + c \tag{6.2}$$

Values for $a = 1664525$ and $c = 1013904223$ are suggested. Note that there is an implicit mod $2^{32}$ operator when executed on a 32 bit processor, so the algorithm is equivalent to 6.1.

As a sanity check that the experimental method of ranking the RNGs using Diehard was the same as that used by Meysenburg, the generator known as “the mother of all generators” was also implemented and run against the Diehard suite. This is a multiply with carry generator and is described by Marsaglia [Marsaglia 94]. It was not implemented in the hardware GP system because of its complexity.

### 6.4.6 Non Random Sequences

Until now we have considered pseudo random sequences. These are sequences where it is hard to guess the next number in a sequence. As an experiment, a further set of runs were performed with an obviously non-random number generator. For this a sequential generator that output the sequence $n, n+1, n+2, \ldots$ was used. Rather surprisingly this also worked to produce 100% correct programs, though substantially fewer than the other generators achieved.

### 6.4.7 Truly Random Sequences

All the RNGs considered so far are not true random sequences, relying on the manipulation of objects of finite size, and so fail one or more of the Diehard battery of tests. So a set of random numbers was obtained from a source generated by using the atmospheric noise captured by a radio receiver [random.org 02]. Each GP run for the artificial ant problem needs about half a million random numbers, so a block of 10 MiB was downloaded from `www.random.org` and a randomly selected 2 MiB block was transferred to one of the SRAM on the FPGA system using DMA. The FPGA read this block sequentially to get its random numbers.

## 6.5 Experimental Results

The results from running the Diehard tests are summarised in Table 6.1. This shows the total results for each test and ranks them according to their Diehard score. The worst score that Diehard can award is 876. Note that the Sequential generator fails all Diehard tests. Appendix D gives the results of the Diehard tests in full.

**Table 6.1:** Summary results of running the Diehard tests on the random number generators.

| RNG | Score |
|---|---:|
| Mother | 20 |
| True | 22 |
| 32LFSR | 162 |
| EQG | 288 |
| 32CA | 640 |
| 1DCA | 676 |
| LFSR | 756 |
| Sequential(2) | 876 |

The number of correct programs that were produced by each random number generator was recorded and is shown in Table 6.2. The results are ranked according to the number of correct programs produced. The table also shows the slice count for the RNG implemented using Handel-C and the maximum frequency as reported by the place and route tools. The slice count is a vendor dependent measure of the number of FPGA blocks used. The clock rate is an indication of the logic depth required to implement the generator, with deeper logic exhibiting a greater gate delay and

(2)Note. This is not a true RNG. See Section 6.4.6

therefore a lower maximum clock rate. The slice count and frequency for the true RNG assumes that the source of random numbers is supplied by an external device to the FPGA, and that the FPGA simply needs to read the value from a port and write it to a register.

**Table 6.2:** Summary of GP performance for all random number generators tested from 500 runs.

| RNG | Correct | Slice | Clock rate |
|---|---|---|---|
| 32CA | 82 | 284 | 105 MHz |
| True | 81 | 6 | >200 MHz |
| 32LFSR | 79 | 130 | 134 MHz |
| EQG | 78 | 288 | 42 MHz |
| ID CA | 78 | 22 | 125 MHz |
| LFSR | 68 | 18 | 188 MHz |
| Sequential | 39 | 21 | 155 MHz |

## 6.6 Discussion

The Diehard score obtained by the mother RNG was close to that obtained by Meysenburg, the difference being explained by the fact that Meysenburg used the average of 32 runs, while the work described here used a single run. Despite this difference, this confirms that the experimental method used for ranking the RNGs using Diehard is comparable.

Despite the apparently serious deficiencies found in both the simple LFSR used in the original implementation and the simple one dimensional CA random number generator, the overall effect of implementing a more sophisticated RNG on the overall GP performance appeared to be small. This result generally agrees with the work by Meysenburg and Foster [Meysenburg 99a], with the exception that they did not consider a single-cycle LFSR. The single-cycle LFSR performs the least well of the RNGs.

What was surprising was the emergence of solutions when a non-random sequence was used. Clearly, a non-random sequence does not allow GP to operate as efficiently in terms of producing 100% correct programs, presumably because of the failure to explore some areas of the search space. Nevertheless, the sequential generator allowed GP to explore enough of the search space to discover viable solutions.

Despite the small differences in performance between the RNGS, with the exception of the single-cycle LFSR and the sequential generator, from the results we can say that using a different

RNG from the single LFSR would improve the performance of the hardware GP implementation, and that an RNG based on multiple LFSRs or multiple CAs would be a better choice for a hardware GP system. The use of a truly random number source did not appear to improve performance over the 1DCA, 32CA and 32LFSR RNGs. This provides more evidence countering the notion that GP needs a very high quality RNG.

Table 6.2 shows that the difference in GP performance between the 32CA, True, 32LFSR, EQG and 1DCA generators is small. However, these 5 generators have very different Diehard scores, so there does not appear to be a straightforward relationship between the Diehard score and the performance of GP. This raises a question about the role that RNGs play in GP. Is a RNG that scores well in some of the standard tests for randomness the best RNG for GP?

When looking at the FPGA slice counts and maximum clock rates, the 32LFSR uses about half the FPGA resources that the 32CA does, and exhibits a smaller delay than the 32CA. This result agrees with Yoshida *et al.* [Yoshida 01] where they found an LFSR RNG used fewer gates than a CA base RNG. As predicted, the EQG uses the most FPGA resources and has very deep logic, meaning that it can only run at a much slower rate than any of the other generators. The EQG RNG could be re-implemented in the FPGA using pipelines to achieve much higher clock rate, but since it performed no better than the 32CA or 32LFSR, this was not investigated any further.

## 6.7 Summary

This chapter has presented two behavioural analysis of the hardware GP system. First, the behaviour was analysed using the program length distributions, and second, the impact of the random number generator was evaluated.

### 6.7.1 Program Length Distribution

This analysis, based on measuring the program length distributions, was inspired by the results from the work on a general schema theory of GP. It has led to an implementation of a crossover operator that constrains the maximum program lengths in a natural way, without needing to know anything about the program structure. For the artificial ant problem and Boolean even-6-parity problem another mechanism has been discovered that avoids the effects of unconstrained program growth, and also also yields more correct programs than the other crossover operators examined.

In conclusion, all three crossover operators are effective in the hardware implementation when applied to the artificial ant problem, with the single-child limiting crossover performing ahead of the other two. The behavior of the single-child limiting crossover in the presence of fitness is interesting and suggests another mechanism for controlling code growth.

### 6.7.2 Random Number Generator

The main conclusion from this investigation is that for the hardware GP system, the simple LFSR used in the original design can be improved upon by using a generator based on multiple LFSRs, multiple CAs, or if available, a high speed source of true random numbers. A secondary conclusion is that with the exception of the non-random sequence and the LFSR generator, there is no significant difference in GP performance when different hardware RNGs are used.

# Chapter 7

# Economic Analysis of Hardware GP

Previous chapters presented the results of implementing GP in hardware mainly in terms of throughput and behaviour. Taken in isolation, the results suggest that there is a strong case for implementing GP in hardware, especially where the user wants to minimise the time to run an experiment. However, using Handel-C and FPGAs to implement GP involves additional costs when compared with a software implementation of GP. These costs may be broken down into the capital cost of the equipment and software, the time needed to learn about Handel-C and FPGAs, and the cost of the additional development time required when using Handel-C.

The chapter begins with a review of the capital costs, followed by a discussion on the changes that need to be made to traditional software engineering when using Handel-C. The economic factors that affect the suitability of using a hardware implementation of GP are then analysed using two problems. The economics are viewed from the perspective of two groups of users. First, the academic researcher who is exploring GP, and who would benefit from the potential for reduced run times. This user will typically be able to make use of academic discounts for equipment and software. The second user group is concerned primarily with exploiting GP in a commercial or industrial context.

## 7.1 Hardware and Software Costs

Two items of equipment were needed to implement the work described in Chapters 4, 5 and 6: a high performance workstation, and the FPGA development board. In addition, two software tools were required: Handel-C and the Xilinx FPGA synthesis tools. Because the price of hardware is continually changing, and different vendors have different pricing policies, only general hardware pricing information is provided in this section. Similarly, because some software vendors do not make their pricing publicly available, no detailed pricing information for the software can be provided.

Handel-C and the FPGA vendors synthesis tools typically require a highly specified workstation. To illustrate this, a design from Section 5.3.2 that implemented the artificial ant problem is used as an example. This example used 32 parallel evaluations which required nearly 80% of the total slices available on the FPGA. To complete the compilation of the design using Handel-C, a PC with 750 MiB of RAM was required. A PC with a 1.4 GHz Athlon CPU required nearly four hours to complete the place and route. This is in contrast to a modest 500 MHz Pentium machine capable of compiling and running lilgp and other popular GP packages. However, smaller designs, for example the XOR problem with two parallel fitness evaluations in Chapter 5, could be compiled and synthesised using a 500 MHz PC with 256 MiB of memory in about 10 minutes.

There is a wide variety of FPGA boards available from vendors, though many smaller FPGAs, for example the Spartan II 200K-gate FPGA fitted to some low cost boards, do not have sufficient resources to host the GP system described in Chapters 4 and 5 without some modifications. Board costs typically range from €200 for the simple boards fitted with small devices to over €10,000 for boards fitted with the largest devices.

The costs for Handel-C and the synthesis tools must also be added to the total capital cost. For the academic researcher, the tool vendors often very generously make these available at low or zero cost. However, for the commercial user, these tools can represent a significant cost.

## 7.2 Software Engineering Issues

Handel-C is designed to shield the programmer from the internal details of FPGAs and low level hardware logic programming. In this regard it performs well, allowing the designer to concentrate on the functionality required in an FPGA and still use an algorithmic approach to programming. However, this approach has its limitations, particularly when it comes to issues of performance that have traditionally been the preserve of hardware design engineers.

To understand the effect that using FPGAs and Handel-C can have on engineer productivity, this section will consider the additional work needed by an engineer implementing a hypothetical program in software who then translates the same program in hardware using Handel-C. This approach is used because one of the potential user groups of Handel-C are software engineers who want to use FPGAs as part of a design. The assumptions made about the engineer are that they are competent in the use of a popular imperative programming language such as C, they have a good understanding of the problem, and that there is a comprehensive and accurate statement of the requirements for the program. In this scenario it is assumed that the software engineer has designed the program using well established techniques, and implemented and tested the program using the C language and standard tools. A straightforward translation, or port, of the program to Handel-C will probably require some changes to accommodate the lack of a stack, restricted data types, limited data memory capacity and limited code space. This will mean the engineer having to learn about the basic features of Handel-C and FPGAs. Once the program has been successfully ported to Handel-C, and proved to be functionally correct, the performance of the design will need to be considered. It is highly unlikely that, for any non-trivial program, a straight port to Handel-C will result in a design with the required performance. The biggest contributor to this will probably be the effect of the deep combinatorial logic that some expressions generate. The design will need to be changed to reduce logic depth by simplifying expressions and breaking complex expressions into more steps. Further changes would probably be needed to optimise loop control structures, and possibly make use of optimised core components available from the FPGA manufacturers. An example of this is the use of the Coregen components when using the Xilinx Alliance tool set. However, to get the best performance it will usually be necessary to revise the algorithm to exploit the parallelism available with hardware. The engineer must also learn to interpret the detailed statistics available from the place and route tools, which also requires an understanding of the FPGA internal architecture, and an understanding of how Handel-C constructs hardware.

These issues are recognised by the vendors of Handel-C, and are documented in the user manuals and in a range of application notes. However, once the designer starts to optimise the design, a more detailed knowledge of the underlying hardware structure of FPGAs is needed in order to understand the optimisations.

The second cost that is incurred when using Handel-C and FPGAs is the time needed by Handel-C to compile the source language into a netlist. Compiling the hardware GP system for the artificial ant problem using 32 parallel fitness evaluations required approximately 30 minutes. This figure was obtained with full optimisation enabled, and can be reduced by disabling most of the optimisations at the expense of a slower and larger design. A related cost is the time needed to synthesise the netlist into an FPGA configuration file. The artificial ant problem required about 4 hours. The time can be reduced if the timing constraints are relaxed, but this results in a design that can be clocked only at a much lower rate. During the development of the GP system use was made of reducing the compile times by disabling optimisations and reducing the clock rate so that the development time was reduced. The optimisations were enabled and stricter timing constraints were only used when the design was functionally correct.

In summary, the cost of using Handel-C can be broken down into the cost of training, the time needed to become familiar with new techniques and the additional time required to iteratively improve the design.

## 7.3 Genetic Programming Costs

The previous sections give some general observations about the costs of using Handel-C and FPGAs. This section discusses some of the implications of implementing GP in hardware. Chapter 2 suggested that running the GP algorithm was part of a process called Meta-GP, and that implementing GP in hardware could help speed up the process of discovering good parameter settings. However, when it takes several hours to recompile and synthesise an FPGA based GP system, the benefits of a faster GP system are not so obvious. The reason the synthesis takes so long is due to the algorithms that the tools use internally. This is not something the user has any great control over, other than by relaxing the timing constraints and accepting a slower design.

The benefits of a fast GP system are also eroded because the design of the GP machine, as described in Chapters 4 and 5, is statically configured. For example, no provision was made to allow the population size, number of generations, operator proportions, or other run time parameters to be modified without having to recompile the design. This means that every time a GP run-time parameter is changed, the design must be recompiled and synthesised. One improvement that could be made is to allow some of the run-time parameters to be configured before the GP run. For example, the run-time parameter values could be written to SRAM by the host and then read by the FPGA, in the same way that the random number seed is initialised.

Another aspect of the static design is that both the fitness evaluation and the function set are hard coded into the design. This leads to the idea of dynamically configuring the function set and the fitness evaluation, possibly leaving the selection and breeding components as statically configured parts of the design. Reconfigurability was discussed in Chapter 3. Approaches to a reconfigurable hardware GP system using Xilinx Virtex FPGAs include the use of JBits [Sundararajan 00] or a configurable virtual machine [Sekanina 00].

## 7.4 Quantifying the Costs

To help a practitioner decide whether a hardware implementation would be cost effective, it would be useful to be able to quantitatively assess the costs. This section provides a quantitative method of assessing the costs of using hardware GP. The costs are then compared with the costs of using a software GP system. The costs are approximate values in Euros, in 2002. All times are in hours. The times assigned to the hardware development are based on the times needed for carrying out the experiments in the previous chapters.

The cost is expressed as a cost per GP run, $C_{\text{run}}$. The total cost of running a GP experiment is the total capital cost of equipment $C_{\text{cap}}$ plus the cost associated with developing the GP application, $C_{\text{dev}}$, plus a cost for the time to run the experiment, $C_{\text{gp}}$. Because it is likely that the equipment and development software would be used for more than one experiment, the total hardware and software costs are divided equally between $E$ GP experiments. If $C_{\text{hw}}$ is the cost of hardware and $C_{\text{sw}}$ is the cost of software, then the cost of the equipment and software for one experiment is given by:

$$C_{\text{cap}} = \frac{C_{\text{hw}} + C_{\text{sw}}}{E} \tag{7.1}$$

From this it can be seen that the more experiments the capital cost is spread over, the smaller the value of the per-experiment capital cost $C_{\text{cap.}}$ An approximation of the cost of developing and running of a GP program is made by assuming that the a users time has a uniform cost of $C_{\text{ut}}$ per hour. The cost of developing the GP system, $C_{\text{dev}}$, in time $T_{\text{dev}}$ is given by:

$$C_{\text{dev}} = C_{\text{ut}} \times T_{\text{dev}} \tag{7.2}$$

The cost of making a single GP run, $C_{\text{gp}}$, in time $T_{\text{run}}$ is given by:

$$C_{\text{gp}} = C_{\text{ut}} \times T_{\text{run}} \tag{7.3}$$

The total cost, $C_{\text{tot}}$, to implement a GP system and perform one run is given by:

$$C_{\text{tot}} = C_{\text{cap}} + C_{\text{dev}} + C_{\text{gp}} \tag{7.4}$$

Running the same GP system for $N$ runs, the cost per run is given by:

$$C_{\text{run}} = \frac{C_{\text{cap}} + C_{\text{dev}} + (N \times C_{\text{gp}})}{N} \tag{7.5}$$

From this it is clear that the greater the number of runs performed, the more cost effective the GP implementation will be. To see the effect of a different number of runs, two problems are used as examples. The first is the Boolean even-6-parity problem, described in Section 5.3.3. The second is a hypothetical problem that takes 24 hours to run when implemented in software. Table 7.1 shows the costs used for the examples. The time needed to implement the hardware GP system is taken as 100 and the time to implement the software GP system 50. These times assume that each GP system was developed independently. The longer time needed for the hardware system is due to the longer time needed for each program iteration, as discussed in Section 7.2. $C_{\text{ut}}$ is assumed to be constant for both implementation and running, and is taken as €10/hour.

**Table 7.1:** Costs for implementing hardware GP and software GP. All costs are approximate and are in Euros as of 2002.

| | Hardware GP | Software GP |
|---|---:|---:|
| Hardware Costs $C_{hw}$ | €10,000 | €2,000 |
| Software Costs $C_{sw}$ | €10,000 | €100 |
| Implementation cost $C_{impl}$ | €2,000 | €1,000 |

The Boolean even-6-parity problem using 64 parallel fitness evaluations was clocked at 34 MHz, and required $40 \times 10^6$ cycles. Therefore a single run took $40 \times 10^6 / 34 \times 10^6$ s = 1.175 s or 0.00034 h. The corresponding software implementation running at 200 MHz required $419 \times 1.175$ s = 492 s or 0.136 h. Using a value of 100 for $E$, Figure 7.1 shows the cost for both the hardware and the software implementation for different values of $N$ between 1 and 1000. Note the log scale for the cost per run. The graph shows, for the Boolean even-6-parity problem, that as the number of runs increases there is a point at which the hardware implementation has a lower cost per run than the same algorithm implemented in software.

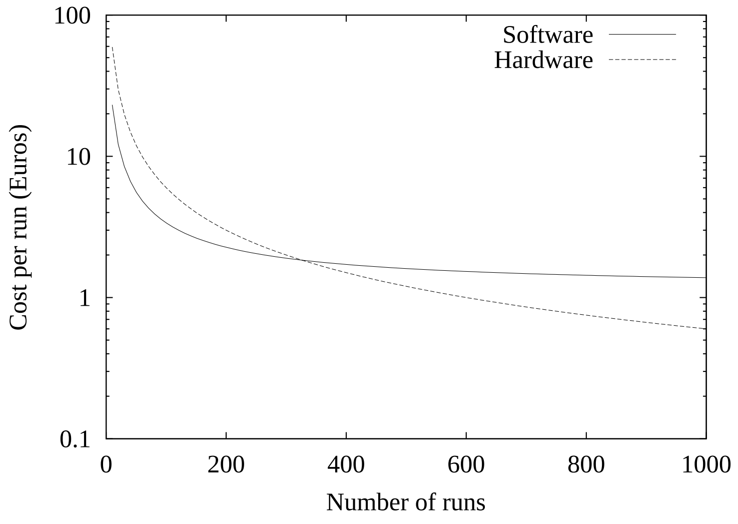


**Figure 7.1:** Cost benefit of using hardware GP for the Boolean even-6-parity problem.

The cost per run was also calculated for the hypothetical problem that requires 24 hours to complete one run when implemented in software. It is assumed that the hardware GP system can achieve a Speedup$_{\text{time}}$ of 500, so the hardware GP system would require 0.048 h. This is plotted in Figure 7.2. This shows that for experiments that require a long time to run, a hardware GP system can reduce the cost of running the experiments by an order of magnitude or more.

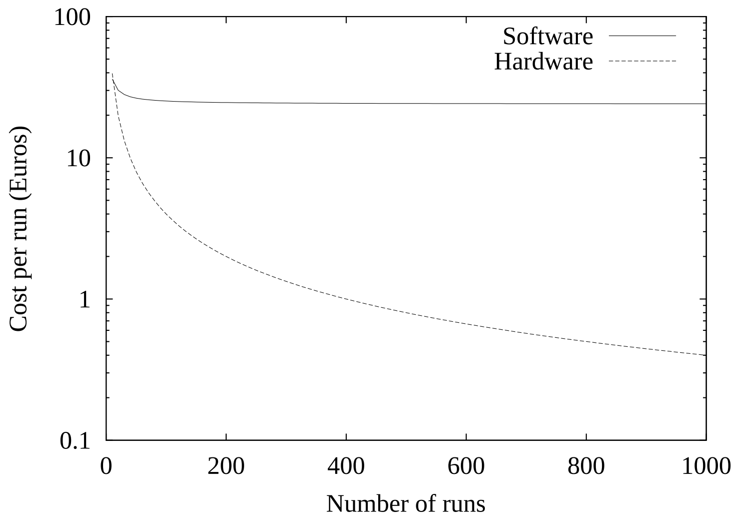


**Figure 7.2:** Cost benefit using hardware GP for a hypothetical problem that requires 24 hours to run when implemented in software.

However, the absolute cost of developing and running a GP system is often not as important as the total time available to carry out experiments. Within the total time two activities must be completed; the design and implementation of the GP system, and running the GP experiments using the process of Meta-GP as described in Section 2.6.3. In this scenario the number of runs that can be achieved in that time is often the limiting factor and maximising the number of runs is therefore an important goal.

The number of runs, $N$, that can be completed in time $T_{\text{tot}}$ is:

$$N = \frac{T_{\text{tot}} - T_{\text{dev}}}{T_{\text{run}}} \tag{7.6}$$

where $T_{\text{dev}}$ is the time to develop the GP system and $T_{\text{run}}$ is the time required to complete a single run of the GP experiment.

Using the Boolean even-6-parity problem as an example, $T_{\text{dev}}$ is 100 for the hardware implementation and 50 for the software implementation. The time for one run of the hardware implementation was 0.00034 h and for the software implementation was 0.16 h.

Figure 7.3 shows the number of runs that can be run for values of $T_{\text{tot}}$ between 1 and 150 hours. Note the log scale for the number of runs. This graph shows that where the total time is limited

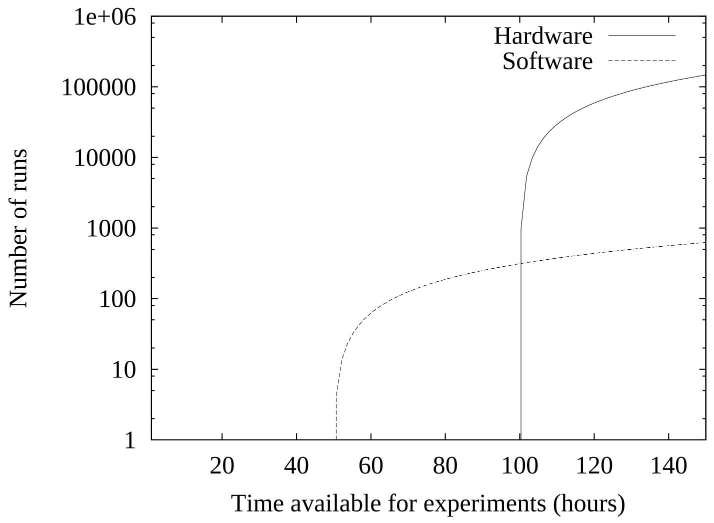


**Figure 7.3:** Number of runs that can be achieved using the Boolean even-6-parity problem for a software implementation and a hardware implementation

or the number of runs is not critical, that the software implementation may be adequate. However, where the number of runs in a given time needs to be maximised, the hardware implementation has a clear advantage for this experiment.

The number of runs for the hypothetical problem that requires 24 hours to run is shown in Figure 7.4. This graph shows that for problems with extended running times, when the total time available

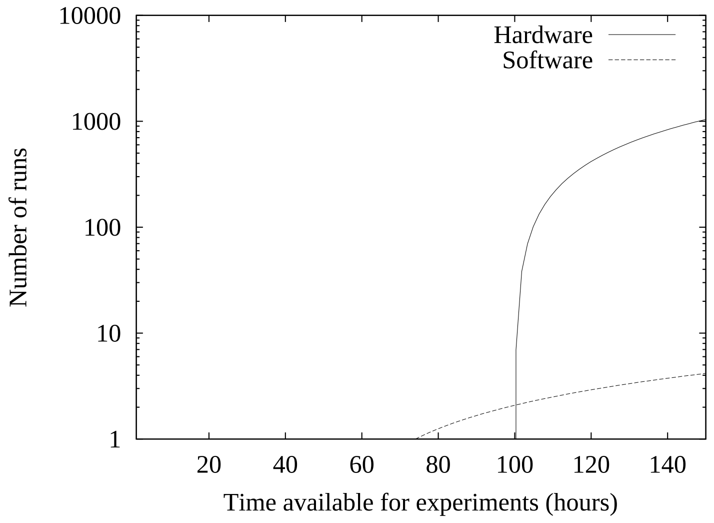


**Figure 7.4:** Number of runs that can be achieved using a hypothetical problem that requires 24 hours to run when implemented in software.

is greater than the time to implement a hardware GP system, the hardware GP system allows many more runs to be completed in the time available. The results in Figures 7.3 and 7.4 ignore the effect of having to recompile the GP system for every change made to the GP parameters. This is reasonable given that a method of achieving this has already been proposed (7.3).

## 7.5 Summary

The use of Handel-C and FPGAs by a software engineer has a number of costs associated with it. Firstly, the cost of training the engineer in the use of Handel-C and the underlying hardware. Secondly, the capital costs of equipment and software. Thirdly, the extended time needed to implement a design using Handel-C and the FPGA vendor's synthesis tools. Weighed against these costs are the potential for greatly reduced run times for a GP problem, and the potential to explore areas of GP that would be intractable using traditional software implementations.

To obtain the best advantage from using hardware GP as part of the Meta-GP process, the design should be modified to allow the dynamic modification of the run time parameters. This would avoid the time consuming process of having to recompile the design every time a parameter change was required.

A method of assessing the cost effectiveness of using hardware GP has been presented, which should help a user decide whether a hardware GP system can be justified in terms of cost. A method of predicting the number of runs that a GP implementation can complete in a given time has also been presented. Two examples have been used to illustrate these methods. The method of assessing the cost per run is sensitive to the capital cost and the value of $C_{\mathrm{ut}}$. Where the capital cost is higher or the cost of implementing the GP system is higher, then the use of hardware GP may not be justifiable on cost alone. Where the equipment is only used for a small number of experiments, it is unlikely that the user of GP would find a hardware implementation a good economic proposition. However, for problems that would be impractical to explore using traditional software implementations, a hardware GP system could be justified on the basis of cost.

The use of hardware GP can also be justified where the number of runs that can be completed in a fixed time needs to be maximised, for example when the problem requires an extended running time.

All the examples have assumed that the GP system is implemented completely for each experiment. In practice, once a GP system has been designed, the time needed to develop a variant will be much smaller, so the advantages of using hardware GP will be more pronounced.

# Chapter 8

# Conclusions

The Genetic Programming algorithm requires a lot of computing resources because it often has to evaluate hundreds, thousands or millions of programs during a run. Furthermore, the algorithm often needs to be run multiple times to find an acceptable solution to a problem. It is also the case that numerous runs of the GP algorithm are needed when empirically investigating the behaviour of a GP system.

The hypothesis of this research is that implementing GP in hardware will reduce the time needed to run the GP algorithm. A reduced run time for GP would allow the detailed operation of GP to be explored in ways that previously would have required an uneconomic investment in time and equipment. Reducing the running time of GP would also allow GP to be applied to problems that, so far, have been hard or impossible for GP to solve.

## 8.1 Recapitulation of the Contributions

### 8.1.1 Hardware Genetic Programming

This thesis approached the idea of implementing GP in hardware from a practical standpoint by combining GP with two technologies – a *Field Programmable Gate Array* (FPGA) and Handel-C – to produce the first published complete GP system in hardware. The implementation of a proof of concept design, described in Chapter 4, shows that implementing all the stages of GP in hardware is a practical proposition, and that using Handel-C it was possible to design the hardware using an algorithmic approach. By using a simplified design, and accepting that there would be limitations in such a design, practical issues surrounding the use of Handel-C and designing for an FPGA were addressed. However, the proof of concept design used an essentially serial GP algorithm, as used in software implementations of GP. The results from the problems used to test the design (simple regression and the XOR problem) showed that ignoring the opportunities to fully exploit the parallelism offered by hardware meant that throughput was limited and that the potential speedups were not achieved. The proof of concept design also showed that, without external memory, an FPGA can only support a very limited population size.

The results from the optimised implementation (Chapter 5), which used pipelines, confirmed that exploiting the parallelism in hardware could lead to substantial increases in throughput. The results from the optimised implementation also showed that, by using external memory, it is feasible to implement some common GP benchmark problems entirely in hardware, and, for one of the problems used (Boolean even-6-parity problem), speed up the execution of GP by a factor of over 400 when compared to the same algorithm executed in software.

### 8.1.2 Meta-GP

To support the hypothesis, this research also considered the question of why it is important to reduce the run time of GP. Using GP is almost always presented as a linear series of steps. In reality, most users know this is not an accurate reflection and that using GP often requires extensive experimentation to produce acceptable results. In this thesis the process of using the GP algorithm has been presented explicitly as an algorithm – called Meta-GP. Furthermore, the term Meta-GP has been used to describe the broad activity of using an evolutionary approach to discovering acceptable operational parameters for the GP algorithm, rather than the narrow interpretation often used to describe the automatic optimisation of a single parameter. By recognising that Meta-GP is an essential part of developing and using GP systems, it is shown that the time to execute the core GP algorithm is more important than is normally assumed. Reducing the time for the core algorithm means that more GP runs can be completed, given a fixed time budget.

A practical demonstration of the benefits of using a hardware GP implementation in the Meta-GP process was highlighted during the investigation into the behaviour in Chapter 6. The speedup offered by the hardware implementation meant that it was feasible to repeatedly run the experiments for 500 runs in order to get an average of many runs, while still being able to experiment with parameter settings and modify the crossover operators. Using a software implementation would have required a much longer running time, nearly 12 hours for the Boolean even-6-parity problem, which would have hindered experimentation with parameter settings and exploration of different operators.

### 8.1.3 Exploring GP Behaviour

The standard method of describing the behaviour of GP systems in the published literature is to plot a graph of average fitness versus generation, sometimes together with a plot of the effort required to find a solution, as originally described in [Koza 92]. While this method shows how quickly a GP system reaches an average fitness level, it does not show how the GP system is behaving internally. Furthermore, as shown in Chapter 6, the standard method does not clearly show the differences between two similar GP systems. In this thesis, the number of correct programs at the end of a run has been used, which shows more directly the differences in behaviour between experiments. Although the number of correct programs has previously been used as a measure of effort, for example by Miller and Thomson in [Miller 00], this thesis also considered the program size distribution. The results were presented as a histogram of the lengths of the 100% correct programs. This gave another insight into the choice of operational parameters. The program size distribution for the artificial ant problem suggested modifying the maximum length that programs were allowed to grow to. The modifications showed that the number of 100% correct solutions could be increased.

When investigating the behaviour of different crossover operators, the changes in the population length distribution during a GP run gives an insight into the pressures that different operators can have on the program lengths of the population. Using this insight, a new crossover operator – called single child limiting crossover – has been developed. The single child limiting crossover operator has been shown to be effective in constraining the maximum length of programs and in controlling the tendency of GP programs to bloat.

### 8.1.4 Taxonomy of GP Attributes

Arranging the attributes of GP into a taxonomy was prompted because there has not been a unified classification of this information before. The taxonomy makes it explicit which attributes are related to the problem and which attributes are GP specific. Just as in software engineering, where there is a clear distinction between the specification of a problem and the implementation, explicitly identifying which attributes are problem specific and which are GP specific should help in identifying suitable attribute values for problems. The survey of problems highlighted several features of how GP has been applied. Firstly, the lack of formal methods to specify problems runs counter to the general adoption of such methods in general software engineering. Secondly, the ranges of population sizes, generations run and resultant program sizes all suggest that GP has only tackled problems with modest dimensions. Lastly, it showed that software implementations of GP dominate the published results. These three features all suggest areas of further research, indeed, the lack of hardware GP systems was one motivation for the research described in this thesis.

It is not claimed that the taxonomy presented in Chapter 2 is complete but it is given as a starting point for further refinement.

### 8.1.5 High Level Language Hardware Compilation

The principle benefit of using Handel-C as an implementation language was that the implementation could be expressed as an algorithm, rather than a structural description of the hardware. A second benefit was that the same algorithm could be implemented in both hardware and software with minimal changes. Being able to re-use the same algorithm for a software implementation and for an FPGA was found to be important for three reasons. Firstly, it facilitated the initial development of the algorithm in software. This is advantageous because using Handel-C meant the hardware development process required a longer time to iterate through the design-code-test cycle than the software cycle. Secondly, because the code can be made portable between Handel-C and ISO-C, the porting of the algorithm from the software development environment to Handel-C, and vice versa, is simplified, reducing the opportunities for errors to be introduced in the porting process. Thirdly, it allowed the direct comparison of the algorithm's performance when implemented in software and hardware. In the published literature of evolutionary computation, most of the previously reported comparisons between hardware and software have either used a software simulation of the hardware design, or have only used very loosely related software algorithm equivalents of the hardware.

### 8.1.6 Random Number Generation for Hardware

There is a popular theme in the literature of evolutionary computation which implies that a very good source of random numbers is essential. Previous studies have shown that, given a reasonably good generator, the quality of a random number generator is probably not as important as some have stated. The analysis of the hardware random number generators confirmed that for most hardware generators the quality of the generator has little measurable effect on the performance of the GP algorithm. However, it also showed that a demonstrably poor generator, such as a simple linear feedback shift register generator, can impair the performance of the GP algorithm.

In addition to using pseudo random number generators and true random number generators, the analysis also showed that using a non-random source of numbers allowed the GP algorithm to find solutions, though with a much lower probability of success than the pseudo random and truly random generators.

### 8.1.7 Economics of Hardware Genetic Programming

This thesis started with the hypothesis that implementing GP in hardware could result in reduced running times for the GP algorithm. The experimental evidence shows that it is indeed practical to implement GP in hardware and for the implementation to be able to solve some test problems in a much shorter time than an equivalent software implementation. However, it is unlikely that a hardware GP system would be appropriate for all problems and for all classes of user. Because the initial investment in hardware and software is significant, using hardware GP may not be cost-effective for some problems. By considering both the time to run an experiment and the potential for running many more experiments in a given time, a method of quantifying the cost benefit of using hardware GP has been presented, which allows a practitioner to objectively evaluate the usefulness of hardware GP.

## 8.2 Further Work

### 8.2.1 Alternative FPGAs

#### Higher Performance FPGAs

The design in Chapters 4 and 5 used the Xilinx XCV2000e device, which was one of the most advanced devices available at the time the research was carried out. However, technological advances have taken place and the more recent Virtex-II FPGAs from Xilinx are bigger and faster than the Virtex-E devices and promise even better speedups. Initial studies using these devices have indicated that a further speedup of two to three times is possible. This speedup is achieved by a higher clock rate, up to 420 MHz, and better routing resource utilisation. The Virtex-II devices also have larger gate counts and bigger on-chip BRAMs. The largest of these devices currently available (mid-2002), – the XC2V8000 – has 46 592 slices. Early work suggests that this device could accommodate up to 128 parallel fitness evaluations for the artificial ant problem, and 256 for the Boolean even-6-parity problem. These devices also have on-chip multipliers which would make it possible to use function sets that require multiplication and division operators, often required for regression problems. On-chip multipliers would also mean that other PRNGs could be used, such as the Mother PRNG described in Section 6.4. The initial indication is that by combining the faster clock frequency and more parallel fitness evaluations the ant problem could see a further 10 times speedup, and the Boolean even-6-parity problem a speedup of 20 times. Further work needs to be carried out to verify this.

#### Low Cost FPGAs

At the other end of the device spectrum, there are some low cost FPGAs that may be suitable for embedding into equipment. For example, the Spartan range from Xilinx which is fitted to the low cost RC100 development board, available from Celoxica. This board has a Spartan XC2S200 FPGA which has 5 292 logic cells. Initial experiments using this device indicate that it could accommodate the hardware GP system.

**Alternative FPGA Manufacturers**

While this work has used FPGAs from Xilinx, FPGAs are available from a number of other manufacturers. To date, none of these has been considered for implementing GP. However, one of the features of Handel-C is that it is not tied to one device manufacturer. This allows a single design to be targeted at many different FPGA families, often with little or no change to the design. For particular applications, other manufacturers devices may offer advantages such as gate count, specialised hardware features, speed, power consumption or cost.

### 8.2.2 Generalised Design

As already noted in Chapter 7, the cost in time to make small changes to a GP system in an FPGA can be large, so it would seem reasonable to add the means of varying some of the run-time parameters for a given design. This could be done by supplying them during start-up. For example, a design with a fixed maximum population size could have a run-time limit below the maximum specified by the host program. This would make the technique more useful for exploring particular aspects of GP behavior.

Two possible approaches to realising a more general design have been suggested in Chapter 7: JBits [Sundararajan 00] and a configurable virtual gate array [Sekanina 00]. Both these approaches use the idea of being able to reconfigure the FPGA so that the logic can be altered externally. Another alternative, mentioned in Chapter 4, is to implement a general set of functions, modelled, for example, on the Java virtual machine. Implementing a virtual machine in hardware would then allow the function set and the fitness functions to be compiled externally and then made available to the FPGA for execution, without needing to alter the FPGA contents. The fitness function could be placed in SRAM and executed directly by the FPGA hosted virtual machine. However, using this approach is likely to reduce the speedups that are obtained when an optimised and problem specific function set is implemented in the FPGA.

### 8.2.3 Techniques from Standard Genetic Programming

Several specialised techniques have been used to facilitate the implementation of GP in hardware. In particular, the use of a linear representation and the crossover operators described in Chapter 6.

The crossover operators investigated in Chapter 6 used a linear representation, but from the results in [Poli 01a] similar behavior would be expected when the truncating crossover and the single child limiting crossover techniques are applied to standard tree based GP. This is an area that should be investigated.

Other techniques have been suggested for controlling the program size during evolution such as homologous and size fair operators [Langdon 99] and smooth operators [Page 99], which could also be adapted to a hardware implementation.

The linear representation was used because of its simplicity and regularity. Other regular representations that should be straightforward to implement in hardware include Cartesian Genetic Programming by Miller and Thompson [Miller 00].

### 8.2.4 Applications

In this thesis, hardware GP has been used to implement only a limited number of problems. To get a more complete picture of the effects of the design decisions, and the limitations of the design, more problems would need to be implemented and analysed.

None of the applications described in this thesis, or suggested as future applications, require large fitness functions. Although the initial population creation, breeding and support functions used around 20% of the chip area, the amount of logic that can be accommodated on an FPGA is severely limited. It is therefore not clear how fitness functions that require a lot of computing resources will be able to be accommodated on an FPGA. For example, Bennett [Bennett III 96] [Bennett III 99], Koza *et al.* [Koza 96b] [Koza 00a] [Koza 97c] [Koza 99a] and others have used one or more Spice circuit simulators to evolve electrical circuits. It is doubtful whether Spice could be ported to an FPGA using current technologies, so there is going to be a class of problems that would be hard to implement completely in hardware.

None of the applications implemented in this thesis have used floating point mathematical functions. Although Celoxica provide a fixed point and floating point library with Handel-C, no work has yet been done to evaluate how efficiently fitness functions that use fixed or floating point arithmetic can be implemented.

Although hardware GP has been shown to give useful speedups for the problems already implemented, these are all problems that have already been solved by standard GP. However, hardware

GP offers the opportunity to tackle problems that have not yet been considered practical. Because the FPGA system has the potential to evaluate individuals in a far shorter time than even the fastest Pentium class computers, it becomes possible for a GP system to process fast real-time data. This would be particularly appropriate where fitness data are available only as a real-time data stream, for example in signal processing applications, where real-time data, such as telemetry from remote sensors, could be processed directly by the FPGA. For example, this might find use in applications that need to continuously optimise a signal processing algorithm, such as a filter, in response to changing conditions, or where real-time constraints must be met. An example of such an applications is scheduling hard disc operations as investigated by Turton and Arslan [Turton 95].

FPGAs are typically endowed with a large number of *Input/Output* (I/O) pins. Using these I/O pins, a fitness function implemented in an FPGA would have direct access to external signals. These pins could, for example, be connected directly to an image sensor device, giving the fitness functions direct access to raw image data. This would find applications in image recognition problems.

Another interesting possibility for using an FPGA is that the output signals can be directly encoded into the function set, thereby opening up the possibility of embedding the GP system and having it directly control hardware devices while evaluating the fitness of the programs. An example of this would be a robotic control that reads sensor inputs directly using some of the I/O pins on the FPGA. The evolving programs would then generate control signals directly to the robot, again using some of the I/O pins. Although FPGAs have previously been embedded in robots, for example in [Thompson 97], [Seok 00] and [Haddow 99] using hardware GP would mean that the robot's behaviour could be evolved using GP without the need for an external PC.

The applications suggested in this section have only hinted at the possibilities made possible by implementing GP in an FPGA. It is hoped that the fast execution of the hardware GP algorithm coupled with the compact hardware requirements of an FPGA would suggest other applications of this work.

## 8.3 Conclusions About the Hypothesis

The research has supported the hypothesis that implementing GP in hardware can speedup the operation of the GP algorithm. It has also been shown that a hardware GP system reduces the time needed to investigate the detailed operation of GP. However, the analysis of the economic factors that influence the choice of whether to use a hardware or software GP system shows that, although a large speedup is possible, a hardware GP system would not be appropriate in every application due to the high initial capital cost and the time required to develop the GP system. Finally, some of the potential applications suggest that hardware GP could open up areas of research which would otherwise be impractical for software based GP.

# Bibliography

[Adorni 98] Giovanni Adorni, Federico Bergenti & Stefano Cagnoni. *A cellular-programming approach to pattern classification.* In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer & Terence C. Fogarty, (Eds.), Proceedings of the First European Workshop on Genetic Programming, Vol. 1391 of *LNCS*, pp 142–150, Paris, France, 14-15 April 1998. Springer-Verlag.

[Adorni 99] Giovanni Adorni, Stefano Cagnoni & Monica Mordonini. *Genetic programming of a goal-keeper control strategy for the robocup middle size competition.* In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming; Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 109–119, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Ahluwalia 98] M. Ahluwalia & L. Bull. *Co-evolving functions in genetic programming: dynamic ADF creation using GLiB.* In William V. Porto, N. Saravanan, D. Waagen & A. E. Eiben, (Eds.), Evolutionary Programming VII: Proceedings of the 7th Annual Conference on Evolutionary Programming, Vol. 1447 of *LNCS*, pp 809–818, Mission Valley Marriott, San Diego, CA, USA, 25-27 March 1998. Springer-Verlag.

[Aiyarak 97] P. Aiyarak, A. S. Saket & M. C. Sinclair. *Genetic programming approaches for minimum cost topology optimisation of optical telecommunication networks.* In Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA, University of Strathclyde, Glasgow, United Kingdom, 1-4 September 1997. IEE.

[Alfke 00] Peter Alfke. *Choices, choices, and options. How to choose the best Xilinx programmable logic technology for your application.* The Quarterly Journal for Xilinx Programmable Logic Users, vol. 37, November 2000. `http://www.xilinx.com/xcell/xcell37.htm`.

[Altera 01] Altera. *Linear feedback shift register megafunction.* `http://www.altera.com/literature/sb/sb11_01.pdf`, December 2001.

[Amoroso 94] Edward Amoroso. *Fundamentals of Computer Security Technology.* Prentice-Hall, Upper Saddle, NJ, USA, 1994.

[Andersson 99] Bjorn Andersson, Per Svensson, Peter Nordin & Mats Nordahl. *Reactive and memory-based genetic programming for robot control.* In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 161–172, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Andre 94] David Andre. *Learning and upgrading rules for an OCR system using genetic programming*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Andre 96] David Andre & John R. Koza. *Parallel genetic programming: A scalable implementation using the transputer network architecture*. In Peter J. Angeline & Kenneth E. Kinnear Jr., (Eds.), Advances in Genetic Programming 2, Chap. 16, pp 317–338. MIT Press, Cambridge, MA, USA, 1996.

[Andrews 94] Martin Andrews & Richard Prager. *Genetic programming for the acquisition of double auction market strategies*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 16, pp 355–368. MIT Press, Cambridge, MA, USA, 1994.

[Angeline 97] Peter J. Angeline. *Subtree crossover: building block engine or macromutation?* In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba & Rick L. Riolo, (Eds.), Genetic programming 1997: proceedings of the 2$^{nd}$ annual conference, pp 9–17, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Angeline 98] Peter J. Angeline. *A historical perspective on the evolution of executable structures*. Fundamenta Informaticae, vol. 36, no. 1–4, pp 179–195, August 1998.

[Aporntewan 01] Chatchawit Aporntewan & Prabhas Chongstitvatana. *A hardware implementation of the compact genetic algorithm*. In Proceedings of IEEE Congress on Evolutionary Computation, pp 624–629, Seoul, Korea, 27-30 May 2001. IEEE Press.

[A|RT 02] A|RT. *A|RT Designer Pro*. `http://www.frontierd.com`, 2002. Vendors of the A|RT designer pro C to hardware synthesis tools.

[Atkin 94] Marc S. Atkin & Paul R. Cohen. *Learning monitoring strategies: A difficult genetic programming application*. In Proceedings of the 1994 IEEE world congress on computational intelligence, pp 328–332, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Baker 87] J. E. Baker. *Reducing bias and inefficiency in the selection algorithm*. In J. J. Grefenstette, (Ed.), Proceedings of an International Conference on Genetic Algorithms and their Applications, pp 14–21. Lawrence Erlbaum Associates, 1987.

[Banzhaf 93] Wolfgang Banzhaf. *Genetic programming for pedestrians*. In Stephanie Forrest, (Ed.), Proceedings of the 5$^{th}$ International Conference on Genetic Algorithms, ICGA-93, p 628, University of Illinois at Urbana-Champaign, USA, 17-21 July 1993. Morgan Kaufmann.

[Banzhaf 96] Wolfgang Banzhaf, Frank D. Francone & Peter Nordin. *The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets*. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg & Hans-Paul Schwefel, (Eds.), Parallel Problem Solving from

Nature IV, Proceedings of the International Conference on Evolutionary Computation, Vol. 1141 of *LNCS*, pp 300–309. Springer-Verlag, 22-26 September 1996.

[Banzhaf 97] Wolfgang Banzhaf, Peter Nordin & Markus Olmer. *Generating adaptive behavior for a real robot using function regression within genetic programming*. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba & Rick L. Riolo, (Eds.), Genetic Programming 1997: Proceedings of the 2$^{nd}$ Annual Conference, pp 35–43, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Banzhaf 98] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller & Frank D. Francone. *Genetic programming – an Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, Verlag fur digitale Technologie GmbH, Heidelberg., January 1998.

[BBN Technologies 02] BBN Technologies. *EvolvaWare*. `http://vishnu.bbn.com/papers/evolvaware/`, 2002. Web site for EvolvaWare with an online demonstration of evolving a sorting network.

[Bennett III 96] Forrest H. Bennett III, John R. Koza, David Andre & Martin A. Keane. *Evolution of a 60 decibel op amp using genetic programming*. In Tetsuya Higuchi, Iwata Masaya & Weixin Liu, (Eds.), Proceedings of international conference on evolvable systems: from biology to hardware (ices-96), Vol. 1259 of *Lecture Notes in Computer Science*, Tsukuba, Japan, 7-8 October 1996. Springer-Verlag.

[Bennett III 99] Forrest H. Bennett III, Martin A. Keane, David Andre & John R. Koza. *Automatic synthesis of the topology and sizing for analog electrical circuits using genetic programming*. In Kaisa Miettinen, Marko M. Mäkelä, Pekka Neittaanmäki & Jacques Periaux, (Eds.), Evolutionary algorithms in engineering and computer science, pp 199–229, Jyväskylä, Finland, 30 May - 3 June 1999. John Wiley & Sons.

[Bhattacharya 01] Maumita Bhattacharya, Ajith Abraham & Baikunth Nath. *A linear genetic programming approach for modeling electricity demand prediction in victoria*. In Ajith Abraham & Mario Koppen, (Eds.), Proceedings 2001 international Workshop on Hybrid Intelligent Systems, pp 379–394, Adelaide, Australia, 11-12 December 2001. Springer-Verlag.

[Bhattacharyya 02] Siddhartha Bhattacharyya, Olivier V. Pictet & Gilles Zumbach. *Knowledge intensive genetic discovery in foreign exchange markets*. IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp 169–181, April 2002.

[Blickle 94] Tobias Blickle & Lothar Thiele. *Genetic programming and redundancy*. In J. Hopf, (Ed.), Genetic algorithms within the framework of evolutionary computation (workshop at ki-94, saarbrücken), pp 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik (MPI-I-94-241).

[Blickle 95a] Tobias Blickle. *YAGPLIC user manual*. Technical report, Computer Engineering and Communication Network Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, CH-8092, Zurich, 1995.

[Blickle 95b] Tobias Blickle & Lothar Thiele. *A Comparison of Selection Schemes Used in Genetic Algorithms*. TIK-Report 11, TIK Institut fur Technische Informatik und Kommunikationsnetze, Computer Engineering and Networks Laboratory, ETH, Swiss Federal Institute of Technology, Gloriastrasse 35, 8092 Zurich, Switzerland, December 1995. revision 2.

[Boehm 81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[Bowen 02] Jonathan Bowen. *The Z notation*. `http://www.afm.sbu.ac.uk/z/`, 2002.

[Brameier 01] Markus Brameier & Wolfgang Banzhaf. *A comparison of linear genetic programming and neural networks in medical data mining*. IEEE Transactions on Evolutionary Computation, vol. 5, no. 1, pp 17–26, February 2001.

[Bright 00] Marc Bright & Brian C.H. Turton. *An efficient random number generator architecture for hardware parallel genetic algorithms*. In Marc Schoenauer, Kalyanmoy Deb, Guenter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo & Hans-Paul Schwefel, (Eds.), Parallel Problem Solving from Nature - PPSN VI, Vol. 1917 of *LNCS*, 16 September 2000.

[Callahan 98] Timothy J. Callahan & John Wawryzynek. *Instruction-level parallelism for reconfigurable computing*. In Reiner W. Hartenstein & Andres Keevallik, (Eds.), Field-Programmable Logic: From FPGAs to Computing Paradigm, pp 248–257. Springer-Verlag, September 1998.

[Campo Novales 02] Andrés del Campo Novales. *Genetic programming studio*. `http://www.uco.es/i52canoa/`, 2002.

[Cantu-Paz 98] Eric Cantu-Paz. *A Survey of Parallel Genetic Algorithms*. Calculateurs Parallels, Reseaux et Systems Repartis, vol. 10, no. 2, pp 141–171, 1998.

[Cantú-Paz 02] Erick Cantú-Paz. *On random numbers and the performance of genetic algorithms*. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke & N. Jonoska, (Eds.), GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp 311–318, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[Celoxica Ltd 01a] Celoxica Ltd. *Handel-C language reference manual, revision 2.1*. Celoxica Ltd., 20 Park Gate, Milton Park, Abingdon, Oxfordshire, OX14 4SH, United Kingdom., 2001. Vendors of Handel-C.

[Celoxica Ltd 01b] Celoxica Ltd. *RC1000 Funcational Reference Manual*. 20 Park Gate, Milton Park, Abingdon, Oxfordshire, OX14 4SH, United Kingdom, 2001. Part number RM-1140-0, revision 1.3.

[Celoxica Ltd 01c] Celoxica Ltd. *RC1000 Hardware Reference Manual*. 20 Park Gate, Milton Park, Abingdon, Oxfordshire, OX14 4SH, United Kingdom, 2001. Part number RM-1120-0, revision 2.3.

[Celoxica Ltd 01d] Celoxica Ltd. *RC1000 Software Reference Manual*. 20 Park Gate, Milton Park, Abingdon, Oxfordshire, OX14 4SH, United Kingdom, 2001. Part number RM-1130-0, revision 1.3.

[Celoxica Ltd 01e] Celoxica Ltd. *Web site of Celoxica Ltd.* `www.celoxica.com`, 2001. Vendors of Handel-C. Last visited 15/June/2001.

[Celoxica 02] Celoxica. *Academic papers*. `http://www.celoxica.com/programs/university/academic_papers.htm`, 2002.

[Chen 98] S-H. Chen & C-H. Yeh. *Genetic programming in the overlapping generations model: an ilustration with the dynamics of the inflation rate*. In V. William Porto, N. Saravanan, D. Waagen & A. E. Eiben, (Eds.), Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming, Vol. 1447 of *LNCS*, pp 829–837, Mission Valley Marriott, San Diego, CA, USA, 25-27 March 1998. Springer-Verlag.

[Choi 00] Yun-Ho Choi & Duck Jin Chung. *VLSI Processor of parallel genetic algorithm*. In Proceedings of the 2nd IEEE ASIA-Pacific Conference on ASICs, Cheju Island, Korea, August 2000. IEEE.

[Chong 99] Fuey Sian Chong & William B. Langdon. *Java based distributed genetic programming on the internet*. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela & Robert E. Smith, (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, Vol. 2, p 1229, Orlando, FL, USA, 13-17 July 1999. Morgan Kaufmann.

[Christensen 02] Steffan Christensen & Franz Oppacher. *An analysis of Koza's computational effort statistic for genetic programming*. In James. A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan & Andrea G. B. Tettamanzi, (Eds.), Proceedings of the 5th European Conference, EuroGP'2002, Vol. 2278 of *LNCS*, pp 182–191, Kinsale, Ireland, April 2002. EvoGP, Springer-Verlag.

[Chu 99] Pong P. Chu & Robert E. Jones. *Design techniques of FPGA based random numbers*. In Proceedings. of Military and Aerospace Applications of Programmable Devices and Technology Conference, MAPLD'99, Kossiakoff Conference Centerm The John Hopkins University - Applied Physics Laboratory, September 1999. RS Information Systems, Greenbelt, MD, USA.

[Clack 97a] Chris Clack, Jonny Farringdon, Peter Lidwell & Tina Yu. *Autonomous document classification for business*. In W. Lewis Johnson, (Ed.), The First International Conference on Autonomous Agents (agents '97), pp 201–208, Marina del Rey, CA, USA, February 5-8 1997. ACM Press.

[Clack 97b] Chris Clack & Tina Yu. *Performance enhanced genetic programming*. In Peter J. Angeline, Robert G. Reynolds, John R. McDonnell & Russ Eberhart, (Eds.), Proceedings of the Sixth Conference on Evolutionary

Programming, Vol. 1213 of *LNCS*, Indianapolis, Indiana, USA, 1997. Springer-Verlag.

[Clark 81] Randy Clark. *UCSD p-System and UCSD Pascal users' manual*. SoftTech Microsystems, San Diego, 2nd edition, 1981.

[Clark 96] David A. Clark. *Supporting FPGA microprocessors through retargetable software tools*. Master's thesis, Department of Electrical and Computer Engineering, Brigham Young University, April 1996.

[Conrads 98] Markus Conrads, Peter Nordin & Wolfgang Banzhaf. *Speech sound discrimination with genetic programming*. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer & Terence C. Fogarty, (Eds.), Proceedings of the First European Workshop on Genetic Programming, Vol. 1391 of *LNCS*, pp 113–129, Paris, France, 14-15 April 1998. Springer-Verlag.

[Cramer 85] Nichael Lynn Cramer. *A representation for the adaptive generation of simple sequential programs*. In John J. Grefenstette, (Ed.), Proceedings of an International Conference on Genetic Algorithms and their Applications, pp 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[CynApps 02] CynApps. *CynApps. Cynthesis applications for higher level design*. `http://www.forteds.com`, 2002.

[Daida 96] Jason M. Daida, Jonathan D. Hommes, Tommaso F. Bersano-Begey, Steven J. Ross & John F. Vesecky. *Algorithm discovery using the genetic programming paradigm: extracting low-contrast curvilinear features from SAR images of arctic ice*. In Peter J. Angeline & K. E. Kinnear, Jr., (Eds.), Advances in Genetic Programming 2, Chap. 21, pp 417–442. MIT Press, Cambridge, MA, USA, 1996.

[Daida 97] Jason Daida, Steven Ross, Jeffrey McClain, Derrick Ampy & Michael Holczer. *Challenges with verification, repeatability, and meaningful comparisons in genetic programming*. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba & Rick L. Riolo, (Eds.), Genetic Programming 1997: Proceedings of the Second Annual Conference, pp 64–69, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Daida 99] Jason M. Daida, Derrick S. Ampy, Michael Ratanasavetavadhana, Hsiaolei Li & Omar A. Chaudhri. *Challenges with verification, repeatability, and meaningful comparison in genetic programming: Gibson's magic*. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela & Robert E. Smith, (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, Vol. 2, pp 1851–1858, Orlando, FL, USA, 13-17 July 1999. Morgan Kaufmann.

[Das 94] Sumit Das, Terry Franguidakis, Michael Papka, Thomas A. DeFanti & Daniel J. Sandin. *A genetic programming application in virtual reality*. In Proceedings of the First IEEE Conference on Evolutionary Computation, Vol. 1, pp 480–484, Orlando, FL, USA, 27-29 June 1994. IEEE Press. Part of 1994 IEEE World Congress on Computational Intelligence, Orlando, FL.

[de Vega 00] F. Fernandez de Vega, Laura M. Roa, Marco Tomassini & J. M. Sanchez. *Multipopulation genetic programing applied to burn diagnosing*. In Proceedings of the 2000 Congress on Evolutionary Computation CEC'00, pp 1292–1296, La Jolla Marriott Hotel La Jolla, CA, USA, 6-9 July 2000. IEEE Press.

[Deschain 00] Larry M. Deschain, Fred A. Zafran, Janardan J. Patel, David Amick, Robert Pettit, Frank D. Francone, Peter Nordin, Edward Dilkes & Laurene V. Fausett. *Solving the unsolved using machine learning, data mining and knowledge discovery to model a complex production process*. In M. Ades, (Ed.), Advanced Technology Simulation Conference, Washington, DC, USA, 22-26 April 2000.

[Deschain 01] Larry M. Deschain, Janardan J. Patel, Ronald D. Guthrie, Joseph T. Grimski & M. J. Ades. *Using linear genetic programming to develop a C/C++ simulation model of a waste incinerator*. In M. Ades, (Ed.), Advanced Technology Simulation Conference, Seattle, USA, 22-26 April 2001.

[Dorado 02] Julian Dorado, Juan R. Rabuñal, Jerónimo Puertas, Antonino Santos & Daniel Rivero. *Prediction and modelling of the flow of a typical urban basin through genetic programming*. In Stefano Cagnoni, Jens Gottlieb, Emma Hart, Martin Middendorf & Günther Raidl, (Eds.), Applications of Evolutionary Computing, Proceedings of Evoworkshops2002: EVOCOP, EVOIASP, EVOSTIM, Vol. 2279 of *LNCS*, pp 190–201, Kinsale, Ireland, 3-4 April 2002. Springer-Verlag.

[Dunay 94] B. D. Dunay, F. E. Petry & W. P Buckles. *Regular language induction with genetic programming*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, pp 396–400, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Ebner 99] Marc Ebner. *Evolving an environment model for robot localization*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 184–192, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[EDIF 02] EDIF. *EDIF web pages*. `http://www.edif.org/`, January 2002.

[Eggermont 99] J. Eggermont, A. E. Eiben & J. I. van Hemert. *Adapting the fitness function in GP for data mining*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 193–202, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Erba 01] Massimiliano Erba, Roberto Rossi, Valentino Liberali & Andrea Tettamanzi. *An evolutionary approach to automatic generation of VHDL code for low-power digital filters*. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi & William B. Langdon, (Eds.), Genetic Programming, Proceedings of EuroGP'2001, Vol. 2038 of *LNCS*, pp 36–50, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[Eskin 99] E. Eskin & Eric V. Siegel. *Genetic programming applied to othello: introducing students to machine learning research*. In 30th Technical Symposium of the ACM Special Interest Group in Computer Science Education, New Orleans, LA, USA, March 1999.

[Esparcia-Alcázar 96] Anna I. Esparcia-Alcázar & Ken C. Sharman. *Some applications of genetic programming in digital signal processing*. In John R. Koza, (Ed.), Late Breaking Papers at the Genetic Programming 1996 Conference, pp 24–31, Stanford University, CA, USA, July 1996. Stanford Bookstore.

[Evonet 02] Evonet. *Genetic Programming systems*. `http://evonet.dcs.napier.ac.uk/evoweb/resources/software/keywordlist2.05.html`, 2002.

[Fairfield 84] R.C Fairfield, R.L. Mortenson & K.B. Coulthart. *An LSI random number generator*. In G.R. Blakley & David Chaum, (Eds.), Proceedings of CRYPTO ‘84, Vol. 196 of *LNCS*, pp 203–241. Springer-Verlag, 1984.

[Fernandez 99] Francisco Fernandez, Marco Tomassini & J. M. Sanchez. *Solving the ant and the even parity-5 problems by means of parallel genetic programming*. In Scott Brave & Annie S. Wu, (Eds.), Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference, pp 88–92, Orlando, FL, USA, 13 July 1999.

[Fernandez 00a] Francisco Fernandez & Marco Tomassini. *Genetic programming and reconfigurable hardware: A proposal for solving the problem of placement and routing*. In Conor Ryan, Una-May O’Reilly & William B. Langdon, (Eds.), Graduate Student Workshop, pp 265–268, Las Vegas, Nevada, USA, 8 July 2000.

[Fernandez 00b] Francisco Fernandez, Marco Tomassini, William F. Punch III & J. M. Sanchez. *Experimental study of multipopulation parallel genetic programming*. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin & Terence C. Fogarty, (Eds.), Genetic Pprogramming, Proceedings of EuroGP’2000, Vol. 1802 of *LNCS*, pp 283–293, Edinburgh, United Kingdom, 15-16 April 2000. Springer-Verlag.

[Ferreira 01] Candida Ferreira. *Gene expression programming: a new adaptive algorithm for solving problems*. Complex Systems, vol. 13, no. 2, pp 87–129, 14 November 2001.

[Ferrer 95] Gabriel J. Ferrer & Worthy N. Martin. *Using genetic programming to evolve board evaluation functions for a boardgame*. In 1995 IEEE Conference on Evolutionary Computation, Vol. 2, p 747, Perth, Australia, 29 November - 1 December 1995. IEEE Press.

[Feynman 96] Richard P. Feynman, Robin W. Allen & Anthony J. G. Hey. Feynman Lectures on Computation, Chap. Theory of Computation. Addison Wesley, 1st edition, 1996.

[Fogarty 98] Terrance Fogarty, Julian Miller & Peter Thompson. *Evolving digital logic circuits on Xilinx 6000 family FPGAs*. In P K Chawdhry, R Roy & R K Pant, (Eds.), Soft Computing in Engineering Design and Manufacturing, pp 299–305. Springer-Verlag, 1998.

[Fogel 66] Lawrence J. Fogel, Alvin J. Owens & Michael J. Walsh. *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons Inc., 1966.

[Forsyth 81] Richard Forsyth. *BEAGLE A Darwinian approach to pattern recognition*. Kybernetics, vol. 10, pp 159–166, 1981.

[Fourmilab 02] Fourmilab. *HotBits: Genuine random numbers, generated by radioactive decay*. `http://www.fourmilab.ch/hotbits/`, February 2002.

[Fraser 95] Christopher Fraser & David Hanson. *A retargetable C compiler: Design and implementation*. The Benjamin/Cummings Publishing Company Inc., 1st edition, 1995. ISBN: 0-8053-1670-1.

[Friedberg 58] R Friedberg. *A learning machine, part I*. IBM Journal of Research and Development, vol. 2, pp 2–13, 1958.

[Friedberg 59] R. Friedberg, B Dunham & J. North. *A learning machine, part II*. IBM Journal of Research and Development, vol. 3, pp 282–287, 1959.

[Fujiki 87] Cory Fujiki & John Dickinson. *Using the genetic algorithm to generate LISP source code to solve the prisoners dilemma*. In John J Grefenstette, (Ed.), Genetic Algorithms and the Applications: Proceedings of the Second International Conference of Genetic Algorithms., pp 236–240. MIT, Cambridge., 1987.

[Galloway 95] David Galloway. *The Transmogrifier C Hardware Description Language and Compiler for FPGAs*. In Peter Athanas & Kenneth L. Pocek, (Eds.), IEEE Symposium on FPGAs for Custom Computing Machines, pp 136–144, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[Garcia 99] Santiago Garcia, Fermin Gonzalez & Luciano Sanchez. *Evolving fuzzy rule based classifiers with GAP: A grammatical approach*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 203–210, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Ghanea-Hercock 94] R. Ghanea-Hercock & A. P Fraser. *Evolution of autonomous robot control architectures*. In T. C. Fogarty, (Ed.), Evolutionary Computing, LNCS, Leeds, United Kingdom, 11-13 April 1994. Springer-Verlag.

[Gilbert 99] Richard J. Gilbert, Helen E. Johnson, Michael K. Winson, Jem J. Rowland, Royston Goodacre, Aileen R. Smith, Michael A. Hall & Douglas B. Kell. *Genetic programming as an analytical tool for metabolome data*. In W. B. Langdon, Riccardo Poli, Peter Nordin & Terry Fogarty, (Eds.), Late-Breaking Papers of EuroGP'99, pp 23–33, Goteborg, Sweeden, 26-27 May 1999.

[Goldberg 89] David E. Goldberg, Bradley Korb & Kalyanmoy Deb. *Messy Genetic Algorithms: Motivation, Analysis, and First Results*. Complex Systems, vol. 3, no. 5, pp 493–530, 1989.

[Golubski 99] Wolfgang Golubski & Thomas Feuring. *Evolving neural network structures by means of genetic programming*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 211–220, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Goodacre 99] Royston Goodacre & Richard J. Gilbert. *The detection of caffeine in a variety of beverages using curie-point pyrolysis mass spectrometry and genetic programming*. The Analyst, vol. 124, pp 1069–1074, 1999.

[Goodacre 00] Royston Goodacre, Beverley Shann, Richard J. Gilbert, Éadaoin M. Timmins, Aoife V. McGovern, Bjorn K. Alsberg, Douglas B. Kell & Niall A. Logan. *The detection of the dipicolinic acid biomarker in bacillus spores using curie-point pyrolysis mass spectrometry and fourier-transform infrared spectroscopy*. Analytical Chemistry, vol. 72, no. 1, pp 119–127, 1 January 2000.

[GPMail 02] GPMail. *Genetic Programming mailing list*, 2002.

[Graham 96] Paul Graham & Brent Nelson. *Genetic algorithms in software and in hardware - a performance analysis of workstation and custom computing machine implementations*. In Kenneth L. Pocek & Jeffrey Arnold, (Eds.), Proceedings of the Fourth IEEE Symposium of FPGAs for Custom Computing Machines., pp 216–225, Napa Valley, CA, April 1996. IEEE Computer Society Press.

[Grant 00] Michael Sean Grant. *An investigation into the suitability of genetic programming for computing visibility areas for sensor planning*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, United Kingdom, May 2000.

[Gruau 94] Frederic Gruau. *Genetic micro programming of neural networks*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 24, pp 495–518. MIT Press, Cambridge, MA, USA, 1994.

[Gruska 97] Jeff Gruska. *Foundations of Computing*. International Thompson Computer Press, 1997.

[Haddow 99] Pauline Haddow & Gunnar Tufte. *Evolving a Robot Controller in Hardware*. In In Proceedings of Norsk Informatikkonferanse (NIK'99), Radisson SAS Royal Garden Hotel, Trondheim, November 1999.

[Handley 93] Simon G. Handley. *Automatic learning of a detector for alpha-helices in protein sequences via genetic programming*. In Stephanie Forrest, (Ed.), Proceedings of the $5^{\text{th}}$ International Conference on Genetic Algorithms, ICGA-93, pp 271–278, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[Handley 94a] Simon G. Handley. *The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions.* In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 18, pp 391–407. MIT Press, Cambridge, MA, USA, 1994.

[Handley 94b] Simon G. Handley. *On the use of a directed acyclic graph to represent a population of computer programs.* In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, pp 154–159, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Harris 96] Christopher Harris & Bernard Buxton. *GP-COM: A distributed, component-based genetic programming system in C.* In John R. Koza, David E. Goldberg & David B. Fogel, (Eds.), Genetic Programming 1996: Proceedings of the First Annual Conference, pp 28–31, Stanford University, CA, USA, July 1996. MIT Press.

[Harris 00] Sarah Harris. *Genetically-learned 7-input parity function by an 8 x 8 FPGA.* In John R. Koza, (Ed.), Genetic Algorithms and Genetic Programming at Stanford 2000, pp 214–220. Stanford Bookstore, Stanford, CA, 94305-3079 USA, June 2000.

[Haynes 95] Thomas D. Haynes, Roger L. Wainwright, Sandip Sen & Dale A. Schoenefeld. *Strongly typed genetic programming in evolving cooperation strategies.* In L. Eshelman, (Ed.), Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA-95), pp 271–278, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Haynes 96] Thomas D. Haynes, Dale A. Schoenefeld & Roger L. Wainwright. *Type inheritance in strongly typed genetic programming.* In Peter J. Angeline & Kenneth. E. Kinnear, Jr., (Eds.), Advances in Genetic Programming 2, Chap. 18, pp 359–376. MIT Press, Cambridge, MA, USA, 1996.

[Hernández-Aguirre 00] Arturo Hernández-Aguirre, Bill P. Buckles & Carlos A. Coello-Coello. *Gate-level synthesis of boolean functions using binary multiplexers and genetic programming.* In Proceedings of the 2000 Congress on Evolutionary Computation CEC'00, pp 675–682, La Jolla Marriott Hotel La Jolla, CA, USA, 6-9 July 2000. IEEE Press.

[Heywood 00] Malcom I. Heywood & Nur Zincir-Heywood. *Register based genetic programming on FPGA computing platforms.* In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'2000, Vol. 1802 of *LNCS*, pp 44–59, Edinburgh, United Kingdom, April 2000. Springer-Verlag.

[Hicklin 86] Joseph F. Hicklin. *Application of the genetic algorithm to automatic program generation.* Master's thesis, Dept of Computer Science, University of Idaho., 1986.

[Higuchi 93] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis & T. Furuya. *Evolvable hardware - Genetic based hardware evolution at gate and hardware description (HDL) levels.* Technical report, ETL Tech. Report 93-4, 1993.

[Hoare 85] C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Holland 75] John Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[Holland 92] John Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 2nd edition, 1992.

[Hollingworth 00] Gordon Hollingworth, Steve Smith & Andy Tyrrell. *Safe Intrinsic Evolution of Virtex Devices*. In Proceedings of 2nd NASA/DoD Workshop on Evolvable Hardware, July 2000.

[Holmes 96] Paul Holmes & Peter J. Barclay. *Functional languages on linear chromosomes*. In John R. Koza, David E. Goldberg, David B. Fogel & Rick L. Riolo, (Eds.), Genetic Programming 1996: Proceedings of the First Annual Conference, p 427, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Howard 99] Daniel Howard, Simon C. Roberts & Richard Brankin. *Evolution of ship detectors for satellite SAR imagery*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 135–148, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Howard 02] Daniel Howard & Simon C. Roberts. *The prediction of journey times on motorways using genetic programming.* In Stefano Cagnoni, Jens Gottlieb, Emma Hart, Martin Middendorf & Günther Raidl, (Eds.), Applications of Evolutionary Computing, Proceedings of Evoworkshops2002: EVOCOP, EVOIASP, EVOSTIM, Vol. 2279 of *LNCS*, pp 210–211, Kinsale, Ireland, 3-4 April 2002. Springer-Verlag.

[Iba 94] Hitoshi Iba, Hugo de Garis & Taisuke Sato. *Genetic programming using a minimum description length principle*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 12, pp 265–284. MIT Press, Cambridge, MA, USA, 1994.

[IBM and Motorola 95] IBM and Motorola. *PowerPC Embedded Application Binary Interface, Version 1.0*. `http://www.esofta.com/pdfs/ppceabi.pdf`, 10 January 1995.

[Jamro 01] Ernset Jamro & Wiatr Kazimierz. *Genetic programming in FPGA implementation of addition as a part of the convolution*. In Proceedings of the IEEE International Conference Digital System Design, pp 466–473. IEEE Computer Society Press, September 2001.

[Jannink 94] Jan Jannink. *Cracking and co-evolving randomizers*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 20, pp 425–443. MIT Press, Cambridge, MA, USA, 1994.

[Jefferson 90] David Jefferson, Robert Collins, Claus Cooper, Michael Dyer, Margot Flowers, Richard Karf, Charles Taylor & Alan Wang. *Evolution as a theme in Artificial Life: The Genesys/Tracker system*. In Christopher G. Langton, Charles Taylor, Doyne J. Farmer & Steen Rasmussen, (Eds.), Artificial

Life II: Proceedings of the Workshop on Artificial Life. Addison-Wesley Publishing Company Inc., February 1990.

[Johanson 98] Brad Johanson & Riccardo Poli. *GP-music: an interactive genetic programming system for music generation with automated fitness raters*. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba & Rick Riolo, (Eds.), Genetic Programming 1998: Proceedings of the Third Annual Conference, pp 181–186, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Johnson 96] Brad C. Johnson. *Radix-b extensions to some common empirical tests for pseudorandom number generators*. ACM Transactions on Modelling and Computer Simulation, vol. 6, no. 4, pp 261–273, 1996.

[Juille 96] Hugues Juille & Jordan B. Pollack. *Massively parallel genetic programming*. In Peter J. Angeline & Kenneth. E. Kinnear, Jr., (Eds.), Advances in Genetic Programming 2, Chap. 17, pp 339–358. MIT Press, Cambridge, MA, USA, 1996.

[Kajitani 98] Isamu Kajitani, Tsutomu Hoshino, Daisuke Nishikawa, Hiroshi Yokoi, Shougo Nakaya, Tsukasa Yamauchi, Takeshi Inuo, Nobuki Kajihara, Masaya Iwata, Didier Keymeulen & Tetsuya Higuch. *A gate-level EHW chip: Implementing GA operations and reconfigurable hardware on a single LSI*. In M. Sipper, D. Mange & A. Perez-Uribe, (Eds.), Proceedings of the Second International Conference, Evolvable systems: From Biology to Hardware, ICES ’98, Vol. 1478 of *LNCS*, Lausanne, Switzerland, September 1998. ICES, Springer Verlag.

[Kantschik 99] Wolfgang Kantschik, Peter Dittrich, Markus Brameier & Wolfgang Banzhaf. *MetaEvolution in Graph GP*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP’99, Vol. 1598 of *LNCS*, pp 15–28, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Kantschik 01] Wolfgang Kantschik & Wolfgang Banzhaf. *Linear-Tree GP and its comparison with other GP structures*. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi & William B. Langdon, (Eds.), Genetic Programming, Proceedings of EuroGP’2001, Vol. 2038 of *LNCS*, pp 302–312, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[Kantschik 02] Wolfgang Kantschik & Wolfgang Banzhaf. *Linear graph GP - a new GP structure*. In James. A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan & Andrea G. B. Tettamanzi, (Eds.), Proceedings of the 5$^{th}$ European Conference, EuroGP’2002, Vol. 2278 of *LNCS*, pp 83–92, Kinsale, Ireland, 2002. Springer Verlag.

[Kernighan 88] Brian W. Kernighan & Dennis M. Ritchie. *The C programming language*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 2$^{nd}$ edition, 1988.

[Kinnear, Jr. 93] Kenneth E. Kinnear, Jr. *Generality and difficulty in genetic programming: evolving a sort*. In Stephanie Forrest, (Ed.), Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, pp 287–294, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[Kirkwood 97] I. M. A. Kirkwood, S. H. Shami & M. C. Sinclair. *Discovering simple fault-tolerant routing rules using genetic programming*. In Proceedings of the Third International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA'97, University of East Anglia, Norwich, United Kingdom, 4 April 1997. Springer Verlag.

[Kitaura 99] Osamu Kitaura, Hideaki Asada, Motoaki Matsuzaki, Takamitsu Kawai, Hideki Ando & Toshio Shimada. *A custom computing machine for genetic algorithms without pipeline stalls*. In Proceedings of the 1999 IEEE International Conference on Systems, Man and Cybernetics, Vol. V, pp 577–584, October 1999.

[Klahold 98] Stefan Klahold, Steffen Frank, Robert E. Keller & Wolfgang Banzhaf. *Exploring the possibilites and restrictions of genetic programming in Java bytecode*. In John R. Koza, (Ed.), Late Breaking Papers at the Genetic Programming 1998 Conference, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Stanford University Bookstore.

[Knuth 69] Donald E. Knuth. *Seminumerical Algorithms*, Vol. 2. Addison-Wesley Publishing Company, 1969.

[Koizumi 01] Shinya Koizumi, Shin'ichi Wakabayashi, Tetsushi Koide, Kazunari Fujiwara & Norimichi Imura. *A RISC processor for high-speed execution of genetic algorithms*. In Lee Spector, Erik D. Goodman, Annie Wu, William B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon & Edmund Burke, (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pp 1338–1345, San Francisco, CA, USA, July 2001. International society for genetic and evolutionary computation (ISGEC), Morgan Kaufmann Publishers, San Francisco, USA.

[Koopman Jr. 89] Phillip J. Koopman Jr. *Stack computers: the new wave*. Ellis Horwood, 1989. Also avilable online: `http://www.cs.cmu.edu/~koopman/stack_computers/`.

[Koza 89] John R. Koza. *Hierarchical genetic algorithms operating on populations of computer programs*. In N. S. Sridharan, (Ed.), Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89, Vol. 1, pp 768–774. Morgan Kaufmann, 20-25 August 1989.

[Koza 90a] John R. Koza. *Genetic evolution and co-evolution of computer programs*. In Christopher G. Langton, Charles Taylor, Doyne J. Farmer & Steen Rasmussen, (Eds.), Artificial Life II: Proceedings of the Workshop on Artificial Life, pp 603–629. Addison-Wesley, February 1990.

[Koza 90b] John R. Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Technical Report

STAN-CS-90-1314, Dept. of Computer Science, Stanford University, June 1990.

[Koza 92] John R. Koza. *Genetic programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[Koza 94] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, May 1994.

[Koza 96a] John R. Koza, David Andre, Forrest H. Bennett III & Martin A. Keane. *Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming*. In John R. Koza, David E. Goldberg, David B. Fogel & Rick L. Riolo, (Eds.), Genetic Programming 1996: Proceedings of the First Annual Conference, pp 132–149, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Koza 96b] John R. Koza, Forrest H. Bennett III, David Andre & Martin A. Keane. *Automated design of both the topology and sizing of analog electrical circuits using genetic programming*. In John S. Gero & Fay Sudweeks, (Eds.), Artificial Intelligence in Design '96, pp 151–170, Dordrecht, 1996. Kluwer Academic.

[Koza 97a] John R. Koza, David Andre, Forrest H. Bennett III & Martin A. Keane. *Design of a high-gain operational amplifier and other circuits by means of genetic programming*. In Peter J. Angeline, Robert G. Reynolds, John R. McDonnell & Russ Eberhart, (Eds.), Evolutionary Programming vi. 6th International Conference, EP'97, Vol. 1213 of *LNCS*, pp 125–136, Indianapolis, Indiana, USA, 1997. Springer-Verlag.

[Koza 97b] John R. Koza, Forrest H. Bennett III, Jeffrey L. Hutchings, Stephen L. Bade, Martin A. Keane & David Andre. *Evolving sorting networks using genetic programming and the rapidly reconfigurable Xilinx 6216 field-programmable gate array*. In Proceedings of the 31st Asilomar Conference on Signals, Systems, and Computers. IEEE Press, 1997.

[Koza 97c] John R. Koza, Forrest H. Bennett III, Jason Lohn, Frank Dunlap, Martin A. Keane & David Andre. *Automated synthesis of computational circuits using genetic programming*. In Proceedings of the 1997 IEEE international conference on evolutionary computation, pp 447–452, Indianapolis, 13-16 April 1997. IEEE Press.

[Koza 98a] John R. Koza, Forrest H. Bennett III & David Andre. *Using programmatic motifs and genetic programming to classify protein sequences as to extracellular and membrane cellular location*. In V. William Porto, N. Saravanan, D. Waagen & A. E. Eiben, (Eds.), Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming, Vol. 1447 of *LNCS*, Mission Valley Marriott, San Diego, CA, USA, March 1998. Springer-Verlag.

[Koza 98b] John R. Koza, Forrest H. Bennett III, Jeffrey L. Hutchings, Stephen L. Bade, Martin A. Keane & David Andre. *Evolving computer programs* *using rapidly reconfigurable FPGAs and genetic programming*. In Jason Cong, (Ed.), FPGA'98 Sixth International Symposium on Field Programmable Gate Arrays, pp 209–219, Doubletree Hotel, Monterey, CA, USA, 22-24 February 1998. ACM Press.

[Koza 99a] John R. Koza & Forrest H. Bennett III. *Automatic synthesis, placement, and routing of electrical circuits by means of genetic programming*. In Lee Spector, William B. Langdon, Una-May O'Reilly & Peter J. Angeline, (Eds.), Advances in Genetic Programming 3, Chap. 6, pp 105–134. MIT Press, Cambridge, MA, USA, June 1999.

[Koza 99b] John R. Koza, Forrest H. Bennett III & Oscar Stiffelman. *Genetic programming as a Darwinian invention machine*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 93–108, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[Koza 99c] John R. Koza, David Andre, Forrest H. Bennett III & Martin A. Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999.

[Koza 00a] John R. Koza, Forrest H. Bennett III, David Andre & Martin A. Keane. *Automatic design of analog electrical circuits using genetic programming*. In Hugh Cartwright, (Ed.), Intelligent data analysis in science, Chap. 8, pp 172–200. Oxford University Press, Oxford, 2000.

[Koza 00b] John R. Koza, William Comisky & Jessen Yu. *Automatic synthesis of a wire antenna using genetic programming*. In Darrell Whitley, (Ed.), Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, pp 179–186, Las Vegas, Nevada, USA, 8 July 2000.

[Koza 00c] John R. Koza, Martin A. Keane, Jessen Yu, Forrest H. Bennett III & William Mydlowec. *Evolution of a controller with a free variable using genetic programming*. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'2000, Vol. 1802 of *LNCS*, pp 91–105, Edinburgh, United Kingdom, April 2000. Springer-Verlag.

[Kraft 94] D. H. Kraft, F. E. Petry, W. P. Buckles & T. Sadasivan. *The use of genetic programming to build queries for information retrieval*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, pp 468–473, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Kuhling 02] Felix Kuhling, Krister Wolff & Peter Nordin. *A brute-force approach to automatic induction of machine code on CISC architectures*. In James. A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan & Andrea G. B. Tettamanzi, (Eds.), Proceedings of the 5th European Conference, EuroGP'2002, Vol. 2278 of *LNCS*, pp 288–297, Kinsale, Ireland, April 2002. EvoGP, Springer-Verlag.

[Langdon 97] W. B. Langdon. *Fitness causes bloat: simulated annealing, hill climbing and populations*. Technical report CSRP-97-22, University of Birmingham, School of Computer Science, 2 September 1997.

[Langdon 98a] William B. Langdon. *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, Vol. 1 of *Genetic Programming*. Kluwer, Boston, USA, 24 April 1998.

[Langdon 98b] William B. Langdon & Riccardo Poli. *Why ants are hard*. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba & Rick Riolo, (Eds.), Genetic programming 1998: proceedings of the third annual conference, pp 193–201, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Langdon 99] William B. Langdon. *Size fair and homologous tree genetic programming crossovers*. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela & Robert E. Smith, (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, Vol. 2, pp 1092–1097, Orlando, FL, USA, July 1999. Morgan Kaufmann.

[Langdon 00] William B. Langdon & Peter Nordin. *Seeding GP populations*. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin & Terence C. Fogarty, (Eds.), Genetic programming, proceedings of EuroGP'2000, Vol. 1802 of *LNCS*, pp 304–315, Edinburgh, United Kingdom, 15-16 April 2000. Springer-Verlag.

[Langdon 01] William B. Langdon & Peter Nordin. *Evolving hand-eye coordination for a humanoid robot with machine code genetic programming*. In Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi & William B. Langdon, (Eds.), Genetic Programming, Proceedings of EuroGP'2001, Vol. 2038 of *LNCS*, pp 313–324, Lake Como, Italy, April 2001. EvoNET, Springer-Verlag.

[Langdon 02a] William B. Langdon. *GPdata*. `ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code`, 2002.

[Langdon 02b] William B. Langdon, S. J. Barrett & B. F. Buxton. *Combining decision trees and neural networks for drug discovery*. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan & Andrea G. B. Tettamanzi, (Eds.), Genetic Programming, Proceedings of the 5th European Conference, EuroGP'2002, Vol. 2278 of *LNCS*, pp 60–70, Kinsale, Ireland, April 2002. Springer-Verlag.

[Lavarand 02] Lavarand. *Harnessing the power of Lava Lite lamps to generate truly random numbers*. `http://www.lavarnd.org/`, February 2002.

[L'Ecuyer 88] P. L'Ecuyer. *Efficient and portable combined random number generators*. Communications of the ACM, vol. 31, no. 6, 1988.

[L'Ecuyer 90] P. L'Ecuyer. *Random numbers for simulation*. Communications of the ACM, vol. 33, no. 10, 1990.

[Levi 99] Delon Levi & Steven A. Guccione. *Genetic FPGA: Evolving stable circuits on mainstream FPGA devices*. In A Stoica, D Keymeulen & J Lohn, (Eds.), Proceedings of the First NASA/DoD Workshop on Evolvable Hardware, pp 12–17. IEEE Computer Society, July 1999.

[Lindholm 99] Tim Lindholm & Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 8 April 1999.

[Loizides 01] A. Loizides, M. Slater & W. B. Langdon. *Measuring facial emotional expressions using genetic programming*. In WSC6, 6th World Conference on Soft Computing in Industrial Applications. Springer-Verlag, 10–24 September 2001. Forthcoming.

[LOTOS 00] LOTOS. *Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. ISO 8807:1989, 21 June 2000.

[Luke 97] Sean Luke & Lee Spector. *A comparison of crossover and mutation in genetic programming*. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba & Rick L. Riolo, (Eds.), Genetic Programming 1997: Proceedings of the Second Annual Conference, pp 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[Luke 98a] Sean Luke. *Genetic programming produced competitive soccer softbot teams for robocup97*. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba & Rick Riolo, (Eds.), Genetic Programming 1998: Proceedings of the Third Annual Conference, pp 214–222, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Luke 98b] Sean Luke & Lee Spector. *A revised comparison of crossover and mutation in genetic programming*. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba & Rick Riolo, (Eds.), Genetic Programming 1998: Proceedings of the Third Annual Conference, pp 208–213, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[Luke 00a] Sean Luke. *Code growth is not caused by introns*. In Darrell Whitley, (Ed.), Late breaking papers at the 2000 genetic and evolutionary computation conference, pp 228–235, Las Vegas, Nevada, USA, 8 July 2000.

[Luke 00b] Sean Luke. *Two fast tree-creation algorithms for genetic programming*. IEEE Transactions on Evolutionary Computation, vol. 4, no. 3, pp 274–283, September 2000.

[Lukschandl 98] Eduard Lukschandl, Mangus Holmlund & Erik Moden. *Automatic evolution of Java bytecode: first experience with the Java virtual machine*. In Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty & Wolfgang Banzhaf, (Eds.), Late Breaking Papers at EuroGP'98: The First European Workshop on Genetic Programming, pp 14–16, Paris, France, April 1998. CSRP-98-10, The University of Birmingham, United Kingdom.

[Marconi 00] Marconi. SS7 benchmark trial of Handel-C. (Unpublished techical report), 2000.

[Marsaglia 94] George Marsaglia. *Yet another RNG*. Posted to sci.stat.math, 1 August 1994.

[Marsaglia 01] George Marsaglia. *Web site for Diehard random number test suite.* `http://stat.fsu.edu/geo/`, 2001. Last visited 15/June/2001.

[Martin 98] Peter Martin. *An investigation into the use of Genetic Programming for Intelligent Network service creation.* Masters dissertation, Bournemouth University, 1998.

[Martin 00] Peter Martin. *Genetic programming for service creation in intelligent networks*. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'2000, Vol. 1802 of *LNCS*, pp 106–120, Edinburgh, United Kingdom, April 2000. Springer-Verlag.

[Martin 01] Peter Martin. *A hardware implementation of a genetic programming system using FPGAs and Handel-C.* Genetic Programming and Evolvable Machines, vol. 2, no. 4, pp 317–343, 2001.

[Martin 02a] Peter Martin. *An analysis of random number generators for a hardware implementation of genetic programming using FPGAs and Handel-C.* In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke & N. Jonoska, (Eds.), GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp 837–844, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[Martin 02b] Peter Martin. *A pipelined hardware implementation of genetic programming using FPGAs and Handel-C.* In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan & Andrea G. B. Tettamanzi, (Eds.), Proceedings of Eurogp'2002, number 2278 in LNCS, pp 1–12, Kinsale, Ireland, March 2002. EvoNet, Springer Verlag.

[Martin 02c] Peter Martin & Riccardo Poli. *Crossover operators for a hardware implementation of GP using FPGAs and Handel-C.* In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke & N. Jonoska, (Eds.), GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp 845–852, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[Maruyama 99] Tsutomu Maruyama, Terunobu Funatsu, Minenobu Seki, Yoshiki Yamaguchi & Tsutomu Hoshino. *A field-programmable gate-array system for evolutionary computation.* IPSJ Journal, vol. 40, no. 5, 1999.

[Maruyama 00] Tsutomu Maruyama & Tsutomu Hoshino. *A C to HDL compiler for pipeline processing on FPGAs.* In In Proceedings of the 8th Symposium on Field-Programmable Custom Computing Machines, FCCM'00, pp 101–110, Marriott at Napa Valley, Napa, CA, USA, April 2000. IEEE Computer Society Press.

[Masand 94] Brij Masand. *Optimising confidence of text classification by evolution of symbolic expressions*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 21, pp 445–458. MIT Press, Cambridge, MA, USA, 1994.

[Maxwell III 94] Sidney R. Maxwell III. *Experiments with a coroutine model for genetic programming*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Vol. 1, pp 413–417a, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[McPhee 02] Nicholas Freitag McPhee & Riccardo Poli. *Using schema theory to explore interactions of multiple operators*. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke & N. Jonoska, (Eds.), GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp 853–860, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[Meissner 01] M. Meissner. *Web site for Power-pc simulator - psim*. `http://sources.redhat.com/psim/`, 2001. Last visited 15/June/2001.

[Meysenburg 97a] Mark M. Meysenburg. *The effect of pseudo-random number generator quality on the performance of a simple genetic algorithm*. Master's thesis, University of Idaho, 1997.

[Meysenburg 97b] Mark M. Meysenburg & James A. Foster. *The quality of pseudo-random number generators and simple genetic algorithm performance*. In Thomas Bäck, (Ed.), Genetic Algorithms: Proceedings of the Seventh International Conference, pp 276–282, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[Meysenburg 99a] Mark M. Meysenburg & James A. Foster. *Random generator quality and GP performance*. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela & Robert E. Smith, (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, Vol. 2, pp 1121–1126, Orlando, FL, USA, July 1999. Morgan Kaufmann.

[Meysenburg 99b] Mark M. Meysenburg & James A. Foster. *Randomness and GA performance, revisited*. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela & Robert E. Smith, (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, Vol. 1, pp 425–432, Orlando, FL, USA, July 1999. Morgan Kaufmann.

[Meysenburg 02] Mark M. Meysenburg, Daniel Hoelting, Duane McElvain & James A. Foster. *How random generator quality impacts GA performance*. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke & N. Jonoska, (Eds.), GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp 480–487, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[Miller 00] Julian F. Miller & Peter Thomson. *Cartesian genetic programming*. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'2000, Vol. 1802 of *LNCS*, pp 121–132, Edinburgh, United Kingdom, April 2000. Springer-Verlag.

[Montana 95] David J. Montana. *Strongly typed genetic programming*. Evolutionary Computation, vol. 3, no. 2, pp 199–230, 1995.

[Montana 98] David Montana, Robert Popp, Suraj Iyer & Gordon Vidaver. *Evolvaware: genetic programming for optimal design of hardware-based algorithms*. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba & Rick Riolo, (Eds.), Genetic Programming 1998: Proceedings of the Third Annual Conference, pp 869–874, University of Wisconsin, Madison, Wisconsin, USA, July 1998. Morgan Kaufmann.

[Moore 70] Charles H. Moore & Geoffrey C. Leach. *FORTH – A Language for Interactive Computing*. Technical report, Mohasco Industries, Inc., Amsterdam, NY, 1970.

[Moore 98] Frank W. Moore. *Genetic programming solves the three-dimensional missile countermeasures optimization problem under uncertainty*. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba & Rick Riolo, (Eds.), Genetic Programming 1998: Proceedings of the Third Annual Conference, pp 242–245, University of Wisconsin, Madison, Wisconsin, USA, July 1998. Morgan Kaufmann.

[Moore 01] Chuck Moore. *Forth chips*. `http://www.colorforth.com/chips.html`, July 2001.

[Motorola 98] Motorola. *PowerQUICC MPC860 User's Manual*. Motorola Inc., Motorola Literature Distribution, P.O. Box 5405, Denver, Colorado 80217, USA., 1998. Revision 1.

[Nguyen 94] Thang Nguyen & Thomas Huang. *Evolvable 3D modeling for model-based object recognition systems*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 22, pp 459–475. MIT Press, Cambridge, MA, USA, 1994.

[NIST 02] NIST. *Prefixes for binary multiples*. `http://physics.nist.gov/cuu/Units/binary.html`, 3 January 2002.

[Nordin 94] Peter Nordin. *A compiling genetic programming system that directly manipulates the machine code*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 14, pp 311–331. MIT Press, Cambridge, MA, USA, 1994.

[Nordin 95a] Peter Nordin & Wolfgang Banzhaf. *Complexity compression and evolution*. In L. Eshelman, (Ed.), Genetic algorithms: proceedings of the sixth international conference (icga95), pp 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Nordin 95b] Peter Nordin & Wolfgang Banzhaf. *Evolving turing-complete programs for a register machine with self-modifying code*. In L. Eshelman, (Ed.), Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA-95), pp 318–325, Pittsburgh, PA, USA, July 1995. Morgan Kaufmann.

[Nordin 99a] Peter Nordin, Wolfgang Banzhaf & Frank D. Francone. *Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover*. In Lee Spector, William B. Langdon, Una-May O'Reilly & Peter J. Angeline, (Eds.), Advances in Genetic Programming 3, Chap. 12, pp 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[Nordin 99b] Peter Nordin, Anders Eriksson & Mats Nordahl. *Genetic reasoning: evolutionary induction of mathematical proofs*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 221–231, Goteborg, Sweden, April 1999. Springer-Verlag.

[Nowostawski 99] Marius Nowostawski & Riccardo Poli. *Parallel genetic algorithm taxonomy*. In Proceedings of the Third International Conference on Knowlege-based Intelligent Information Engineering Systems KES'99, pp 88–92. IEEE Computer Society, August 1999.

[Oakley 94] Howard Oakley. *Two scientific applications of genetic programming: stack filters and non-linear equation fitting to chaotic data*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 17, pp 369–389. MIT Press, Cambridge, MA, USA, 1994.

[Olsson 95] J. R. Olsson. *Inductive functional programming using incremental program transformation*. Artificial Intelligence, vol. 74, no. 1, pp 55–83, March 1995.

[O'Neill 99a] Michael O'Neill. *Automatic programming with grammatical evolution*. In Una-May O'Reilly, (Ed.), GECCO-99 Student Workshop, Orlando, FL, USA, 13 July 1999.

[O'Neill 99b] Michael O'Neill & Conor Ryan. *Under the hood of grammatical evolution*. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela & Robert E. Smith, (Eds.), Proceedings of the Genetic and Evolutionary Computation Conference, Vol. 2, pp 1143–1148, Orlando, FL, USA, July 1999. Morgan Kaufmann.

[Page 91] Ian Page & Wayne Luk. *Compiling Occam into FPGAs*. In FPGAs. International workshop on Field Programmable Logic and Applications, pp 271–283, Oxford, United Kingdom, September 1991.

[Page 96] Ian Page. *Constructing hardware-software systems from a single description*. Journal of VLSI Signal Processing, vol. 1, no. 12, pp 87–107, January 1996.

[Page 97] Ian Page. *Compiling video algorithms into hardware*. Embedded System Enginerring, September 1997.

[Page 99] Jonathan Page, Riccardo Poli & Wiiliam B. Langdon. *Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 39–49, Goteborg, Sweden, April 1999. Springer-Verlag.

[Patterson 96] David A. Patterson & John L. Hennessy. *Computer Architecture. A Quantitative Approach*. Morgan Kaufmann publishers Inc., San Francisco, CA, USA, 2nd edition, 1996. ISBN: 1-55860-329-8.

[Perkins 00] Simon Perkins, Reid Porter & Neal Harvey. *Everything on the chip: a hardware-based self-contained spatially-structured genetic algorithm for signal processing*. In Julian Miller, Adrian Thompson, Peter Thomson & Terrance C. Fogarty, (Eds.), Proceedings Of the 3rd International Conference on Evolvable Systems: From Biology to Hardware (ICES 2000), Vol. 1801 of *LNCS*, pp 165–174, Edinburgh, United Kingdom, 2000. Springer-Verlag.

[Perkis 94] Tim Perkis. *Stack-based genetic programming*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Vol. 1, pp 148–153, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Perry 94] J. E. Perry. *The effect of population enrichment in genetic programming*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, pp 456–461, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Poli 99] Riccardo Poli. *Parallel Distributed Genetic Programming*. In David Corne, Marco Dorigo & Fred Glover, (Eds.), New Ideas in Optimization, Chap. 27. McGraw-Hill, 1999.

[Poli 00] Riccardo Poli & Jonathan Page. *Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code GP and demes*. Genetic Programming And Evolvable Machines, vol. 1, no. 1/2, pp 37–56, April 2000.

[Poli 01a] Riccardo Poli. *General schema theory for genetic programming with subtree-swapping crossover*. In Julian F. Miller, Marco Tomassini, Pier Luca Lanz, Connor Ryan, Andrea G. B. Tettamanzi & William B. Langdon, (Eds.), Genetic Programming, Proceedings of EuroGP'2001, Vol. 2038 of *LNCS*, pp 143–159, Lake Como, Italy, April 2001. EvoNET, Springer-Verlag.

[Poli 01b] Riccardo Poli & Nicholas Freitag McPhee. *Exact Schema Theorems for GP with One-Point and Standard Crossover Operating on Linear Structures and their Application to the Study of the Evolution of Size*. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi & William B. Langdon, (Eds.), Genetic Programming, Proceedings of EuroGP'2001, Vol. 2038 of *LNCS*, pp 126–142, Lake Como, Italy, April 2001. Springer-Verlag.

[Polito 97] J. Polito, J. Daida & T. F. Bersano-Begey. *Musica ex machina: composing 16th-century counterpoint with genetic programming and symbiosis*.

In Peter J. Angeline, Robert G. Reynolds, John R. McDonnell & Russ Eberhart, (Eds.), Evolutionary programming vi: Proceedings of the Sixth Annual Conference on Evolutionary Programming, Vol. 1213 of *LNCS*, Indianapolis, Indiana, USA, 1997. Springer-Verlag.

[Press 86] William H. Press, Brian P. Flannery, Saul A. Teukolsky & William T. Vetterling. *Numerical Recipes, the Art of Scientific Computing*. Cambridge University Press, 1986.

[Pringle 95] William R. Pringle. *ESP: Evolutionary Structured Programming*. Technical report, Penn State University, Great Valley Campus, PA, USA, 1995.

[Putnam 94] Jeffrey B. Putnam. Genetic programming of music. `http://www.nmt.edu/jefu/notes/ep.ps`, 30 August 1994.

[Qureshi 00] Adil Qureshi. *GPsys 2b*. `http://www.cs.ucl.ac.uk/staff/A.Qureshi/gpsys_doc.html`, July 2000.

[Racine 99] Alain Racine, Sana Ben Hamida & Marc Schoenauer. *Parametric coding vs genetic programming: A case study*. In W. B. Langdon, Riccardo Poli, Peter Nordin & Terry Fogarty, (Eds.), Late Breaking Papers of EuroGP'99, pp 13–22, Goteborg, Sweeden, 26-27 May 1999.

[Radi 99] Amr Radi & Riccardo Poli. *Genetic programming discovers efficient learning rules for the hidden and output layers of feedforward neural networks*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 120–134, Goteborg, Sweden, May 1999. Springer-Verlag.

[random.org 02] random.org. *random.org: random numbers*. `http://www.random.org`, 2002. Last visited Jan 2 2002.

[Rechenberg 65] Ingo Rechenberg. *Cybernetic solution path of an experimental problem: Kybernetische losungsansteuerung einer experimentellen forschungsaufgabe*. English translation of a German technical report RAE/LT-1122 (Report-C), Held by Cranfield University, 1965.

[Rechenberg 73] Ingo Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog, Stuttgart, 1973.

[Reynolds 92] Craig W. Reynolds. *An evolved, vision-based behavioral model of coordinated group motion*. In Meyer & Wilson, (Eds.), From Animals to Animats (Proceedings of Simulation of Adaptive Behaviour). MIT Press, 1992.

[Reynolds 94a] Craig W. Reynolds. *Evolution of corridor following behavior in a noisy world*. In Simulation of Adaptive Behaviour (SAB-94), 1994.

[Reynolds 94b] Craig W. Reynolds. *Evolution of obstacle avoidance behaviour:using noise to promote robust solutions*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 10, pp 221–241. MIT Press, Cambridge, MA, USA, 1994.

[Rosca 94] J. P. Rosca & D. H. Ballard. *Learning by adapting representations in genetic programming*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Rosé 99] Carolyn Penstein Rosé. *A genetic programming approach for robust language interpretation*. In Lee Spector, William B. Langdon, Una-May O'Reilly & Peter J. Angeline, (Eds.), Advances in Genetic Programming 3, Chap. 4, pp 67–88. MIT Press, Cambridge, MA, USA, June 1999.

[Rush 94] J. R. Rush, A. P. Fraser & D. P. Barnes. *Evolving co-operation in autonomous robotic systems*. In Proceedings of the IEE International Conference on Control, March 1994.

[Ryan 94] Conor Ryan. *Pygmies and civil servants*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 11, pp 243–263. MIT Press, Cambridge, MA, USA, 1994.

[Ryan 98] Conor Ryan, J. J. Collins & Michael O'Neill. *Grammatical evolution: evolving programs for an arbitrary language*. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer & Terence C. Fogarty, (Eds.), Proceedings of the First European Workshop on Genetic Programming, Vol. 1391 of *LNCS*, pp 83–95, Paris, France, April 1998. Springer-Verlag.

[Samuel 59] Arthur L. Samuel. *Some studies in machine learning using the game of checkers*. IBM Journal of Research and Development, vol. 3, no. 3, pp 210–229, 1959.

[Sarafopoulos 99] Anargyros Sarafopoulos. *Automatic generation of affine IFS and strongly typed genetic programming*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 149–160, Goteborg, Sweden, May 1999. Springer-Verlag.

[Savage 98] John Savage. *Models of Computation: Exploring the Power of Computing*. Addison Wesley, 1998.

[Schmidhuber 87] Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook.* Master's thesis, Institut für Informatik, Technische Universität München, 14 May 1987.

[Schneier 96] Bruce Schneier. *Applied Cryptography. Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc., 1996.

[Schoenauer 96] Marc Schoenauer, Michele Sebag, Francois Jouve, Bertrand Lamy & Habibou Maitournam. *Evolutionary identification of macro-mechanical models*. In Peter J. Angeline & K. E. Kinnear, Jr., (Eds.), Advances in Genetic Programming 2, Chap. 23, pp 467–488. MIT Press, Cambridge, MA, USA, 1996.

[Scott 94] Stephen D. Scott. *HGA: A hardware-based genetic algorithm.* Master's thesis, University of Nebraska-Lincoln, August 1994. Also available in technical report UNL-CSE-94-020, University of Nebraska-Lincoln.

[Sedgewick 84] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1984.

[Sekanina 00] Lukas Sekanina & R. Ruzicka. *Design of the special fast reconfigurable chip using common FPGA*. Proceedings of the Design and Diagnostic of Electronic Circuits and Systems IEEE DDECS'2000, pp 161–168, 2000.

[Seok 00] Ho-Sik Seok, Kwang-Ju Lee & Byoung-Tak Zhang. *An On-Line Learning Method for Object-Locating Robots using Genetic Programming on Evolvable Hardware*. In Masanori Sugisaka, (Ed.), International Symposium on Artificial Life and Robotics, pp 321–324, Oita, Japan, 26-28 January 2000.

[Shackleford 01] Barry Shackleford, Greg Snider, Richard J. Carter, Etsuko Okushi, Mitsuhiro Yasuda, Katsuhiko Seo & Hiroto Yasuura. *A high performance, pipelined, FPGA-based genetic algorithm machine*. Genetic Programming and Evolvable Machines, vol. 2, no. 1, pp 33–60, March 2001.

[Sharman 95] Ken C. Sharman, Anna I. Esparcia Alcazar & Yun Li. *Evolving signal processing algorithms by genetic programming*. In A. M. S. Zalzala, (Ed.), First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA, Vol. 414, pp 473–480, Sheffield, United Kingdom, September 1995. IEE.

[Sidhu 99] Reetinder P.S. Sidhu, Alessandro Mei & Viktor Prasanna. *Genetic programming using self-reconfiguring FPGAs*. In Patrick Lysaght, James Irvine & Reiner Hartenstein, (Eds.), Proceedings Field-Programmable Logic and Applications, Vol. 1673 of *LNCS*, pp 301–312, Glasgow, United Kingdom, August 1999. Springer-Verlag.

[Siegel 94] Eric V. Siegel. *Competitively evolving decision trees against fixed training cases for natural language processing*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 19, pp 409–423. MIT Press, Cambridge, MA, USA, 1994.

[Siegel 96] Eric V. Siegel & Alexander D. Chaffee. *Genetically optimizing the speed of programs evolved to play tetris*. In Peter J. Angeline & K. E. Kinnear, Jr., (Eds.), Advances in Genetic Programming 2, Chap. 14, pp 279–298. MIT Press, Cambridge, MA, USA, 1996.

[Silva 99] Arlindo Silva, Ana Neves & Ernesto Costa. *Evolving controllers for autonomous agents using genetically programmed networks*. In Riccardo Poli, Peter Nordin, William B. Langdon & Terence C. Fogarty, (Eds.), Genetic Programming, Proceedings of EuroGP'99, Vol. 1598 of *LNCS*, pp 255–269, Goteborg, Sweden, May 1999. Springer-Verlag.

[Sims 91] Karl Sims. *Artificial evolution for computer graphics*. ACM Computer Graphics, vol. 25, no. 4, pp 319–328, July 1991. SIGGRAPH '91 Proceedings.

[Singleton 94] Andy Singleton. *Genetic Programming with C++*. BYTE, pp 171–176, February 1994.

[Smith 80] Steven F. Smith. *A learning system based on genetic adaptive algorithms*. PhD thesis, University of Pittsburgh, 1980.

[Smith 91] Robert E. Smith, David E. Goldberg & Jeff A. Earickson. *SGA-C: A C-Language implementation of a simple genetic algorithm.* Technical report, TCGA report No. 91002, May 1991.

[Smith 97] Douglas J. Smith. *HDL Chip design, A practical guide for designing, synthesising and simulating ASICs and FPGAs using VHDL or Verilog.* Doone publications, 1997.

[Somerville 97] Ian Somerville. *Software Engineering.* Addison-Wesley, 5th edition, 1997.

[Soule 98] Terence Soule. *Code growth in genetic programming.* PhD thesis, University of Idaho, Moscow, Idaho, USA, 15 May 1998.

[Spector 95a] Lee Spector. *Evolving control structures with automatically defined macros.* In E. V. Siegel & J. R. Koza, (Eds.), Working Notes for the AAAI Symposium on Genetic Programming, pp 99–105, MIT, Cambridge, MA, USA, November 1995. AAAI.

[Spector 95b] Lee Spector & Adam Alpern. *Induction and Recapitulation of Deep Musical Structure.* In Proceedings of International Joint Conference on Artificial Intelligence, IJCAI'95 Workshop on Music and AI, Montreal, Quebec, Canada, 20-25 August 1995.

[Spector 99] Lee Spector, Howard Barnum, Herbert J. Bernstein & Nikhil Swami. *Finding a Better-than-Classical Quantum AND/OR Algorithm using Genetic Programming.* In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao & Ali Zalzala, (Eds.), Proceedings of the Congress on Evolutionary Computation, Vol. 3, pp 2239–2246, Mayflower Hotel, Washington D.C., USA, July 1999. IEEE Press.

[Spector 02] Lee Spector. *Push, PushGP, and Pushpop.* `http://hampshire.edu/lspector/push.html`, 2002.

[Spencer 94] Graham F. Spencer. *Automatic generation of programs for crawling and walking.* In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 15, pp 335–353. MIT Press, Cambridge, MA, USA, 1994.

[Spice 02] Spice. *The Spice homepage.* `http://bwrc.eecs.berkeley.edu:80/Classes/IcBook/SPICE/`, 2002.

[Spivey 92] Mike Spivey. *The Z Notation: A Reference Manual.* Prentice hall International, 2nd edition, 1992.

[Stallings 00] William Stallings. *Computer Organization and Architecture: Designing for Performance.* Prentice Hall, 5th edition, 2000.

[Stanhope 98] Stephen A. Stanhope & Jason M. Daida. *Genetic programming for automatic target classification and recognition in synthetic aperture radar imagery.* In V. William Porto, N. Saravanan, D. Waagen & A. E. Eiben, (Eds.), Evolutionary Programming VII: Proceedings of the Seventh Annual Conference on Evolutionary Programming, Vol. 1447 of *LNCS*, pp 735–744, Mission Valley Marriott, San Diego, CA, USA, March 1998. Springer-Verlag.

[Sterling 95] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake & C. V. Packer. *BEOWULF: A Parallel Workstation for Scientific Computation*. In Proceedings of the 24th International Conference on Parallel Processing, pp I:11–14, Oconomowoc, WI, August 1995.

[Stevens 98] Richard Stevens, Peter Brook, Ken Jackson & Stuart Arnold. *Systems Engineering Coping with Complexity.* Prentice Hall, 1998. ISBN: 0-13-0950858-8.

[Stiliadis 96] Dimitrios Stiliadis & Anujan Varma. *FAST: An FPGA-based simulation testbed for ATM networks.* Proceedings ICC'96, June 1996.

[Sulik 00] Daniel Sulik, Milan Vasilko, Daniela Durackova & P. Fuchs. *Design of a RISC microcontroller core in 48 Hours*. Unpublished paper, Bournemouth University, May 2000. `http://dec.bournemouth.ac.uk/drhw/publications/sulik-risc48hrs.pdf`, Embedded Systems Show 2000, London Olympia, United Kingdom.

[Sundararajan 00] Prasanna Sundararajan & Steven A. Guccione. *XVPI: A Portable Hardware / Software Interface for Virtex.* In Reconfigurable Technology: FPGAs for Computing and Applications II, Proc. SPIE 4212, pp 90–95, Bellingham, WA, November 2000. SPIE - The International Society for Optical Engineering.

[Synopsis 02] Synopsis. *Synopsis: System level design*. `http://www.systemc.org`, 2002.

[Syswerda 89] Gilbert Syswerda. *Uniform crossover in genetic algorithms*. In David J. Schaffer, (Ed.), Proceedings of the Third International Conference on Genetic Algorithms, pp 2–9, San Mateo, CA, May 1989. Morgan Kaufmann.

[Tacket 93] Walter Alden Tacket & Aviram Carmi. *SGPC: Simple Genetic Programming in C*. `ftp://cs.ucl.ac.uk/genetic/ftp.io.com/code`, 1993.

[Tackett 93] Walter Alden Tackett. *Genetic programming for feature discovery and image discrimination*. In Stephanie Forrest, (Ed.), Proceedings of the 5th international conference on genetic algorithms, ICGA-93, pp 303–309, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[Tackett 94] Walter Alden Tackett & Aviram Carmi. *The donut problem: scalability and generalization in genetic programming*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 7, pp 143–176. MIT Press, Cambridge, MA, USA, 1994.

[Tanomaru 93] Julio Tanomaru. *Evolving turing machines from examples*. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer & D. Snyers, (Eds.), Artificial Evolution, Vol. 1363 of *LNCS*, Nimes, France, October 1993. Springer-Verlag.

[Teller 94a] Astro Teller. *The evolution of mental models*. In Kenneth E. Kinnear, Jr., (Ed.), Advances in Genetic Programming, Chap. 9, pp 199–219. MIT Press, Cambridge, MA, USA, 1994.

[Teller 94b] Astro Teller. *Genetic programming, indexed memory, the halting problem, and other curiosities*. In Proceedings of the 7th Annual Florida Artificial Intelligence Research Symposium, pp 270–274, Pensacola, FL, USA, May 1994. IEEE Press.

[Teller 94c] Astro Teller. *Turing completeness in the language of genetic programming with indexed memory*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Vol. 1, pp 136–141, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Thompson 95] Adrian Thompson. *Evolving electronic robot controllers that exploit hardware resources*. In F. Morán, A. Moreno, J. J. Merelo & P. Chacón, (Eds.), Advances in Artificial life: Proceedings 3rd European Conference on Artificial Life (ECAL'95), Vol. 929 of *LNCS*, pp 640–656, Granada, Spain, June 1995. Springer-Verlag.

[Thompson 96] Adrian Thompson. *Silicon evolution*. In John R. Koza, David E. Goldberg, David B. Fogel & Rick L. Riolo, (Eds.), Genetic programming 1996: proceedings of the first annual conference, pp 444–452, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[Thompson 97] Adrian Thompson. *Artificial evolution in the physical world*. In Takashi Gomi, (Ed.), Evolutionary Robotics: From intelligent robots to artificial life (ER'97), pp 101–125, Canadian Embassy, 3-38 Akasaka 7-Chome, Minato-ku, Tokyo, Japan, April 1997. AAI Books.

[Thompson 99] Adrian Thompson & Paul Layzell. *Analysis of unconventional evolved electronics*. Communications of the ACM, vol. 42, no. 4, pp 71–79, April 1999.

[Thonemann 94] Ulrich Wilhelm Thonemann. *Finding improved simulated annealing schedules with genetic programming*. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Vol. 1, pp 391–395, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[Tommiska 96] Matti Tommiska & Jarkko Vuori. *Hardware implementation of GA*. In Jarmo T. Alander, (Ed.), Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA), Vaasa, Finland, August 1996.

[Tufte 99] Gunnar Tufte & Pauline C. Haddow. *Prototyping a GA pipeline for complete hardware evolution*. In A Stoica, D Keymeulen & J Lohn, (Eds.), Proceedings of the First NASA/DoD Workshop on Evolvable Hardware, pp 18–25. IEEE Computer Society, July 1999.

[Turing 50] Alan Turing. *Computing machinery and intelligence*. Mind, vol. 59, pp 433–460, 1950.

[Turner 02] Ken Turner. *World-wide Environment for Learning LOTOS (WELL)*. `http://www.cs.stir.ac.uk/~kjt/research/well/`, 2002.

[Turton 94] Brian C.H. Turton, Tughrul Arslan & David H. Horrocks. *A hardware architecture for a parallel genetic algorithm for image registration*. In

Proceedings of the IEE Colloquium of Genetic Algorithms in Image Processing and Vision, pp 11/1–11/6, October 1994.

[Turton 95] Brian C.H. Turton & T Arslan. *A parallel genetic VLSI architecture for combinatorial real-time applications - disc scheduling*. In In Proc. first IEE/IEEE International conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, pp 493–498, September 1995.

[VDM 96] VDM. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*. International Standard ISO/IEC 13817-1, December 1996.

[VDM 02] VDM. *Information on VDM*. `http://www.csr.ncl.ac.uk/vdm/`, 2002.

[Verilog 01] Verilog. *Verilog language reference manual IEEE 1364-2001*. ISBN 0-7381-2827-9, 2001.

[VHDL 93] VHDL. *VHDL*. IEEE Std 1076.1993, 1993.

[Vienna 02] Vienna. *The Vienna University of Economics Genetic Programming Kernel*. `http://sal.kachinatech.com/Z/3/VUEGPK.html`, 2002.

[Weinbrenner 02] Thomas Weinbrenner. *The Genetic Programming Kernel*. `http://ftp.eecs.umich.edu/people/daida/transfer/gpc/gpkernel_1.html`, 2002.

[Weinert 02] Klaus Weinert, Tobias Surmann & Jörn Mehnen. *Parallel surface reconstruction*. In Evelyne Lutton, James A. Foster, Julian Miller, Conor Ryan & Andrea G. B. Tettamanzi, (Eds.), Proceedings of the 4th European Conference on Genetic Programming, EuroGP'2002, Vol. 2278 of *LNCS*, pp 93–102, Kinsale, Ireland, April 2002. Springer-Verlag.

[Whigham 95] Peter A. Whigham. *Grammatically-based genetic programming*. In Justinian P. Rosca, (Ed.), Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, pp 33–41, Tahoe City, CA, USA, 9 July 1995.

[Whigham 99] Peter A. Whigham & Peter F. Crapper. *Time series modelling using genetic programming: an application to rainfall-runoff models*. In Lee Spector, William B. Langdon, Una-May O'Reilly & Peter J. Angeline, (Eds.), Advances in Genetic Programming 3, Chap. 5, pp 89–104. MIT Press, Cambridge, MA, USA, June 1999.

[White 94] Tony White. *GPEIST (The Genetic Programming Environment in Smalltalk)*. `ftp://ftp.mad-scientist.com/pub/genetic-programming/code/GPEIST.tar.Z`, 13 May 1994.

[Wirth 98] Niklaus Wirth. *Hardware compilation: Translating programs into circuits*. IEEE Computer, vol. 31, no. 6, pp 25–31, June 1998.

[Wolfram 84] Stephen Wolfram. *University and complexity in cellular automata*. Physica, vol. 10, pp 1–35, 1984.

[Wolfram 86] Stephen Wolfram. *Random sequence generation in cellular automata*. Adv. Appl. Math., vol. 7, pp 123–169, 1986.

[Wolpert 97] David H. Wolpert & William G. Macready. *No Free Lunch Theorems for Optimization*. IEEE Transactions on Evolutionary Computation, vol. 1, no. 1, pp 67–82, April 1997.

[Woodward 99] Andrew M. Woodward, Richard J. Gilbert & Douglas B. Kell. *Genetic programming as an analytical tool for non-linear dielectric spectroscopy*. Bioelectrochemistry and Bioenergetics, vol. 48, no. 2, pp 389–396, 1999.

[Xilinx 01a] Xilinx. *Pseudo random number generator*. `http://www.xilinx.com/xcell/xl35/xl35_44.pdf`, December 2001.

[Xilinx 01b] Xilinx. *Web site of Xilinx for FPGA data sheets*. `http://www.xilinx.com`, 2001. Last visited 15/June/2001.

[Yamaguchi 00] Yoshiki Yamaguchi, Akira Miyashita, Tsutomu Marutama & Tsutomu Hoshino. *A co-processor system with a Virtex FPGA for evolutionary computation*. In Reiner W. Hartenstein & Herbert Grunbacher, (Eds.), 10th International Conference on Field Programmable Logic and Applications (FPL2000), Vol. 1896 of *Lecture notes in Computer Science*, pp 240–249. Springer-Verlag, August 2000.

[Yeun 00] Yun Seog Yeun, Jun Chen Suh & Young-Soon Yang. *Function approximations by superimposing genetic programming trees:with applications to engineering problems*. Informaion Sciences, vol. 122, no. 2-4, pp 259–280, 2000.

[Yoshida 01] Norihoko Yoshida, Tomohiro Yasuoka, Toshiomi Moriki & Toshinko Shimokawa. *VLSI hardware design for genetic algorithms and its parallel and distributed extensions*. International Journal of Knowledge-Based Intelligent Engineering Systems, vol. 5, no. 1, pp 14–21, January 2001.

[Yu 99] Gwoing Tina Yu. *An analysis of the impact of functional programming techniques on genetic programming*. PhD thesis, University College, London, Gower Street, London, WC1E 6BT, United Kingdom, 1999.

[Zongker 96] Douglas Zongker & Bill Punch. *Lilgp 1.01 user's manual*. Technical report, Michigan State University, USA, 26 March 1996.

[Zongker 02] Douglas Zongker & Bill Punch. *Lilgp 1.01*. `ftp://garage.cps.msu.edu/pub/GA/lilgp`, 2002.

# Appendix A

# Attributes of the Experimental Problems

This appendix presents the attributes of the problems used in the experimental work in this thesis. The attributes are shown using the format developed in Chapter 2, Table 2.7 on page 32.

Table A.1 describes the regression problem from Chapter 4. Table A.2 describes the exclusive OR problem used in Chapter 4 and Chapter 5. Table A.3 describes the artificial ant problem from Chapter 5, and Table A.4 describes the Boolean even-6-parity problem from Chapter 5.

**Table A.1:** Simple regression attributes

| Category | Value | Comments |
|---|---|---|
| **EXTERNAL** | | |
| Problem category | Regression | Simple linear regression |
| Formal specification | Equation | $x = a + 2b$ |
| Number of fitness cases | 4 | |
| Language | FM | |
| **GP SPECIFIC** | | |
| Function set size | 8 | `add`$(\mathtt{R}_n, \mathtt{R}_m)$, `sub`$(\mathtt{R}_n, \mathtt{R}_m)$, `shl`$(\mathtt{R}_n)$, `shr`$(\mathtt{R}_n)$, `nop`, `halt`$(\mathtt{R}_n)$, `ldim`$(\mathtt{R}_n, \mathtt{K}_m)$, `jmpifz`$(\mathtt{R}_n, \mathtt{R}_m)$ |
| Function set abstraction | Low | |
| Terminal set size (T) | 4 | |
| Automatic feature discovery | None | |
| Memory | None | |
| Program representation | Linear | |
| Polymorphism and data typing | None | |
| Representation mapping | Direct | |
| Crossover operator | 67% | |
| | Xmod | True. Truncating crossover. |
| Mutation operator | 12.5% | |
| | Mmod | False |
| Reproduction operator | 21.5% | |
| | Rmod | False |
| Creation method | Uniform | |
| Seeding | None | |
| Population size | 16 | |
| Generations | 511 | |
| Selection method | Tournament | |
| Generational method | Steady state | |
| **RESULTS** | | |
| Program size | 4 | |
| Successful | True | |
| Effort | | |
| Wall clock time | | |
| **IMPLEMENTATION** | | |
| Class of GP platform | Hard | |
| Fitness function implementation | Intrinsic | |
| Model of parallelisation | MasterSlave | |

**Table A.2:** Exclusive OR problem attributes

| Category | Value | Comments |
|---|---|---|
| **EXTERNAL** | | |
| Problem category | Regression | Boolean expression regression |
| Formal specification | Equation | $x = a \oplus b$ |
| Number of fitness cases | 4 | |
| Computability | FM | |
| **GP SPECIFIC** | | |
| Function set size | 4 | $\texttt{AND}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{OR}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{NOR}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{NAND}(\texttt{R}_n, \texttt{R}_m)$ |
| Function set abstraction | Low | |
| Terminal set size (T) | 4 | 4 registers. |
| Automatic feature discovery | None | |
| Memory | None | |
| Program representation | Linear | |
| Polymorphism and data typing | None | |
| Representation mapping | Direct | |
| Crossover operator | 66% | |
| | Xmod | True - Truncating crossover |
| Mutation operator | 12.5% | |
| | Mmod | False |
| Reproduction operator | 21.5% | |
| | Rmod | False |
| Creation method | Uniform | |
| Seeding | None | |
| Population size | 16 | |
| Generations. | 511 | |
| Selection method | Tournament | |
| Generational method | Steady state | |
| **RESULTS** | | |
| Program size | 4 | |
| Successful | True | |
| Effort | | |
| Wall clock time | | |
| **IMPLEMENTATION** | | |
| Class of GP platform | Hard | |
| Fitness function implementation | Intrinsic | |
| Model of parallelisation | MasterSlave | |

**Table A.3:** Santa Fe Artificial Ant problem attributes

| Category | Value | Comments |
|---|---|---|
| **EXTERNAL** | | |
| Problem category | Control | |
| Formal specification | English | A description of how the ant should behave. |
| Number of fitness cases | 4 | |
| Computability | FM | |
| **GP SPECIFIC** | | |
| Function set size | 4 | `IF_FOOD`($T_a$, $T_b$), `PROGN`($T_a$, $T_b$) |
| Function set abstraction | Low | |
| Terminal set size (T) | 4 | `MOVE`,`LEFT`,`RIGHT`,`NOP` |
| Automatic feature discovery | None | |
| Memory | None | |
| Program representation | Linear | |
| Polymorphism and data typing | None | |
| Representation mapping | Direct | |
| Crossover operator | 66% | |
| | Xmod | True - Truncating crossover |
| Mutation operator | 12.5% | |
| | Mmod | False |
| Reproduction operator | 21.5% | |
| | Rmod | False |
| Creation method | Uniform | |
| Seeding | None | |
| Population size | 1024 | |
| Generations. | 32 | |
| Selection method | Tournament | |
| Generational method | Steady state | |
| **RESULTS** | | |
| Program size | 4 | |
| Successful | True | |
| Effort | | |
| Wall clock time | | |
| **IMPLEMENTATION** | | |
| Class of GP platform | Hard | |
| Fitness function implementation | Intrinsic | |
| Model of parallelisation | MasterSlave | |

**Table A.4:** Boolean even-6-parity problem attributes

| Category | Value | Comments |
|---|---|---|
| **EXTERNAL** | | |
| Problem category | Regression | Boolean expression regression |
| Formal specification | Equation | |
| Number of fitness cases | 4 | |
| Computability | FM | |
| **GP SPECIFIC** | | |
| Function set size | 4 | $\texttt{AND}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{OR}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{NOT}(\texttt{R}_n, \texttt{R}_m)$, $\texttt{XOR}(\texttt{R}_n, \texttt{R}_m)$ |
| Function set abstraction | Low | |
| Terminal set size (T) | 16 | 16 registers. The 6 input values were written to registers 0-5. The result was read from register 0. |
| Automatic feature discovery | None | |
| Memory | None | |
| Program representation | Linear | |
| Polymorphism and data typing | None | |
| Representation mapping | Direct | |
| Crossover operator | 66% | |
| | Xmod | True - Truncating crossover |
| Mutation operator | 12.5% | |
| | Mmod | False |
| Reproduction operator | 21.5% | |
| | Rmod | False |
| Creation method | Uniform | |
| Seeding | None | |
| Population size | 2048 | |
| Generations | 64 | |
| Selection method | Tournament | |
| Generational method | Steady state | |
| **RESULTS** | | |
| Program size | 171 | Mean size from 500 runs |
| Successful | True | |
| Effort | | |
| Wall clock time | | |
| **IMPLEMENTATION** | | |
| Class of GP platform | Hard | |
| Fitness function implementation | Intrinsic | |
| Model of parallelisation | MasterSlave | |

# Appendix B

# Taxonomy Data

## B.1 Data Used to Construct the Taxonomy of Genetic Programming

This section presents a summary of the data that was collected in order to construct the taxonomy of Genetic Programming. This data was collected and placed into a MySQL relational data base. To reduce the opportunity for errors, reports from the database were run to provide the data in Chapter 2. The reports were run using a Perl script which then used gnuplot to format the graphs. This table was created using Perl which formatted a LYX table for automatic inclusion into this thesis.

| | Category | Description | Pop | Gen | Size | Reference |
|---|---|---|---|---|---|---|
| 1 | Algorithms | Automatic Programming of a Randomizer | 500 | 51 | 87 | [Koza 92, pp 396–408] |
| 2 | Algorithms | Boolean 6multiplexer | 500 | 51 | 12 | [Koza 92, pp 187–188] |
| 3 | Algorithms | Boolean 11 Multiplexer | 500 | 51 | 37 | [Koza 92, pp 169-186] |
| 4 | Algorithms | Bracket problem | 10000 | 50 | 88 | [Langdon 98a, pp 143–148] |
| 5 | Algorithms | Directed acyclic graphs | – | – | – | [Handley 94b] |
| 6 | Algorithms | Discovering learning rules for neural networks | 1000 | 1000 | – | [Radi 99] |
| 7 | Algorithms | Dyck language | 10000 | 50 | 32 | [Langdon 98a, pp 149–154] |
| 8 | Algorithms | even 5 parity | 16000 | 51 | 112 | [Koza 94, pp 162] |
| 9 | Algorithms | Even 5 parity (with ADF) | 16000 | 51 | 51 | [Koza 94, pp 162] |
| 10 | Algorithms | Evolution of schedule for simulated annealing | – | – | – | [Thonemann 94] |
| 11 | Algorithms | Evolving a list | 10000 | 100 | 109 | [Langdon 98a, pp 123–142] |
| 12 | Algorithms | Evolving a queue | 10000 | 50 | 16 | [Langdon 98a, pp 81–122] |
| 13 | Algorithms | Evolving a sort | 1000 | 49 | 42 | [Kinnear, Jr. 93] |
| 14 | Algorithms | Evolving a sort using FPGAs | 1000 | 51 | – | [Koza 97b] |
| 15 | Algorithms | Evolving a stack | 1000 | 101 | 16 | [Langdon 98a, pp 61–80] |

| | Category | Description | Pop | Gen | Size | Reference |
|---|---|---|---|---|---|---|
| 16 | Algorithms | Evolving Neural Network Structures by means of Gen | 1000 | 200 | – | [Golubski 99] |
| 17 | Algorithms | Function approximation by superimposing genetic programming trees: with applications to engineering problems | 2000 | 40 | – | [Yeun 00] |
| 18 | Algorithms | Learning by adopting representations | – | – | – | [Rosca 94] |
| 19 | Algorithms | Minimal sorting network problem with GPPS 1.0 | 640000 | 1001 | 241 | [Koza 99c, pp 335–339] |
| 20 | Algorithms | Reasoning about proofs | – | – | – | [Nordin 99b] |
| 21 | Algorithms | Regression using stack based GP | 200 | 21 | – | [Perkis 94] |
| 22 | Algorithms | Reverse polish expression | 10000 | 100 | 38 | [Langdon 98a, pp 154–160] |
| 23 | Algorithms | Sorting network | 500 | 51 | – | [Koza 99c, pp 335–339] |
| 24 | Algorithms | Stack fi lters | – | – | – | [Oakley 94] |
| 25 | Art | Composing 16th century counterpoint | – | – | – | [Polito 97] |
| 26 | Art | Genetic art in virtual reality | – | – | – | [Das 94] |
| 27 | Art | Grammar based genetic programming technique applied to music generation | – | – | – | [Putnam 94] |
| 28 | Art | IFS generation - fractal generation | 50 | 30 | 22 | [Sarafopoulos 99] |
| 29 | Art | Induction and recapitulation | 250 | 50 | 221 | [Spector 95b] |
| 30 | Art | Interactive evolution of equations and images | – | – | – | [Sims 91] |
| 31 | Classifi cation | 2 boxes | 4000 | 51 | 17 | [Koza 94, pp 57] |
| 32 | Classifi cation | Autonomous document classifi -cation for business | – | – | – | [Clack 97a] |
| 33 | Classifi cation | Building queries for information retrieval | – | – | – | [Kraft 94] |
| 34 | Classifi cation | Classify Swedish nouns | 4000 | 100 | 50 | [Nordin 94] |
| 35 | Classifi cation | Classifying protein sequences | 320000 | 51 | – | [Koza 98a] |
| 36 | Classifi cation | Competitively evolving decision trees for natural language processing | 900 | 500 | 18 | [Siegel 94] |
| 37 | Classifi cation | Cracking and co-evolving randomisers | 2048 | 100 | – | [Jannink 94] |
| 38 | Classifi cation | Credit approval | 1000 | 40 | – | [Eggermont 99] |
| 39 | Classifi cation | Credit card problem | 600 | 80 | – | [Ahluwalia 98] |
| 40 | Classifi cation | Detection of the dipicolinic acid biomarker in Bacillus spores using Curie-point pyrolysis mass spectrometry and genetic programming | – | – | – | [Goodacre 00] |
| 41 | Classifi cation | Digit recognition | 512000 | 201 | – | [Koza 99c, pp 253] |

| | Category | Description | Pop | Gen | Size | Reference |
|---|---|---|---|---|---|---|
| 42 | Classifi cation | Donut problem | – | – | – | [Tackett 94] |
| 43 | Classifi cation | Evolving Fuzzy Rule Based Classifi ers with GA-P | – | – | 67 | [Garcia 99] |
| 44 | Classifi cation | Extracting low contrast curvilinear features from SAR images of arctic ice | 357 | 30 | – | [Daida 96] |
| 45 | Classifi cation | Feature discovery | – | – | – | [Tackett 93] |
| 46 | Classifi cation | Fitting chaotic data | 5000 | 101 | 102 | [Oakley 94] |
| 47 | Classifi cation | Flushes and four-of-a-kinds in a Pinochle Deck | 4000 | 51 | 63 | [Koza 94, pp 417] |
| 48 | Classifi cation | Flushes and four-of-a-kinds in a Pinochle Deck (with ADF) | 4000 | 51 | 54 | [Koza 94, pp 417] |
| 49 | Classifi cation | Genetic programming as an analytical tool for non-linear dielectric spectroscopy | 50000 | 200 | – | [Woodward 99] |
| 50 | Classifi cation | GP for analysing metabolic data - classifi cation | 37500 | 500 | – | [Gilbert 99] |
| 51 | Classifi cation | GP for automatic target classifi - cation in radar | 250 | 100 | 11 | [Stanhope 98] |
| 52 | Classifi cation | Iris detection | 600 | 120 | – | [Ahluwalia 98] |
| 53 | Classifi cation | Medical data mining using evolutionary computation (thyroid problem) | 5000 | 250 | – | [Brameier 01] |
| 54 | Classifi cation | Multi population GP applied to burn diagnosing | 12500 | 60 | 43 | [de Vega 00] |
| 55 | Classifi cation | News story classifi cation by Dow Jones | – | – | – | [Masand 94] |
| 56 | Classifi cation | Prediction of secondary structure of proteins | – | – | – | [Handley 93] |
| 57 | Classifi cation | Prediction of transmembrane domains in proteins | 4000 | 21 | 71 | [Koza 94] |
| 58 | Classifi cation | Prediction of transmembrane domains in proteins (with ADF) | 4000 | 21 | 122 | [Koza 94] |
| 59 | Classifi cation | Quintic polynomial | 4000 | 51 | 69 | [Koza 94, pp 118] |
| 60 | Classifi cation | Ship detectors | 5000 | 10 | 17 | [Howard 99] |
| 61 | Classifi cation | Sonar classifi cation | 60 | 50 | 150 | [Iba 94] |
| 62 | Classifi cation | Speaker identifi cation | 10000 | 2000 | 256 | [Conrads 98] |
| 63 | Classifi cation | Speaker identifi cation (task level) | 400 | 50 | – | [Kantschik 99] |
| 64 | Classifi cation | Text classifi cation | 100 | 20 | – | [Masand 94] |
| 65 | Classifi cation | The detection of caffeine in a variety of beverages using Curie-point pyrolysis mass spectrometry and genetic programming | 25000 | 250 | 16 | [Goodacre 99] |
| 66 | Classifi cation | The donut problem | 1024 | 50 | – | [Tackett 94] |

| | Category | Description | Pop | Gen | Size | Reference |
|---|---|---|---|---|---|---|
| 67 | Classifi cation | Transmembrane segment identifi cation with architecture altering operations | – | – | – | [Koza 99c, pp 271] |
| 68 | Classifi cation | Upgrading rules for OCR | 5000 | 100 | – | [Andre 94] |
| 69 | Control | Artifi cial Ant | 500 | 51 | 18 | [Koza 92, pp 147–162] |
| 70 | Control | Artifi cial ant 13x13 grid | 4000 | 51 | 90 | [Koza 94, pp 349] |
| 71 | Control | Artifi cial ant 13x13 grid (with ADF) | 4000 | 51 | 71 | [Koza 94, pp 349] |
| 72 | Control | Box moving Robot | 500 | 51 | 207 | [Koza 92, pp 381–390] |
| 73 | Control | Broom balancing | 500 | 51 | 21 | [Koza 92, pp 289–303] |
| 74 | Control | Broom balancing | 500 | 51 | – | [Koza 92, pp 288–307] |
| 75 | Control | Bumble bee 25 fbwers | 4000 | 51 | 452 | [Koza 94, pp 275] |
| 76 | Control | Bumble bee 25 fbwers (with ADF) | 4000 | 51 | 245 | [Koza 94, pp 275] |
| 77 | Control | Cart centering | 500 | 51 | 55 | [Koza 92, pp 122–147] |
| 78 | Control | Cart centering | 128000 | 201 | – | [Koza 99c, pp 305–308] |
| 79 | Control | Cart Centering (with ADF) | 128000 | 501 | 486 | [Koza 99c, pp 305–308] |
| 80 | Control | Channel equalisation | – | – | – | [Esparcia-Alcázar 96] |
| 81 | Control | Control of robots | – | – | – | [Ghanea-Hercock 94] |
| 82 | Control | Corridor following | – | – | – | [Reynolds 94a] |
| 83 | Control | Evolution of cooperation | – | – | – | [Rush 94] |
| 84 | Control | Evolution of herding behaviour | – | – | – | [Reynolds 92] |
| 85 | Control | Evolving controllers for autonomous agents using GP | 100 | 51 | – | [Silva 99] |
| 86 | Control | Food Place Foraging | 1000 | 61 | 67 | [Koza 92, pp 329–343] |
| 87 | Control | Generating adaptive behaviour for a real robot using function regression within genetic programming | – | – | – | [Banzhaf 97] |
| 88 | Control | Generation of programs for crawling and walking | 1000 | 100 | – | [Spencer 94] |
| 89 | Control | Genetic Programming for service creation in Intelligent Networks | 200 | 200 | 18 | [Martin 00] |
| 90 | Control | Goal keeper control strategy | 200 | 2000 | 32 | [Adorni 99] |
| 91 | Control | Lawnmower | 1000 | 51 | 280 | [Koza 94, pp 225] |
| 92 | Control | Lawnmower (with ADF) | 1000 | 51 | 76 | [Koza 94, pp 225] |
| 93 | Control | Multi-agent problem | 64000 | 901 | – | [Koza 99c, pp 241] |
| 94 | Control | Noise cancellation | – | – | – | [Esparcia-Alcázar 96] |
| 95 | Control | Obstacle avoiding robot | 4000 | 51 | 336 | [Koza 94, pp 365] |
| 96 | Control | Obstacle avoiding robot (with ADF) | 4000 | 51 | 101 | [Koza 94, pp 365] |
| 97 | Control | Reactive and memory based GP for robot controllers | 100 | – | – | [Andersson 99] |
| 98 | Control | Robot controller | – | – | – | [Koza 99c, pp 332–334] |
| 99 | Control | Robot controller problem with GPPS 1.0 | 640000 | 1001 | 260 | [Koza 99c, pp 332-334] |

| | Category | Description | Pop | Gen | Size | Reference |
|---|---|---|---|---|---|---|
| 100 | Control | Robot localisation | 1000 | 5000 | – | [Ebner 99] |
| 101 | Control | Soccer softbot teams | 128 | – | – | [Luke 98a] |
| 102 | Control | Task Prioritisation (PacMan) | 500 | 51 | 115 | [Koza 92, pp 344–355] |
| 103 | Control | The pursuit problem | – | 2000 | – | [Haynes 95] |
| 104 | Control | Truck Backer upper problem | 1000 | 51 | 108 | [Koza 92, pp 307–314] |
| 105 | Control | Wall Following Robot | 1000 | 51 | 145 | [Koza 92, pp 360–380] |
| 106 | Design | An evolutionary approach to automatic generation of VHDL code for low-power digital filters | 1000 | 10000 | – | [Erba 01] |
| 107 | Design | Automatic synthesis of a wire antenna using genetic programming | 500000 | – | – | [Koza 00b] |
| 108 | Design | Design of explicitly or implicitly parallel low resolution character recognition algorithms | 1000 | 1000 | – | [Adorni 98] |
| 109 | Design | Discovering simple fault-tolerant routing rules using genetic programming | 500 | 31 | 25 | [Kirkwood 97] |
| 110 | Design | Electrical filter circuits - reinventing circuits | 1950000 | – | 12 | [Koza 99b] |
| 111 | Design | Evolving signal processing algorithms by genetic programming | 250 | 100 | 69 | [Sharman 95] |
| 112 | Design | Evolving Turing machines from examples | – | – | – | [Tanomaru 93] |
| 113 | Design | Gate-level synthesis of boolean functions using binary multiplexers | 600 | 700 | – | [Hernández-Aguirre 00] |
| 114 | Design | Genetic programming approaches for minimum cost topology optimisation of optical telecommunications networks | 100 | 300 | 323 | [Aiyarak 97] |
| 115 | Design | High gain op-amp circuit | 640000 | 101 | 46 | [Koza 97a] |
| 116 | Design | Jetliner model evolution | 10 | 50 | 221 | [Nguyen 94] |
| 117 | Design | Low pass filter (with ADF) | 320000 | 501 | 27 | [Koza 99c, pp 455–483] |
| 118 | Design | Low-pass filter | 500 | 51 | – | [Koza 99c, pp 455–483] |
| 119 | Design | Low-pass filter (using architecture altering operations) | 500 | 51 | – | [Koza 99c, pp 561–567] |
| 120 | Design | Parallel surface reconstruction | 300 | 2000 | – | [Weinert 02] |
| 121 | Regression | 2 boxes (with ADF) | 4000 | 51 | 33 | [Koza 94, pp 57] |
| 122 | Regression | A genetic programming approach for robust language interpretation | 32 | 4 | – | [Rosé 99] |
| 123 | Regression | Differential Equations | 500 | 51 | 14 | [Koza 92, pp 264–288] |

| | Category | Description | Pop | Gen | Size | Reference |
|---|---|---|---|---|---|---|
| 124 | Regression | Discover a cellular encoding of a neural network for the even-11-parity problem | 1024 | 50 | – | [Gruau 94] |
| 125 | Regression | Discovery of Kepler's Third Law | 500 | 51 | 6 | [Koza 92, pp 255–258] |
| 126 | Regression | Discovery of trigonometric identities | 500 | 51 | 9 | [Koza 92, pp 238–242] |
| 127 | Regression | Double auction market strategies | – | – | – | [Andrews 94] |
| 128 | Regression | Econometric modelling | 500 | 51 | 31 | [Koza 92, pp 247–255] |
| 129 | Regression | Evolution of obstacle avoidance behaviour | 1000 | 50 | 32 | [Reynolds 94b] |
| 130 | Regression | Evolving a sorting network | 500 | 100 | 13 | [Ryan 94] |
| 131 | Regression | Fibonacci Sequence | 600000 | 501 | 70 | [Koza 99c, pp 297–304] |
| 132 | Regression | Four sines | 4000 | 51 | 86 | [Koza 94, pp 144] |
| 133 | Regression | Four sines (with ADF) | 4000 | 51 | 94 | [Koza 94, pp 144] |
| 134 | Regression | GP for combining neural networks for drug discovery | 500 | 50 | – | [Langdon 02b] |
| 135 | Regression | Horse race prediction | – | – | – | [Perry 94] |
| 136 | Regression | Identification of macro mechanical models | – | – | – | [Schoenauer 96] |
| 137 | Regression | Inflation rate modelling | 250 | 1000 | – | [Chen 98] |
| 138 | Regression | Knowledge intensive genetic discovery in foreign exchange markets | 100 | 100 | 11 | [Bhattacharyya 02] |
| 139 | Regression | Measuring facial emotional expressions using GP | 750 | 70 | – | [Loizides 01] |
| 140 | Regression | Missile countermeasures | 500 | 50 | – | [Moore 98] |
| 141 | Regression | Parametric coding - lens profile | 100 | 5277 | – | [Racine 99] |
| 142 | Regression | Playing Othello | – | – | – | [Eskin 99] |
| 143 | Regression | Playing tetris | 1000 | 171 | – | [Siegel 96] |
| 144 | Regression | Predicting journey times on motorways | 2000 | – | – | [Howard 02] |
| 145 | Regression | Predicting the flow of a typical urban basin | 2000 | – | – | [Dorado 02] |
| 146 | Regression | Quartic polynomial using grammatical evolution | 500 | 51 | – | [Ryan 98] |
| 147 | Regression | Quintic polynomial (with ADF) | 4000 | 51 | 64 | [Koza 94, pp 118] |
| 148 | Regression | Regular languages | – | – | – | [Dunay 94] |
| 149 | Regression | Stack filters | 2000 | 101 | 100 | [Oakley 94] |
| 150 | Regression | Symbolic Differentiation | 500 | 51 | 41 | [Koza 92, pp 262–264] |
| 151 | Regression | Symbolic Integration | 500 | 51 | 12 | [Koza 92, pp 258–262] |
| 152 | Regression | Symbolic regression (x^4+x^3+x^2+x) | 500 | 51 | 20 | [Koza 92, pp 162–169] |
| 153 | Regression | Symbolic regression with constant creation | 500 | 51 | 23 | [Koza 92, pp 242–245] |

| | Category | Description | Pop | Gen | Size | Reference |
|---|---|---|---|---|---|---|
| 154 | Regression | The acquisition of double market auction strategies | 300 | 50 | – | [Andrews 94] |
| 155 | Regression | The automatic generation of plans for a mobile robot | 2000 | 51 | 122 | [Handley 94a] |
| 156 | Regression | The tartarus environment | 800 | 100 | 185 | [Teller 94a] |
| 157 | Regression | Time series modelling | 1000 | 50 | – | [Whigham 99] |
| 158 | Regression | Two term | 4000 | 51 | 60 | [Koza 94, pp 151] |

# Appendix C

# Experimental Setup

This appendix describes the experimental setup used in for experimental work in Chapters 4, 5 and 6. It describes the platforms used for the experiments and associated software. This appendix also gives the configuration commands and the commands to execute the tools where relevant. This information is included to assist others to replicate some or all of the work reported in this thesis. URLs are given for the various packages and vendors.

## C.1 Experimental Platforms

### C.1.1 Hardware

The work in Chapter 4 was performed using a 200 MHz AMD K6 PC with 128 KiB of memory and a 500 MHz Pentium II PC, with 384 MiB of memory. The work in Chapters 5 and 6 was performed using a PC, consisting of a 1.4 GHz Athlon processor with 750 MiB of *Double Data Rate* (DDR) RAM. An extra 256 MiB was borrowed for the Boolean even-6-parity problem, bringing the total to 1 GiB DDR RAM.

### C.1.2 RC1000 FPGA Evaluation Board

This board, available from Celoxica Ltd consists of:

Xilinx XCV2000e, Virtex-E series FPGA, in a bg560 package.

4 banks x 2 MiB *Static Random Access Memory* (SRAM)

An Intel 21152 PCI-PCI bridge that connects the host bus to the RC1000.

A PLX PCI9080 PCI controller chip.

A Cypress IDC2061A programmable clock generator.

For full details of the hardware, refer to the RC1000 Hardware Reference Manual [Celoxica Ltd 01c]. For details of the APIs used to configure the board, refer to the Software Reference Manual [Celoxica Ltd 01d] and the Functional Reference Manual [Celoxica Ltd 01b].

### C.1.3 Software

The software environment used to run the Celoxica DK1 and Xilinx Alliance software was Microsoft Windows 98 and Microsoft NT 4.0 for the work described in chapter 4 and Microsoft Windows 2000 professional for the work described in chapters 5 and 6.

The software environment for running the psim PowerPC simulation was Mandrake Linux, version 8.0.

## C.2 Software Tools

### C.2.1 Celoxica DK1

Version 3.0 of DK1 was used. Service pack 1 was installed for the work in chapter 5.

### C.2.2 Celoxica RC1000 Drivers and Utilities

Version 1.32 of the Windows 98 and Windows 2000 drivers were used `http://www.celoxica.com`.

### C.2.3 Xilinx Alliance

Version 3.1 patch level 8. `http://www.xilinx.com/`

### C.2.4 Diehard

Version 19950110 from `http://stat.fsu.edu/geo/`

### C.2.5 PowerPC Cross Compiler and Simulator

For the PowerPC simulation, the code was compiled using the GNU C compiler, version 2.95.2. This was obtained as part of the tpmp package, which contained the following package files:

tpmp2-binutils-2.9.1-1.i386.rpm

tpmp2-gcc-2.95.2-1.i386.rpm

tpmp2-glibc-2.1.3-1.noarch.rpm

The PowerPC simulator is delivered as part of the GNU debugger (gdb) version 5.1.1. It was configured and built using:

***> export CC=gcc; ./configure –target=ppc-unknown-eabi –enable-sim-ppc***

***> make***

***> make install***

The code was compiled for the PowerPC target using the following commands:

***> ppc-linux-gcc -c hwgp.c -I. -I/usr/include -DPSIM -D<problem-type>***

***> ppc-linux-ld -o hwgp-ppc hwgp-ppc***

where problem-type is one of: `ANT, XOR, REGRESSION, BOOLPARITY`.

The following command was used to obtain the cycle counts from the PowerPC simulator:

***> ppc-unknown-eabi-run -I hwgp-ppc***

The standard psim program needed to be modified to allow it to record more than $4 \times 10^9$ cycles. This was because the standard program used variables of type `unsigned long` as counters. Using gcc version 2.95 on an Intel 32 bit processor these are 32 bit integers and can represent values up to $2^{32} - 1$ or approximately $4 \times 10^9$. The counters were replaced with type `unsigned long long`(1) which, again using gcc 2.95, are 64 bit quantities and can represent integers up to $2^{64} - 1$ or approximately $1.8 \times 10^{19}$. The corresponding `printf()` formats were changed to `%ll`.

### C.2.6 Relational Database

MySQL (version 3.23) was used to store the GP attribute data. The Perl API was used to insert the data and perform the queries that produced the results in Chapter 2. A single table stored all the taxonomy data.

### C.2.7 Graph Preparation Tools

All the graphs were created using gnuplot version 3.7. `http://www.ucc.ie/gnuplot/`

## C.3 Other Tools

The LyX document preparation system (`http://www.lyx.org/`) and LaTeX $2_\varepsilon$ were used to prepare and print this thesis.

(1)The `long long` type specifi er has been an unoffi cial extension to ANSI-C for many years and was offi cially incorporated into the C language as part of ISO/IEC 9899-1999, Programming Languages – C.

# Appendix D

# Random Number Generator Test Results

This appendix contains the results of running the Diehard tests for all the RNGs in Chapter 6. Max score represents the case where an RNG fails all the tests. The test method is described in Section 6.3.3 on page 124. Each test is described in detail in the file TESTS.TXT which is delivered as part of the Diehard test suite.

The standard test programs – ***diehard*** and ***diequick*** – generate a lot of commentary while they are running, making it hard to automatically process the results. To allow the automatic collection of test results, a version of the ***diequick*** program – called ***diequiet*** – was created that printed only the p-values to stdout. These were then processed by a Perl script to generate the final test results.

**Table D.1:** Diehard test results for all RNGs examined.

| | Max score | LFSR | EQG | 32LFSR | IDCA | 32CA | True | Mother | Sequential |
|---|---|---|---|---|---|---|---|---|---|
| Birthday | 36 | 36 | 8 | 2 | 0 | 8 | 0 | 0 | 36 |
| Overlapping permutation | 8 | 8 | 0 | 4 | 8 | 8 | 0 | 0 | 8 |
| Binary Rank 32x32 | 8 | 8 | 2 | 8 | 2 | 6 | 0 | 0 | 8 |
| Binary Rank 6x | 104 | 104 | 40 | 8 | 140 | 70 | 4 | 6 | 104 |
| Bitstream | 80 | 80 | 0 | 0 | 80 | 80 | 4 | 0 | 80 |
| Overlapping pairs tests | 328 | 328 | 188 | 94 | 328 | 320 | 6 | 2 | 328 |
| Count the ones (stream) | 8 | 8 | 8 | 8 | 8 | 8 | 0 | 0 | 8 |
| Count the ones (specifi c) | 100 | 100 | 42 | 30 | 100 | 100 | 2 | 4 | 100 |
| Parking Lot | 44 | 4 | 0 | 0 | 4 | 2 | 0 | 0 | 44 |
| Minimum Distance | 4 | 4 | 0 | 4 | 4 | 4 | 0 | 0 | 4 |
| 3D spheres | 84 | 4 | 0 | 2 | 4 | 2 | 4 | 4 | 84 |
| Squeeze | 4 | 4 | 0 | 0 | 4 | 4 | 0 | 0 | 4 |
| Overlapping Sums | 44 | 44 | 0 | 0 | 6 | 0 | 2 | 2 | 44 |
| Runs | 16 | 16 | 0 | 2 | 16 | 8 | 0 | 2 | 16 |
| Craps | 8 | 8 | 0 | 0 | 8 | 12 | 0 | 0 | 8 |
| Total | 876 | 756 | 288 | 162 | 676 | 640 | 22 | 20 | 876 |