Table of Contents

Abstract 2
Acknowledgements 4
Table of Contents 5
List of Figures 10
List of Tables 13
Chapter 1. Introduction 14
1.1. Motivation of the Research 16
1.2. Selection of Case Studies 18
1.3. Contributions of the Research 20
1.4. Structure of the Thesis 21
Chapter 2. Background Issues in the Design Process 24
2.1. Overview 24
2.2. Design Problems 25
2.2.1 Combinational Optimization 25
2.2.2 Historical Methods 26
2.2.3 Design Framework 27
2.2.4 Design Specifications 29
2.2.5 Time Complexity of Design Problems 31
2.2.6 Target Technology 33
2.3. Multi-Objective Optimization 34
2.3.1 Measuring Performance of Multi-Objective Optimisers 34
2.3.2 An Improved Comparison Method 35
2.3.3 Case Studies 36
2.4. Evolutionary Algorithms 37
2.4.1 Overview 37

2.4.2 Nomenclature 38
2.4.3 Process 39
2.4.4 Convergence 40
2.4.5 Implementations 40
2.5. Chapter Summary 41
Chapter 3. System Design Framework 43
3.1. Chapter Overview 43
3.2. Introduction 44
3.3. Design Optimization 46
3.4. Motivating Examples 47
3.5. Design Framework 48
3.5.1 Overview 48
3.5.2 Design Specification 52
3.5.3 STR Example 69
3.5.4 SAD Example 72
3.5.5 EKF Example 75
3.6. Chapter Summary 77
Chapter 4. Genetic Algorithms 78
4.1. Overview 78
4.2. Algorithm Flow 79
4.3. Genetic Operators in Genetic Algorithms 82
4.3.1 Selection 82
4.3.2 Crossover 84
4.3.3 Mutation 87
4.4. Issues in Genetic Algorithms 88
4.4.1 Mating Selection 88
4.4.2 Selecting Search Parameters 89
4.4.3 Controlling the Balance of Exploration and Exploitation 89

4.5. Chapter Summary 90

Chapter 5. Adaptive Speciation Genetic Algorithm (ASGA) 92

5.1. Overview 92

5.2. Selection 94

5.2.1 Development Goals 94

5.2.2 Implementation 95

5.2.3 Extensions 97

5.2.4 Constraint Handling 99

5.3. Crossover 100

5.3.1 Tagging Individuals 100

5.3.2 Mate Selection 102

5.3.3 Ancestry 104

5.3.4 Species Management 105

5.3.5 Implementation 108

5.4. Mutation 109

5.4.1 Adaptive Strategies 110

5.5. Encoding 111

5.6. Chapter Summary 112

Chapter 6. Algorithm Evaluation 114

6.1. Overview 114

6.2. Test Problems 115

6.2.1 70 Ones Problem 115

6.2.2 30 Ones Problem 116

6.2.3 Allele Alphabet Size Problem 116

6.2.4 Iterated Prisoner's Dilemma Problem 117

6.2.5 De Jong's F1 Function 118

6.2.6 De Jong's F4 Function 118

6.2.7 Branin RCOS Function 119

6.2.8 Shubert 120
6.2.9 Michalewicz Sine Function 121
6.2.10 Multi-objective Quadratic Problem 122
6.3. Experimental Design 123
6.4. Results 126
6.5. Chapter Summary 139
Chapter 7. Algorithm Comparison 140
7.1. Overview 140
7.2. Methodology for Comparisons of Optimisation Algorithms 141
7.2.1 Problems in Comparison 141
7.2.2 Background 142
7.2.3 Experiment Design 144
7.2.4 Approach 145
7.2.5 Algorithms used in Comparison 146
7.3. Comparisons on a known problem 149
7.3.1 Multi-objective d-dimensional 0/1 Knapsack Problem 149
7.3.2 Results 150
7.4. Comparisons on Design Case Studies 152
7.4.1 Self Tuning Regulator 152
7.4.2 Sum of Absolute Differences 156
7.4.3 The Extended Kalman Filter 158
7.5. Hybrid-ASGA 161
7.6. Chapter Summary 163
Chapter 8. Conclusions and Future Work 165
8.1. Conclusions 165
8.1.1 Design Problems 165
8.1.2 Genetic Algorithms 166
8.1.3 Comparison with other Algorithms 167

8.2. Future Work 168

References 170

Appendices 175

Appendix A: Genome Encoding for Design Problems 175

Genome Design Requirements 175

Genetic Concepts 176

Implementation 177

SAD Function Example 181

List of Figures

Figure 1 Self-Tuning Regulator 45

Figure 2 Design Flow 49

Figure 3 User Specification 51

Figure 4 System Implementation 51

Figure 5 2 by 2 SAD Specification 53

Figure 6 Absolute Difference Specification 54

Figure 7 Generalised Operation 55

Figure 8 Typical Logic Block 55

Figure 9 System with a channel conflict 57

Figure 10 System with the channel conflict resolved 57

Figure 11 2 by 2 SAD function with channels 58

Figure 12 Example System 59

Figure 13 Example System with Type Converter 60

Figure 14 Data Availability 61

Figure 15 Equation 4 as a System 62

Figure 16 Sample and Cycle time 62

Figure 17 Pipeline Stages in 2 by 2 SAD Architecture 64

Figure 18 Annotated SAD Specification 65

Figure 19 Example Component Profile 66

Figure 20 Example System 69

Figure 21 P44[n] Sub-circuit 71

Figure 22 Time Annotated P44[n] Sub-circuit 71

Figure 23 P44[n] Component 71

Figure 24 Pipeline Stages in 2 by 2 SAD function 72

Figure 25 SAD function Implementation A 73

Figure 26 SAD function Implementation B 73

Figure 27 SAD Component A for Implementation A 74

Figure 28 SAD Component B for Implementation B 75

Figure 29 a_{11} 76

Figure 30 Component for a_{11} 77

Figure 31 Genetic Algorithm 79

Figure 32 Parts of a Genome 80

Figure 33: Number of Species 106

Figure 34: Extinctions 106

Figure 35: Evolution of Species 107

Figure 36: Interspecies Breeding 107

Figure 37: Branin RCOS function 120

Figure 38: Shubert Function 121

Figure 39: Michalewicz Sine Function 122

Figure 40: Large Ones Problem 128

Figure 41: Small Ones Problem 129

Figure 42: Allele Alphabet Size Problem 130

Figure 43: Iterated Prisoners' Dilemma Problem 131

Figure 44: De Jong F1 Function 132

Figure 45: De Jong F4 Function 133

Figure 46: Branin RCOS Function 133

Figure 47: Shubert Function 134

Figure 48: Michalewicz Sine Function 135

Figure 49: Multi-objective Problem Objective 1 136

Figure 50: Multi-objective Problem Objective 2 136

Figure 51: Multi-objective Problem Objective 3 137

Figure 52: Multi-objective Problem Objective 4 137

Figure 53 Crowding Distance 147

Figure 54 Crowding Factor 148
Figure 55 The ASGA Performance against other Genetic Algorithms 151
Figure 56 STR LUTs and Sampling Period 153
Figure 57 Multipliers and Sampling Period 154
Figure 58 Logic Cells and Multipliers 154
Figure 59 STR Problem 156
Figure 60 SAD Problem 158
Figure 61 Logic Cells and Multipliers 159
Figure 62 Logic Cells and Sampling Period 159
Figure 63 Multipliers and Sampling Period 160
Figure 64 EKF Results 161
Figure 65 H-ASGA performance 162
Figure 66 Genome Parts 177
Figure 67 Chromosome Types 178
Figure 68 Chromosome: First Section 179
Figure 69 Chromosome: Second Section 180
Figure 70 Identical Designs 181
Figure 71 SAD function for a 2 by 2 window size 182
Figure 72 Genome Outline 182
Figure 73 Specification - Genome Map 183
Figure 74 SAD Genome 184

List of Tables

Table 1 Pre-processor Lookup Table for 8-bit Fixed Point Divider 50

Table 2 Punnett Square 81

Table 3: Parents A and B 85

Table 4: Children A and B after 2-Point Crossover of Parents A and B 85

Table 5: Children A and B after Uniform Crossover of Parents A and B 86

Table 6: 3 Parents 87

Table 7: 3 Children 87

Table 8 Crossover Example 109

Table 9 Genome 116

Table 10 Best ASGA Combinations 138

Table 11 Parents 148

Table 12 Children 148

Table 13 Mutation 149

Table 14 Ranking for Population Size 100 150

Table 15 STR Design Range 153

Table 16 STR Solutions 155

Table 17 SAD 8 by 8 157

Table 18 ASGA solutions 157

Table 19 EKF 158

Table 20 H-ASGA EKF solutions 163

Chapter 1. Introduction

Improving semiconductor technologies and increasing demands are driving the need for design tools capable of working with large numbers of design combinations and flexible architectures [**1**] [**2**]. This research aims to enhance typical design flow and provide automated support tools for new applications such as video encoders/decoders, video motion detection, various digital signal processing algorithms and digital filters [**3**][**4**][**5**][**6**][**7**][**8**][**9**][**10**][**11**]. A focus of this thesis is to provide a tool that helps automate the design process. This type of tool is topical for those utilising new FPGA technologies of increasing capacities. The increased demand on designers necessitates design tools able to consider design decisions more effectively. Design tools must be able to find better designs than their predecessors.

Higher capacity devices and lower costs are making more design options viable. As an example, the Stratix IV E variant EP4SE680 features the equivalent of 681,100 logic elements, 22,977 Kbits of embedded memory, and 1,360 18 by 18 multipliers [**12**], whereas the Cyclone family of devices is a lower cost FPGA option [**12**]. The increasing availability of such options means more combinations can and should be considered by design tools if they are to produce competitive designs.

The ever increasing capacity and performance of programmable chips means that increasingly complex systems are able to be implemented in single chip designs leading to designs called “System-on-a-Programmable-Chip” or a SoPC [**2**][**13**][**14**]. Those designs have the advantage of being implemented in a single technology, which eliminates compatibility problems. An additional benefit is that the designer does not have to consider direct communication between devices within the system design (although the device still communicates via its inputs and outputs), as in multi-chip designs where the resource requirements to handle such inter-chip communication can be considerable.

Work deadlines mean that few options are considered in many systems. A design may be required to be produced quickly in order to meet time to market requirements. This means that the design will typically use more resources than necessary in order to ensure performance requirements later in the design cycle are met and thus will typically follow design paths previously known to be successful. This also means new design avenues may not be explored due to a lack of time, and hardware costs will be higher due to high hardware

resource usage. Automated design processes can alleviate the time requirements, allowing more design options to be considered, potentially lowering the overall hardware resource usage.

The limiting characteristics of any design tool are its ease of use, the range of systems it can be applied to, how useful the tool will be with a range of different applications, and the quality of results obtained. For a design tool to be adopted, the quality of results must be sufficient to offset the setup cost (normally the time taken for a designer to familiarise themselves with the tool). If a tool lacks in any of these respects it will often be of limited benefit.

Design tools must work with many different and often conflicting design objectives. The most typical design goals are performance (in terms of computation time or throughput), cost (in terms of used resources), and power consumption. High performance designs tend to cost more to produce and as clock speeds increase so does the power usage from state switches. Low cost designs tend to have poorer performance as more tasks are sharing a limited number of resources leading to processing delays, or data widths are restricted leading to higher quantisation noise. Low power designs may achieve their goals by reducing clock speeds and hence performance, or use more costly but more power efficient technologies.

Design tools that work with multiple design objectives should ideally propose the pareto-optimal set of solutions for each design problem encountered. The pareto-optimal set of solutions is the set of all non-dominated solutions to a problem. A solution is said to be dominated if another solution exists that is at least as good in every objective and better in at least one objective [**15**].

To assess the performance of any design tool it is necessary to benchmark it against other design tools but this raises three important issues: how to measure the quality of an approximation to the pareto-optimal set, how to compare different algorithms, and how to ensure results are representative of larger problem classes and not problem instances. Because a solution now has multiple objectives there is no longer a single value that can be compared between two solutions. This leads to situations where neither solution is dominant so that both solutions are equally as good. However given two sets of solutions the quality of the solution set can be based on a determination regarding how much of the true pareto-optimal set was obtained, or how close the solutions were to the true pareto-optimal set. Of two algorithms, the algorithm that produces better sets, more often is the better algorithm.

Problem classes are related problems that differ only in their parameters. If the influence of these parameters is controlled, then algorithms may be compared based on their performance on algorithm classes. It is not necessarily sufficient to directly compare algorithm performance between different problem parameterisations. This makes the assumption that the underlying distribution of design goals is not significantly different, which in general is not true. Extreme parameter values can create extreme problem instances that will cause outliers in empirical studies such as extremely high resource costs. These outliers can exercise undue influence on the overall results of algorithm comparison. This is especially true if the average value of design goals is used in an analysis. One outlier can cause an average greater than the majority of results, meaning the average is no longer a true measure of the middle of the distribution.

The quality of an approximation to the pareto-optimal set is usually measured in terms of distance to the true pareto-optimal set and coverage of the pareto-optimal set [**16**]. The distance to the pareto-optimal set measures design quality while the coverage of the pareto-optimal set measures algorithm quality. However, having two different quality measures makes statistical comparison difficult, as decisions often require a single test statistic about which a hypothesis is made. In this work, design problems are treated as multi-objective problems where there are no known relationships between different objectives. A common approach to use weighted sums cannot be used here, as a weighted sum assumes relationships about the relative importance of objectives. For example, cost may be more important than performance. In this work no such relationships are known and objectives must be handled separately.

1.1. Motivation of the Research

The goal of this research is to optimise, measure, and compare design solutions implemented in Field Programmable Gate Arrays, considering resource costs and data throughputs via an automated and quick process. The target applications are those that feature extensive use of a repeated set of arithmetic and other basic operations. These are typically DSP problems although other problems, with repeated use of a small set of operations could be addressed.

Optimisation algorithms usually require search parameters to be given that guide how the search is performed. In genetic algorithms these are the population size, number of generations, crossover rate, and mutation rate. Genetic algorithms maintain a set or

population of solutions. The population size designates the size of the set or population of solutions. A population is manipulated to find other solutions, using a process that gradually refines solutions over many iterations. The number of generations designates the number of iterations performed. Within each iteration, the genetic algorithm creates new solutions using crossover and mutation operators. The crossover rate determines how many solutions are created using crossover, while the mutation rate determines how likely a solution is to mutate.

The goal within this thesis is to promote ease of use of the optimisation process with reduced search parameter requirements and reuse of components in designs, via component templates and abstractions. This requires an optimisation algorithm, a method for comparing results from different algorithms, and a design framework for representing design problems and different solutions to those problems. The optimisation algorithm should have as few parameters as possible and not be sensitive to poor parameter choice. The proposed algorithm as described in later chapters has only the population size parameter that must be defined. It does not have a pre-defined crossover rate and only requires an initial value for the mutation rate. The number of generations is determined by the algorithm during execution. The design framework also allows reuse of design components in other designs.

In this thesis preference is given to automated techniques rather than Computer Aided Design (CAD) tools to assist the designer due to the target systems being large and not intuitive in nature. Large target systems mean that manual design of multiple options without assistance is not feasible due to the excessive amount of time required. Because problems are not intuitive there is no clear design path to follow. These two factors mean that multiple design paths must be considered, but due to the time requirements design assistance is required. A design framework is proposed to enable the representation of problems in a form that the optimisation algorithm can process. The design framework also includes representations for solutions in VHDL with their cost and performance profiles.

This thesis also seeks to address the quality of results obtained, by investigating and designing an appropriate algorithm to optimise design problems. To this end a suitable measure of the quality of results needs to be first investigated along with an appropriate algorithm that can be modified for such tasks. Finally, a design framework to represent design problems and their solutions must be developed. In this thesis some of the tasks that may be presented for optimization may be extremely large such that many available algorithms would fail to provide a solution, or cannot guarantee a good approximation. In

this work it is desired to develop an approximation algorithm which adaptively identifies when sufficient coverage of a problem is achieved. Exploration and exploitation are two important search processes. Efficient exploitation bases its search on similarities which can be measured as the frequencies of design decisions in the solutions. Exploration searches design decisions not yet considered. Both methods have similarities with crossover and mutation [**15**]. Consequently the genetic algorithm class of optimisation algorithm was chosen as the basis for further development. The algorithm proposed in this thesis and evaluated is a modification of a genetic algorithm.

The Hypervolume Indicator [**17**][**18**] measures the size of the design space dominated by the approximated pareto-optimal set and, importantly, is a single measure that enables comparison of different algorithms with large sample sizes. The Hypervolume Indicator is formally defined as the hyper-area of the union of hyper-cubes encompassed by the obtained solution front and a common reference point. The Hypervolume Indicator is part of a methodology for comparing multi-objective optimisation algorithms by Zitzler et al [**17**], where it is used to evaluate and compare their algorithm with other leading algorithms. In this thesis the Hypervolume Indicator is adapted for use in parameterised problems typically found in digital design work in order to evaluate the proposed algorithm's performance.

1.2. Selection of Case Studies

In this thesis, various case studies are selected for comparison. A modified genetic algorithm is applied to the multi-objective d-dimensional 0/1 knapsack problem [**19**] first to establish basic relationships between the proposed algorithm and other similar algorithms on a common problem. Further comparisons are then made on a number of case studies that all belong to a class of target systems requiring optimisation. These include the design of a Self-Tuning Regulator Parameter Estimation circuit [**20**], a Sum-of-Absolute-Difference function [**21**], and a five state Extended Kalman Filter [**22**]. The modified genetic algorithm is called an Adaptive Speciation Genetic Algorithm (ASGA) and is used to perform design optimisation. The case studies are chosen because they all pose design problems regarding allocation of resources under constrained situations. In each design case, the constraints take into account which individuals are feasible and as the constraints are built into the encoding strategy, all solutions that can be represented are feasible.

The multi-objective d-dimensional 0/1 knapsack problem is an NP-Complete allocation problem [**19**] for which no efficient deterministic algorithms are currently known [**23**]. This problem was chosen because it is used as a benchmark problem in comparing genetic algorithms by other researchers [**17**]. This aids comparison of the work presented in this thesis with existing material, enabling any benefits and disadvantages of the approach to be highlighted.

The Self-Tuning Regulator problem is computationally intensive due to the parameter estimation performance requirements. However, a comparable manual design is available in the literature enabling the design frameworks and algorithm performance to be examined on a design problem of practical complexity with results compared against a manually produced design.

The Extended Kalman Filter (EKF) is a computationally intensive algorithm when implemented with even a modest numbers of states, but it also has several desirable qualities that make its implementation a viable possibility for high-end systems. The EKF allows for process noise in its system model and measurement noise from its sensors. It also requires a considerable number of matrix multiplications which, when converted to scalar equations, can reach the order of hundreds for a five state filter to thousands or more for higher order filters. A high order EKF can easily become too large to implement in a design, thus inclusion of such an element in a system influences many other design decisions regarding what can be done to provide performance within cost constraints.

The Sum-of-Absolute-Difference function is computationally intensive, due to the number of operations that must be performed in a limited time, but is a commonly utilised function in video motion detection. Results can be compared with those obtained manually in the literature. This problem has a degree of uniformity that allows easy optimisation if provided with the information that summations are of large vectors and vectors are the magnitudes of the differences of two images. This case study is investigated to determine if the various algorithms tested can find this uniformity.

In order to automatically optimise designs as introduced above, a new design representation is used that allows data-flow dominated systems and simple control-flow systems to be optimised with respect to resource costs and performance. The design representation is a core of the design framework used for testing the proposed optimisation algorithm on the various design problems. A Mealy Finite State Machine is used as the model to represent the

state based operations in the chosen framework. These operations can be linked via channels to realise most types of systems. It should be noted that each system is pipelined to increase the sampling rate and any pipeline delays should be noted and allowed for in the system specification. For instance, the output of the third operation in a data path is the output for the system inputs three samples ago.

1.3. Contributions of the Research

This research consists of a design framework, optimisation algorithm, and an extended comparison method for parameterised problems. The following contributions are made:

1. An Adaptive Speciation Genetic Algorithm (ASGA) is proposed that is easy to use and produces results competitive with other leading algorithms. The ease of use is accomplished by requiring only the population size to be specified. An initial mutation rate which may also be supplied will adapt to more suitable values if chosen poorly. This allows the ASGA to be more readily applied to a range of problems with little preparation necessary to determine the search parameters to use. ASGA features a novel speciation crossover operator that identifies key genetic sequences using tagging and uses sets of tagged individuals in isolating individual genetic sequences of interest. An adaptive mutation rate is also implemented using the DNA of an individual to encode the rate as this links the mutation rate to the species of an individual, the number of generations elapsed, and the difficulty of the problem. In Chapter 7 it is shown that these operators produce better performance in larger design problems and the knapsack problem.

2. A method for comparing non-deterministic optimisation algorithms on multi-objective parameterised problems with unknown distributions is presented. This method extends existing work by allowing for the influence of parameters on multi-objective problems and accommodating problems for which the distribution type is not known. This type of problem represents a large class of problems of research interest. The proposed extension is based on the sign test which is a distribution-free statistical test. This comparison method is used in Chapter 7 to compare ASGA with other genetic algorithms. Genetic algorithms are non-deterministic algorithms. Design problems are multi-objective problems that are defined by sets of parameters, in this case a design specification. Typically the distribution of objective values in a design

problem is not known. It is necessary to be able to determine between two algorithms which is performing better in such a scenario to determine the algorithm's contribution.

3. The ASGA is compared with a selection of leading genetic algorithms on multi-objective d-dimensional 0/1 knapsack problems and the design of a Self-Tuning Regulator parameter estimation circuit, a Sum-of-Absolute-Difference function, and a five state Extended Kalman Filter. These are all problems of practical complexity involving resource allocation. This is used as a test bed for ASGA's performance advantages on practical problems.

4. A framework for designing systems from specification to synthesis is presented with emphasis on optimisation of designs. The design framework allows reuse where several such synthesised subsystems are linked to produce larger systems. The optimisation algorithm is able to optimise these designs based on profiles generated from descriptions of different implementations of operations and subsystems used in the larger system. The novel aspect of this framework is that it uses a simplified pipeline structure that allows stages to be considered separately without impacting on other elements of a design. For instance, the scheduling of one stage does not influence another, nor does the component positioning of one stage influence another. The overhead introduced is typically small and similar to simplifications a designer might make when implementing a complex system. This allows large design problems to be optimised with little resulting overhead.

1.4. Structure of the Thesis

Chapter 2 addresses background material for this field of research. This chapter briefly discusses Design Problems, Multi-objective Optimisation and Evolutionary Algorithms in order to identify some of the issues involved. Design problems are presented as a combinational optimisation problem emphasising the selection of implementations of operations rather than the tuning of system parameters which is a numerical optimisation problem. A brief history of algorithms used in design problems is given along with some of the issues involved in design frameworks. For multi-objective optimisation the measurement of performance, sign tests and case studies is discussed. Finally the nomenclature used in the

field of evolutionary algorithms, the general process of convergence and implementation is described.

Chapter 3 discusses system design while presenting the design framework proposed in this thesis. This chapter describes the model used for representing design problems and potential solutions. Also covered is the insertion of channels to facilitate component reuse. Design optimisation is discussed with reference to the design framework proposed in this thesis. The motivating examples for the included features in this design framework are given. The proposed design framework is then presented. This includes specification, channels, scheduling of components, the timing model, pipelining, profiling, scheduling of a system, and three design case studies (the parameter estimation circuit of a Self-Tuning Regulator, the Sum of Absolute Difference function, and the Extended Kalman Filter).

Chapter 4 gives an overview of genetic algorithms and indicates where the proposed optimisation algorithm departs from a conventional genetic algorithm. This chapter also describes the flow of a genetic algorithm and the nomenclature used. Common genetic operators in use such as selection, crossover, and mutation are explained. The issues that influenced the development of the proposed algorithm such as mate selection, appropriate optimal search parameters, and controlling the balance of exploration and exploitation are also described.

Chapter 5 presents the proposed Adaptive Speciation Genetic Algorithm (ASGA). This chapter gives the development goals, implementation, extension, and constraint handling for the selection operator. The specific crossover operator chosen is described with details on tagging individuals with a species tag, mate selection, ancestry, species management, and implementation details. The mutation operator is detailed along with discussion on adaptive strategies and encoding strategies. This chapter also describes the influencing factors in the determination of the operators and their implementation.

Chapter 6 presents the experiments conducted in the design of the ASGA. It begins by presenting a test bed of functions used in comparing algorithm performance. The experimental designs (including performance measures that combine the average performance with the variation in performance), are presented along with results obtained from experiments on this test bed of functions. The proposed algorithm is hereafter compared against a Simple Genetic Algorithm (SGA), where the chosen SGA has been specifically tuned for each problem to ensure it operates under optimal conditions whereas

the proposed algorithm is deliberately left un-tuned to look at its robustness and ease of use. These results show that the new algorithm can be applied un-tuned and still produce performance results comparable to an SGA where significant effort has been spent in tuning. This means the algorithm can be applied, with relatively little modification, to a wider range of problems.

Chapter 7 evaluates the new algorithm and compares it with other leading genetic algorithms. First, an improved methodology for comparing multi-objective optimisers is presented. The method extends current methods from the literature to cover classes of related problems instantiated from common templates by supplying sets of parameters, and for which the distribution of objective values is unknown. The major features of each of the algorithms used for comparison are also outlined. These algorithms are the Pareto Envelope-based Selection Algorithm (PESA) [**24**], the 1-1 Pareto Archived Evolution Strategy (PAES) [**25**], the Strength Pareto Evolutionary Algorithm 2 (SPEA2) [**26**], and the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [**16**]. The first comparison is on the well-known Knapsack problem. The ASGA is then evaluated using three design case studies: the parameter estimation circuit of a Self-Tuning Regulator (STR) [**20**], the Sum-of-Absolute-Difference function (SAD) [**21**], and the Extended Kalman Filter (EKF) [**22**]. Lastly a hybrid of the ASGA utilising suitable features of the PESA to gain improved performance, is proposed and evaluated. By combining the PESA and the ASGA improvements are gained in some but not all design problems.

Chapter 8 outlines the conclusions drawn from this research and future work which could be conducted. The design framework and optimisation algorithm comparisons are also summarised. In the future work section an improved specification format is indicated that would allow easier specification of design problems. A pre-compiler to convert specifications to an intermediate format could lead to significant reductions in processing times for solving difficult design problems. An extension of the library to incorporate more component prototypes would also extend the range of systems that can be developed.

Chapter 2. Background Issues in the Design Process

This chapter introduces background material to design problems, multi-objective optimisation, and evolutionary algorithms. This material is used to help develop a design framework, means for comparing multi-objective optimiser performance, and the design of a genetic algorithm for use in the developed design framework.

2.1. Overview

Digital design problems such as digital filters and state observers are commonly encountered in Field Programmable Gate Arrays (FPGAs). These digital design problems are discussed here with emphasis on automated techniques. Design automation covers tools used in assisting or performing design work such as Computer Aided Design (CAD) tools to automated systems such as hardware-software partitioning algorithms [**27**]. Common design problems include filter design, plant controllers, image processing and communication systems. Design tools can range from, interactive, such as Printed Circuit Board layout tools to fully automatic where the tool accepts a specification on the functional requirements of a problem and finds a set of solutions. This thesis is restricted to systems implemented in a single FPGA taking a net list, defining a set of combinations, and then testing a subset of those combinations to approximate the pareto-optimal set.

Multi-objective optimization is introduced with respect to measuring the performance of multi-objective optimizers. Multi-objective optimization attempts to find the pareto-optimal set of solutions to a problem. As discussed briefly in Chapter 1, the pareto-optimal set of solutions is the set of all solutions to a problem that are not dominated by any other solution. A solution dominates another solution if it is at least as good in every objective and better in at least one objective. The performance of an optimizer may be measured by how close the proposed solutions are to the true pareto-optimal front and how well the front is covered or the spread of those solutions [**16**]. Closeness to the front measures solution quality. The coverage of the pareto-optimal set indicates the quality of the optimizer at finding all solutions. The Hypervolume Indicator [**17**] is used in the tests performed in this work against other genetic algorithms. The Hypervolume Indicator is a measure of the amount of the solution space dominated by the proposed solutions.

Evolutionary algorithms are detailed focusing on genetic algorithms. Evolutionary algorithms are algorithms that perform optimization by mimicking biological evolutionary

processes. There are four major areas of evolutionary algorithms. These are evolutionary strategies, genetic algorithms, evolutionary programming, and genetic programming. Evolutionary strategies use real numbers to represent solutions which evolve during searching for solutions to numerical problems. Genetic algorithms use genes with low numbers of alleles to explore combinational problems. A gene is a position on the chromosome. An allele is the value held by a gene on the chromosome. Evolutionary programming evolves the coefficients used in a fixed program. Genetic programming evolves a program to match a set of inputs to the set of desired outputs. The design work considered in this thesis typically involves considering combinations of different design elements in a digital system. This is most suited to a genetic algorithm.

2.2. Design Problems

This section discusses design problems as problems of combinational optimization. A history of previous design algorithms is given along with a brief outline of the features and requirements of a design framework. The requirements and examples of specification formats are then described. Following this, the problem of rapid growth in design combinations or the time complexity of design problems is discussed. In addition, the availability of different technologies and architectures are also discussed.

2.2.1 Combinational Optimization

Improving technologies and increasing demands are a driving factor in design tools for considering large numbers of design combinations in flexible architectures. High capacity devices such as the Stratix IV [**12**] and low cost options such as the Cyclone [**12**] device series lead to higher expectations for subsequent designs. This requires more advanced tools to assist in the design process. There are a large number of design combinations that must be considered in a restricted time.

This work considers operations with multiple implementations each having different performance and cost profiles. The task becomes one to select appropriate combinations of implementations to produce the required functionality but also to explore different alternatives in terms of performance and cost. In general there will be no clearly superior design but rather a set of pareto-optimal solutions. These solutions will represent the range of possible performance and cost scenarios that are viable design alternatives. Bhattacharya et al [**28**] gives a system with a software and hardware implementation of each operation.

Design decisions in this work have multiple alternatives rather than the binary decisions typical in genetic algorithms. The encoding used allows a gene to have more than two alleles allowing more than two possibilities that are mutually exclusive to be easily represented. This is done by encoding all mutually exclusive possibilities in different alleles on the same gene. This impacts on the implicit parallelism [**15**] of genetic algorithms as decisions are no longer binary. The effect is to reduce the level of parallelism which must be compensated for.

In order to automate the design process with many different implementation choices, it is necessary to find common design objectives that measure the characteristics of all implementations in order to profile different implementations. Typical design objectives involve performance, cost, and power consumption. Performance can be measured in terms of the sampling rate, latency, quantisation noise, or approximation error. Here the sampling period which is the inverse of the sampling rate is used. Cost may be measured in terms of financial cost or hardware resources allocated. While financial cost is common to all designs it is subject to change according to market supply and demand. Since this work involves only FPGA implementations, financial cost may be substituted with the number of 4-input lookup tables, embedded multipliers, and on-chip memory used. These three elements are common in many different FPGA devices.

2.2.2 Historical Methods

Historically one of the earlier systems, SOS, used mixed integer linear programming which proved too slow for large problems [**29**] [**30**]. This system had allowed an arbitrary multiprocessor architecture and would schedule and allocate tasks to the system. Given that the problem is NP-Hard [**31**], a deterministic algorithm could be expected to have long processing times. An NP-Hard problem is at least as hard as the hardest problem in NP [**23**] meaning there are no known efficient solutions to the problem.

Heuristic approaches superseded these with strategies such as changing implementations to meet performance or cost goals while starting from a case of the best performance or lowest cost. Vulcan started with an all hardware solution and moved operations to software to meet cost restrictions [**29**]. Cosyma started with an all software solution and moved operations to hardware to meet performance requirements [**29**]. These approaches use a heuristic that makes assumptions about the design path to follow. These assumptions allow the heuristic

algorithm to finish earlier by considering fewer solutions and ideally their heuristics mean that the solutions that are considered are more likely to be better solutions.

Non-deterministic approaches followed that could avoid the pitfalls of previous approaches, but meant that proving the algorithm correct became more difficult. These approaches did not follow set design paths, but rather had a probability of pursuing a path. Simulated Annealing is one such example [**32**]. In this strategy the optimizer moves to a better point near its current solution most of the time. For a small percentage of the time, however, it may move to a point worse than its current solution.

Genetic Algorithms are currently a common approach to design problems featuring multiple design criteria [**33**]. A Genetic Algorithm is a non-deterministic approximation search algorithm that mimics biological evolution. The principal processes in a genetic algorithm are selection of the fittest, mating, and mutation. A genetic algorithm manipulates genotypes which are encodings of solutions to problems and not the solutions themselves [**15**]. This encoding allows a genetic algorithm to separate its operators from domain dependencies in a problem domain. For instance, the same mutation operator can be used for several different problems as it is the encoding that separates the problems and not the mutation operator. The selection operator can also be separated which opens the opportunity for an array of different methods for working with multiple objectives and constraints.

2.2.3 Design Framework

Design work may be subject to a number of constraints. These constraints fall into two categories: constraints that must be met for the design to be acceptable and those that affect only quality. Constraints related to quality may be implemented as objective functions such as performance or cost. Constraints that must be met may be implemented in a selection strategy or in the encoding of an individual. In general, a design problem is a multi-objective constrained optimisation problem corresponding to a given design specification.

System requirements and design features supported need to be considered early in developing design tools in order to ensure good integration of the two. It is more difficult to integrate new design features and systems requirements into a design framework at a later stage than during initial development. System requirements and design features can include dynamic reconfigurability [**34**], software support, component sharing, interrupts, multiple operation implementations and other possibilities. In this work the selected design features have been

restricted to component sharing and allowing multiple implementations of any operation within the same system. This is done to limit the amount of preparation work required before testing of the optimisation algorithm proposed in Chapter 5. The system requirements and design features supported, impact most clearly on the resource cost and performance of a system. This relationship consists of a decrease in cost relating to a decrease in performance, or an increase in cost relating to an increase in performance. Component sharing and multiple implementations allow tradeoffs between resource costs and performance of the system as well. Due to separation of the problem from the optimisation algorithm via genetic encoding, the optimisation algorithm needs only information relating to cost and performance not what system or design features are producing these profiles. The optimisation algorithm should also function with additional system or design features provided with a suitable encoding.

Two broad categories of description methods are behavioural languages and structural description languages. Behavioural languages describe the behaviour of a system in terms of its input and output characteristics. Structural languages describe a system as a set of connected components. In this work the operation model is a Mealy Finite State Machine. This state machine can be implemented behaviourally or structurally as a set of connected components. The top level of the design framework proposed in Chapter 3 is always a structural definition (net-list) as it links existing components to implement a larger system.

The design process has a set of identified stages including specification, verification, simulation, optimisation, and synthesis, as described following:

- Specification is where the functional requirements of a design problem are given. If necessary, a separate additional specification gives any design constraints to be applied.
- Verification is where the function of a design is checked against the specification.
- Simulation is where the design is simulated to assess its input and output characteristics based on an example scenario. Simulation may be used in verification.
- Optimisation is where the performance, cost, and other design criteria are used in the development of a set of designs of high quality.
- Synthesis involves the generation of appropriate files for third-party compilers to produce a design implementation.

Producing a framework that allows for future extensions ensures that unforeseen developments do not render the framework obsolete. Future extensions could include adding additional components to a library for designers to choose from, adding additional design criteria that can be used as objectives, or a range of other options. It is preferable to design a framework such that incorporation of additional extensions requires a minimum of changes to the framework but some features such as dynamic reconfigurability would require large changes if implemented.

The performance and cost of a design decision is essential to the optimisation stage. Synthesis can be separated into the generation of appropriate file types following this stage. The performance and cost of a design decision can be determined by storing profiles on each implementation of every operation. The profile contains the performance and cost details of a single implementation of an operation. From these profiles a composite profile of the entire system can be produced. The composite profile is used to direct the optimisation stage to good regions of system designs. This can be done without having to generate files for third-party compilers, or wait for their compiled results. Consequently, this substantially reduces the time taken, as third-party compilers typically consider more scenarios at a finer granularity than is necessary in the optimisation stage.

2.2.4 Design Specifications

Specifications ideally give a complete, unambiguous and concise representation of the function of an application without bias to its implementation. Design goals and constraints are given separately when processing a specification. A complete specification ensures that all necessary information to produce a design is present. An unambiguous specification ensures that there is only one valid interpretation for the function of an application. A concise specification contains only the details necessary for the specification to be complete and unambiguous. A specification should not bias implementation as this can have adverse effects on design optimisation. Design goals and constraints are given separately so that the same functional specification can be used for a variety of different applications for which different goals or constraints may exist.

Choosing familiar models and languages leads to shorter learning curves for new design frameworks. Languages such as C++ and VHDL are well established and literature covering their use is commonly available. Models such as “Finite State Machines” are also well covered in the literature and familiar to systems designers. Such familiarity means that

resources to solve problems or find out how to implement a design are more readily available shortening the time taken. Models and frameworks based on familiar existing systems tend to benefit from sharing similar syntax and keywords.

Different models and languages have their limitations, which affects the types of systems they can be used for. ESTEREL, for example, is particularly suited to reactive systems [**35**]. Reactive systems must be able to respond to external events. Models and languages for reactive systems must allow for this. In reactive systems, interrupt latency can be important. Signal processing tasks however frequently perform a fixed set of operations cyclically. These systems require a model or language that recognises sampling rates and preferably allows pipelining. The sampling rate of such systems is important.

SystemC [**36**], VHDL [**37**], and Java Modelling Language (JML) [**38**] are examples of languages that might be applied in this area. SystemC is used to model the behaviour of hardware or with appropriate compilers such as those found in Synopsys' System Studio [**39**] used as a Hardware Description Language. Another option is translator algorithms that convert SystemC models to VHDL [**40**]. VHDL is a common hardware description language for which platforms exist that simulate and compile for FPGA implementation, for example Quartus [**12**]. JML is a behavioural specification language. JML combines the design by contract approach of Eiffel and the model-based specification approach of Larch [**41**] with elements of refinement calculus [**38**].

Finite State Machines (FSMs) and Petri nets are examples of graph models that might be applied in this area. Finite State Machines model an operation as a state with a next state function and an output function. Mealy Finite State Machines allow the output function to be dependent directly on the state and the current inputs while Moore Finite State Machines only allow the output to be dependent on the state. Petri nets consist of places and transitions. A place may hold a number of tokens. A transition determines what tokens are necessary before firing. When a transition fires it removes some tokens from its input places and places some tokens in its output places as governed by the transition rule [**42**].

Two common approaches in verification are by formal methods or by simulation. Formal methods are valuable in safety critical applications [**43**][**44**][**45**] while simulation methods have advantages in systems that are approximations of another process. A formal method proves the function of the system to be correct. This is achieved usually via logical analysis of states and is subject to state explosions in complex systems. A state explosion is where the

number of states to check rises quickly as the problem complexity rises [**46**]. This can limit the feasibility of formal methods in some applications unless the problem can be subdivided. Simulation methods only test the system for a limited set of the possible inputs. A simulation method generates the output vectors for the system based on the inputs supplied. As such, usually not all scenarios are tried. If all scenarios are tried the same complications as for a formal method may arise involving state explosions. With complex systems and large input vectors simulations can be slow.

2.2.5 Time Complexity of Design Problems

The time complexity of design work means that fast alternatives are essential if seeking more than a superficial coverage of a problem. This includes two important aspects. Firstly, the design algorithm used must be efficient. This means that relatively few designs must be tried before finding high quality solutions. Secondly, designs must be able to be partitioned and scheduled quickly. In this respect there may not be sufficient time to calculate the best set of times a pipelined system may accept inputs but to rather assume the system accepts inputs at a regular rate. When designing a system pipeline, stages are usually divided equally in terms of processing requirements. This type of division leads to systems that accept inputs at a regular rate.

Given the large numbers of design combinations possible, automated tools or CAD tools are important in aiding the design process. For example a system with 2 identical operations that can be implemented in 4 different ways and shared, has 20 different combinations. This system may be implemented using one resource for each operation or one resource shared by each operation. For single resource systems there are 4 choices using each different implementation. For two resource systems there are 16 choices (4 x 4). The total number of implementations is 20 (16 + 4). A system with 3 identical operations and 4 different ways of implementing has 116 different combinations. This system may be implemented using either one, two, or three resources. A single resource system has 4 choices. A double resource system has 16 (4 x 4) resource type options and 3 sharing options for a total of 48 choices. A three resource system has 64 (4 x 4 x 4) resource type options. This gives a total of 116 (4 + 48 + 64). A system with 4 identical operations and 4 different ways of implementing has 756. This system may be implemented as a single resource, double resource, tripled resource, or quadruple resource system. A single resource system has 4 resource type options. A double resource system has 16 resource type options and 7 different sharing options for a

total of 112 choices. A triple resource system has 64 resource type options and 6 different sharing scenarios for a total of 384 choices. A quadruple resource system has 256 resource type options (4 x 4 x 4 x 4). This gives a grand total of 756 (4 + 112 + 384 + 256). The rate of growth of design combinations as the number of operations increases is super exponential. This is caused by increases in the number of combinations of operation implementations and increases in the number of combinations of different sharing scenarios. These coupled, generate a super exponential sequence. Typically a system will have more than 4 operations and the operations will not be identical. A library could hold more than 4 implementations for an operation. A practical system would have in excess of 100 operations. Given the time to manually process a design, a designer is likely to process only a few designs. Given the large number of possible designs, this may be an insufficient coverage of the problem. To ensure sufficient coverage of the problem, the time taken to process a design must be reduced either by aiding a designer or automating the processing of a design.

Automated techniques can process systems more quickly than a designer with a CAD tool but a designer with a CAD tool is likely to make better decisions about which design paths to explore. In some cases designs include design decisions a designer will make automatically based on experience. This defines a set of design paths that a designer will explore using a CAD tool. On a design by design comparison an automated algorithm will be able to check a design faster than a designer. Generally, an automated algorithm does not take account of intuitive elements. An automated algorithm must discover these elements during its search. As such an automated algorithm typically evaluates more designs than a designer in order to obtain a good set of solutions. Automated algorithms however have advantages where there are no intuitive elements due to their faster processing of designs.

Deciding which design paths to explore in large design problems can be difficult to achieve in the time available. A large design problem can have many different design paths that appear to be good solutions. However in the time available determining which the best design paths are can be difficult. This can require testing many different design paths to obtain a set of solutions. Usually there are many equally good solutions. Multi-objective search algorithms can quickly search different design paths in order to solve this problem. The more recent algorithms will attempt finding the pareto-optimal set in a single run. Older algorithms may require several re-initialisations with different search parameters to find the pareto-optimal set.

2.2.6 Target Technology

For a design system it is necessary to define how general the supported architectures and target technologies are to be. Architectures include how many processors (and types), programmable chips, external memory, and dedicated logic might be included in a design. This affects how designs are partitioned onto the available resources. Multi-processor scheduling is an NP-hard problem [**47**]. Using more than one chip requires communication between the chips and appropriate buses with suitable communication protocols. For each chip there are also many varieties produced by different manufacturers. Each has different numbers of resources and differing performances. Processors can have different instruction sets. The more architectures and target technologies supported the greater the number of design combinations. As the number of design combinations increases the time required to perform a search of the solution space also increases.

Programmable technologies and processor platforms are commonly used architectures. These combine the benefits of fast execution in programmable chips and relatively cheap implementation on processors. A processor is cheaper in terms of resources to implement large algorithms than a programmable chip. A programmable chip offers a high degree of parallelism and customization. A common architecture is one processor and an accompanying programmable chip. With the recent viability of soft-cores single chip architectures featuring only a programmable chip are a suitable alternative. A soft-core is a processor implemented by programming a section of the programmable chip to perform the function of a processor. Using a soft-core means that inter-chip communication between processor and programmable chip is eliminated.

The increasing capacity of programmable devices is making soft-core processors more feasible for greater numbers of applications. Improvements in soft-core design are making them more competitive with hard-core processors [**48**]. These processes are facilitating the ease of producing systems on a single programmable chip. One of the benefits of System on a Programmable Chip (SoPC) designs are decreased communication overheads. Soft-cores allow the single chip design to implement software algorithms on the same hardware as other system operations.

With increasing features in programmable technologies, more complex systems can be implemented in single chips. More recent programmable chips include embedded multipliers, on-chip memory, phase locked loops and transceivers [**12**] in addition to logic

cells. Multiply Accumulate (MAC) units are specialised resources for efficiently performing multiplication and accumulate operations commonly found in filters. On-chip memory is usually configurable and a more effective alternative to implementing memory using logic cells. [**49**] discusses issues in FPGA-based embedded system design.

2.3. Multi-Objective Optimization

This section discusses how to measure algorithm performance and describes improved means for comparing algorithms. Finally, the case studies used in this work are discussed.

2.3.1 Measuring Performance of Multi-Objective Optimisers

To determine whether or not one algorithm performs better than another algorithm on a design problem requires a comparison to be made between sets of solutions for a multi-objective problem. A single multi-objective solution represents a problem in that when compared to another solution, there may be no clearly better solution. One may excel in some criteria while the other is better in the remaining criteria. Sets of solutions, however, are easier to compare as overall, they represent a differing degree of success at solving a given problem. The algorithm with the higher degree of success performs better for the problem on which comparisons were made. Whether this success can be extended to other problems depends on the experimental design of the analysis.

Multi-objective optimisation algorithms may be compared using the concept of pareto-optimality and dominance. A solution J is said to dominate another solution K if and only if all objectives in J are as good as those in K and at least one is better. The pareto-optimal set is the set of all non-dominated solutions to a problem. This set represents all the possible tradeoffs of different objectives where no clear advantage is present. This scenario reflects the idea that different objectives cannot be measured in terms of each other. For example objective A and B are treated as separate entities, so that no relation A = f(B) exists that allows a value B to be converted to an equivalent value in A. This differs from aggregate methods which assume a function “f” exists where this is possible.

The success of an algorithm can be measured in terms of the distance to and spread along the pareto-optimal front of any solutions obtained from the algorithm [**16**]. The distance to the pareto-optimal front is a direct measure of the quality of the solutions obtained. The closer to the front the fewer better options exist. The spread along the front represents how well the

algorithm covered the possible range of solutions. This is a measure of the quality of the algorithm. The distance to the front is also a measure of the quality of the algorithm. Having two separate measures for algorithm quality presents the problem that different algorithms may excel at either distance or spread and may themselves be pareto-optimal with regard to each other. In this case one algorithm would have a better spread and the other a better distance measure. Another important problem is that in many problems of practical interest the position of the pareto-optimal front is not known. This means that neither the distance to the front, nor the spread along the front can be calculated.

The Hypervolume Indicator is an important solution to both the problem of an unknown pareto-optimal front and multiple measures of algorithm quality. The Hypervolume Indicator measures the amount of the design space dominated by the solutions proposed by an algorithm. The Hypervolume Indicator is thus a single measure of algorithm quality. It also does not require knowledge of the position of the pareto-optimal front. However the ability to differentiate between a good set due to coverage and a good set due to closeness is lost. This loss only occurs if the pareto-optimal front is known. In problems of interest to this work the location and shape of the pareto-optimal front is not known.

2.3.2 An Improved Comparison Method

A sign test can be conducted on Hypervolume Indicators between algorithms to assess algorithm performance with a degree of confidence [**50**]. A sign test involves pairing samples from two populations. The signs of the differences of each pair form a new sample set. If the two populations are not equal then the set of signs will be inconsistent with a binomial distribution with $p = 0.5$. If the two populations are equal the set of signs will be a binomial distribution with $p = 0.5$.

A sign test avoids the problem of unknown distribution type in design problems by instead focussing effort on showing results to be inconsistent with the conclusion of no difference. There is no dependence of the sign test on the type of distribution of either population in terms of showing a difference. This is because the sign test is based on showing an inconsistency with the signs belonging to a binomial distribution with $p = 0.5$. This is irrespective of the distributions of either population from which the original samples were taken.

2.3.3 Case Studies

Case studies from plant controllers, motor control applications and video motion detection were used to assess algorithm performance on design problems. The multi-objective d-dimensional 0/1 Knapsack problem was used as a test benchmark for comparing performance with other algorithms. These case studies (see Chapter 3 for definitions and Chapter 7 for results) were chosen because they are of similar size to anticipated problems that would be encountered and most have manually designed solutions documented in the literature.

Advanced motor control applications call for controllers that require few sensors or place limits on the types of sensors that can be used [**22**]. This requires estimation techniques to determine motor parameters which involve intensive data processing where resources and time are at a premium. The Extended Kalman Filter offers good performance in such applications but is computationally intensive. This means that EKFs are often implemented in reduced orders so that they do not exceed device capabilities. Higher order EKFs offer improved performance but usually their implementation costs are too high for them to be feasible. For this work, a five state EKF for estimating the speed, stator current and rotor flux of an AC cage induction motor measuring only the stator currents and controlling the stator voltage is used [**22**].

Video motion detection systems have a stream of data windows arriving and must process each window before the arrival of the next window, but fully parallel implementations are usually too expensive to implement. The majority of the computational requirements are usually composed of calculating the difference between a reference image and the current image. This is usually done by summing the absolute value of the differences between pixels in the images at various offsets [**21**]. This procedure is repeated for each offset to find the offset that gives the minimum sum. Although featuring relatively few types of operations the number of operations required to be performed in a sample period is high. Usually the number of operations is too high for a completely parallel solution. This problem involves deciding how many resources to allocate and how much parallelism to allow for, trading performance and cost to implement.

2.4. Evolutionary Algorithms

This section gives an overview of evolutionary algorithms, a definition of the nomenclature used, a description of evolutionary process, issues with algorithm convergence, and implementation issues.

Common design problems of interest are NP-Complete. This means that it is necessary to use non-deterministic search algorithms that due to problem size do not search every alternative. As such, an optimisation problem was viewed as a task to minimize the probability of an alternative algorithm producing a better result. The probability of pareto-optimal designs making given design decisions was viewed as the source for making decisions on optimisation. A high probability associated with either a poor or good design would indicate the influence of that decision. Each design tried was viewed as a sample of the true population of designs (the set of all designs possible). Optimisation was to be achieved by biasing the sampling scheme to increase some probabilities and decrease others dependent on which designs were better. Between any designs changes can be made based on either similarities or differences. Using a larger sample gives a higher the degree of confidence in any decisions. This sample was akin to the population in a genetic algorithm, while the crossover operator was similar to working with similarities and the mutation operator was similar to working with differences. The genetic algorithm was therefore chosen for the base algorithm design. The concepts of genetics, evolution, and species were adopted as analogies for how operators worked.

2.4.1 Overview

Evolutionary algorithms base their optimisation techniques on biological evolution in order to solve complex problems. Evolutionary algorithms utilise the idea of survival of the fittest to select potential solutions and iteratively refine these solutions through mating and mutation to find progressively better solutions. The idea of survival of the fittest is that those most suited to their environment are most likely to survive and reproduce. Mating combines the characteristics of parents into one or more children. The replication is seldom without minor variations or mutations that occur as genes are combined and replicated.

Evolutionary algorithms can be applied to a wide range of problems, for which the objectives to be optimised are treated as black boxes. Evolutionary algorithms utilise payoff functions to guide their search. They do not need to incorporate specialised knowledge about the type

of objective functions they are supplied with. For example, linear programming assumes linear functions while gradient descent bases its process on having access to the derivative of a function or an approximation to the derivative. For evolutionary algorithms this means they do not benefit from special cases where objective functions have simpler solutions. It does mean that they can be applied to a wider range of problems than more conventional optimisers.

The field of evolutionary algorithms is split into four: evolutionary strategies which work with numerical optimisation problems, genetic algorithms which work with combinational optimisation problems, evolutionary programming which evolves program coefficients, and genetic programming which evolves programs. Evolutionary strategies tend to encode genes as real valued numbers which means effectively the number of alleles is infinite. Chromosomes are usually short in evolutionary strategies compared to genetic algorithms. Genetic algorithms have genes with a small number of alleles per gene but usually have longer chromosomes than evolutionary strategies. Large numbers of alleles and shorter chromosomes favour numerical problems while smaller numbers of alleles and longer chromosomes favour combinational problems. Evolutionary programming and genetic programming are the programming counterparts of evolutionary strategies and genetic algorithms.

2.4.2 Nomenclature

A solution in an evolutionary algorithm is represented by an individual. The term individual is similar in connotation to an individual in a biological view. Each individual can have different characteristics and different qualities. An individual may belong to a group or species that bears some similarities.

An individual includes a genotype and a phenotype. The genotype of an individual stores all the genetic information necessary to produce that individual. The phenotype is how the genetic information functions as a whole.

The genotype is a set of genes. A set of genes may be called a chromosome. In biological systems individuals may have multiple chromosomes. The genotype in an evolutionary algorithm records all the design decisions pertinent to that individual and usually consists of a single chromosome.

A gene holds a single allele from the alphabet of possible alleles for that gene. An allele is the value of a gene. There are usually few alleles in a genetic algorithm and infinite alleles in an evolutionary strategy. It is not unusual for a genetic algorithm to have only 2 alleles per gene.

A phenotype is a set of objective values calculated from the individual's genotype. The phenotype is the functional result of the combination of alleles present on genes in the genotype. For instance if gene 1 had value A then the allele for gene 1 is A, but this might cause the individual to be red. In this case the phenotype would be red. A phenotype need not be unique to an allele. For instance an allele of C might also cause a red individual.

Individuals are grouped into populations. Some genetic algorithms utilise speciation. This means that while all individuals belong to the population not all individuals need belong to the same species. Speciation in genetic algorithms is used to restrict mating and to pursue multiple solutions to the same problem. This has a similar function to biological species.

When a new population is produced this is called the next generation. The next generation is derived from the current population. The term generation emphasises the passing of time. A generation may also refer to the population of that generation. In each generation there is a single population. A set of generations corresponds to a sequence of successive populations.

2.4.3 Process

Evolutionary algorithms use a coding of the problem rather than the actual problem in their solution space exploration. This means that evolutionary algorithms feature a decoder that decodes the genotype into a form from which the phenotype can be produced. This separation of problem from genotype allows evolutionary algorithms to use the same operators for selection, mating, and mutation for a variety of different problems. This contributes to the evolutionary algorithm being able to be applied to a large number of different problems.

Evolutionary algorithms search by maintaining and updating a population of potential solutions. An evolutionary algorithm works with a population of solutions. This is an important distinction from more traditional algorithms that proceed from a single point. This affords evolutionary algorithms a degree of resistance to random variations and poor decisions in their search process. This contributes to evolutionary algorithms being robust.

Evolutionary algorithms use payoff information rather than derivatives to guide their search. As such, objective functions need not be differentiable nor is there any requirement on the type of function an objective is since these functions are treated as black boxes. These payoff functions can be extended in a number of different ways to accommodate more than one objective or allow for constraints in constrained problems.

Individuals are iteratively selected, mated, and are mutated to produce the next generation. This process is repeated until either a user supplied number of generations have elapsed or a stopping condition has been reached. Selection, involves selecting more frequently those individuals that give higher quality solutions to the problem being optimised. In evolutionary algorithms mating or crossover refers to a mechanism where children are produced with similar genetic characteristics to all their parents. Mutation is where there is a random chance of a genetic change not necessarily present in any of the parents or ancestry of an individual.

2.4.4 Convergence

Evolutionary algorithms are non-deterministic. They are based on random chances where those chances are weighted ideally to produce subsequently better solutions to a problem. There is no guarantee that multiple iterations of the same evolutionary algorithm will produce the same result. Results are likely to be similar however for well designed algorithms as the algorithm will closely approximate the pareto-optimal set.

Proving convergence of an evolutionary algorithm is difficult and where cases cannot be reduced in complexity, proofs are practically impossible [**51**]. The proposed genetic algorithm is a non-panmictic algorithm which means that the probability of interactions between different individuals is not the same. This means that the complexity of proving convergence is very high. Given this problem an empirical study is conducted that gives a high degree of confidence to results. However this high degree of confidence is not 100% meaning that although results are almost certainly (99%) representative of the true situation there is a small chance (1%) that results are an outlier or a rare statistical anomaly.

2.4.5 Implementations

Combinational problems feature small alphabets and long chromosomes. Numerical problems feature infinite alphabets and small chromosomes. This impacts on how mating and mutation are performed. Numerical problems usually have crossover operators that

generate new individuals via a mathematical operation or operations applied to the genes of each parent. A simple example for two parents is to pick a random allele between the two alleles in the parents. Combinational problems however do not necessarily have continuity in terms of the interpretation of alleles. While a numerical example's alleles are a number that follows a sequence, a combinational problem's alleles are an option. Option A and option C need not be either side of option B. In combinational problems children will inherit equal numbers of alleles from each parent. For example a child could inherit either option A or option C from either parent but could not receive option B except via mutation. Mutation in numerical problems is achieved by adding a noise function to alleles to generate a new allele. In combinational problems a gene subject to mutation is assigned a new allele.

Alternative algorithms that could have been used were the differential evolution algorithm (a form of evolutionary strategy), gradient descent algorithms although these tend to converge on local optima, simulated annealing which also faces the problem of local optima, tabu search, particle swarm optimisation, or ant colony optimisation. The genetic algorithm had the greatest degree of similarity to how the design problem's optimisation was being approached with re-sampling, selection by probabilities based on samples, sifting out patterns in solutions and maintaining diversity.

2.5. Chapter Summary

The problems of combinational optimization, design framework development, specifications, verification, and target technology have been stated along with a brief historical account of some techniques. The problem of digital design in FPGAs has been presented as a combinational optimisation problem. Early techniques such as SOS, Vulcan, Cosyma, and simulated annealing have been noted [**29**]. The requirements of a design framework have been given. Particular emphasis is placed on specification of a design, the time complexity of optimisation algorithms, and target technologies. This chapter includes a section on multi-objective optimisation as this design work is a multi-objective problem. Evolutionary algorithms are briefly discussed as a solution to multi-objective design problems.

Chapter 3 presents the developed design framework. This design framework prepares designs as a combinational optimisation problem where a design problem is specified and mapped to an FPGA. The framework focuses on being able to describe systems and optimise solutions. A large part of this framework is the optimisation stage which maps specifications

to solutions. This part is implemented by a modified genetic algorithm. Chapters 4, 5, and 6 cover the optimisation algorithm used.

Chapter 3. System Design Framework

This chapter introduces the design problem, followed by a discussion of design optimisation and motivating examples. The chapter finishes by detailing the proposed design framework. This design framework is intended to facilitate the prototyping of systems using design profiling based on task schedules and resource allocations. Given a schedule and a resource allocation an implementation of a design can be made from a problem specification. This schedule and resource allocation is evolved by ASGA using the genome representation in Appendix A. The novel aspect of this framework is the separation of tasks into pipeline stages so that each stage has a very low dependence on other stages. This means that the scheduling of a system is greatly simplified allowing much larger systems to be optimised. This however introduces a small resource and timing overhead into the design. These overheads are similar to simplifications a designer would make when dealing with a complex system.

3.1. Chapter Overview

This chapter introduces a system as a set of concurrent processes, implemented as components, that communicate with each other to achieve system functionality. The common practise of top-down decomposition of complex functions yields this type of system representation. An example is made of a Self-Tuning Regulator at a very high level only identifying key system parts that would themselves further decompose into simpler components.

The concept of operations as processes that perform a function is given as a tool for building specification models for systems without the implementation concern. Components are given as implementations of those operations and in addition to the properties of an operation they also have a profile describing the attributes of the component. Design problems typically have more than one design goal so the problem of multiple design criteria and multi-objective optimisation is introduced as the reason for profiling components.

The specification model is given as a directed graph of nodes representing operations and edges representing the transfer of information. The need for pre-processing of component profiles into lookup tables indexed by operations per component is highlighted by the potential for a reduction of up to three orders of magnitude in the time taken to search design spaces.

The proposed design flow is presented, with emphasis on the optimisation stage and the preparation of the necessary system information to facilitate optimization. The model of an operation is introduced as a Mealy Finite State Machine with latched outputs for pipelining. The component model is an operation with an additional profile that includes details about how the operation is implemented, its costs, performance, input and output types, and any other relevant attributes. Channels are introduced as a method to enhance reusability of components and their compatibility with other components. The implementation of pipelining, time model, and scheduling of components are detailed with examples of a single pipeline stage. The initialisation and operational phases of components are discussed as a simple method of ordering operations in a system. A system that is designed by the tool can then be automatically incorporated into the user library as a component for future use.

3.2. Introduction

As mentioned above, systems can be modeled as a set of concurrent processes that communicate with each other and the external environment. Such a model is a typical product of a top-down design where functionality of the system is divided into subsections which are then further refined. These divisions are made so that a complex function is broken down into a set of basic operations to be performed. Figure 1 shows an example of such a division. The example gives a breakdown of a Self-Tuning Regulator (STR) [**20**] at the highest level showing the controller design circuit, estimation circuit, and controller circuit. Each of these three circuits would have a further breakdown until the last contains only basic operations.

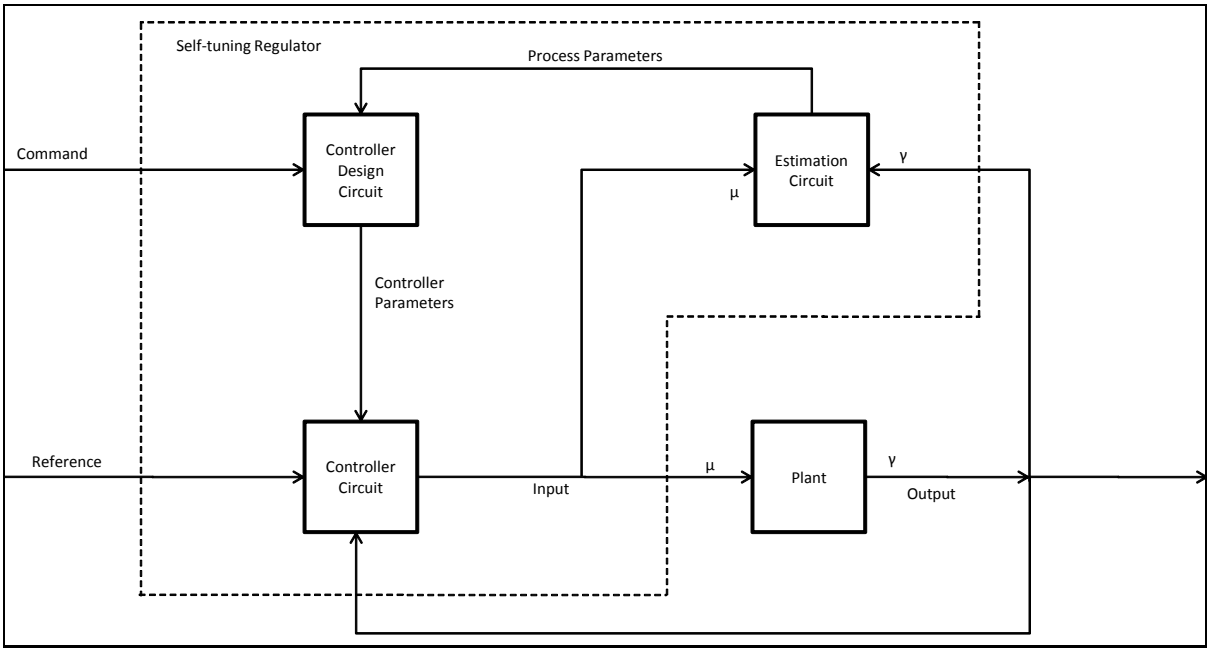

Figure 1 Self-Tuning Regulator

In the STR example (Figure 1), μ and γ represent the communications with the external environment. The dotted line encompasses the STR. The controller circuit takes a reference signal which may be generated from a number of sources; similarly the controller design circuit takes a command. Within the STR the controller communicates μ with the estimation circuit. The estimation circuit estimates the process parameters and provides these to the controller design circuit. The controller design circuit determines the control parameters to use to control the process and communicates these to the controller circuit. The controller circuit uses the reference signal and control parameters to determine the required process inputs to generate the required process outputs and supplies this to the plant.

Operations are described in terms of function without bias to implementation. The Operation designates what a part of the system does, not how it is done. This important distinction enables design systems to substitute different implementations of the same operation. This is useful for exploring alternative implementations when designing a system. Alternative implementations can produce systems that have lower costs, higher performance, lower power consumptions or a combination of these when compared to a smaller set of designs the user may have produced manually.

The quality of the solution is optimized to satisfy multiple criteria. Whenever more than one criterion is involved, quality is determined relative to the pareto-optimal front. In this case,

every proposed design should not have a design that dominates it. The proposed set of solutions, however, gives a designer the range of possible solutions to a design problem.

Automated tools search the design space automatically while CAD tools give designers a choice of design paths to explore. The design space is usually too large in practical problems to search more than a small fraction of the space. Automated tools may quickly search a larger number of design alternatives than a designer with a CAD tool could. However, a designer with a CAD tool is likely to make more informed decisions regarding which design paths to explore. In large systems where the design space is complex and its intricacies are difficult to predict an automated tool can try sufficient numbers of designs to learn how to approach a problem. A designer may not have sufficient time to gain this understanding of the problem before a product deadline.

This work focuses on data-flow dominated systems, in particular digital signal processing tasks with small operation sets. However control-flow dominated systems also contribute a large proportion of design problems, therefore the work adopts a format that allows control-flow systems to also be implemented. Three case studies are considered for the system, all are data-flow dominated systems although some may be present in larger systems that also involve control-flow components. These case studies are the parameter estimation circuit of a Self-Tuning Regulator (STR) [**20**], an Extended Kalman Filter (EKF) [**22**], and a Sum-of-Absolute Difference (SAD) function [**21**]. As discussed in earlier chapters, the STR involves the adaptive control of a process, the EKF used is an observer to an AC cage induction motor and is required to estimate the motor parameters, and the SAD function is a common function in video motion detection.

3.3. Design Optimization

New applications usually consider time and spatial requirements but other requirements such as power consumption are becoming important. The importance of meeting multiple requirements mean that most design tasks are multi-objective optimization problems, in some cases they are also constrained problems. This means that there are multiple solutions to any given problem.

Design optimization requires that each operation, such as multiplication or division, be scheduled and operations be allocated into the available components. While the available hardware resources may be limited by the technology, an optimization process still has to

determine which technology to use and how many of the available resources to allocate to implementing components. Both scheduling and allocation are intrinsically linked. The decision of how many resources to allocate to components and the number of operations sharing each component determines how much time is required for each component to complete its required work load. Allocating higher numbers of resources leads to higher costs but more resources mean the time required to complete all operations is generally lower. Conversely, lower numbers of resources leads to components having more operations being bound to them, which means more time is required to complete those operations.

Usually both resource cost and system performance are factors in determining solution quality. As a result problems are generally multi-objective in nature. Power is a typical third objective. The pareto-dominance rank as outlined by Goldberg [**15**] is a common measure of solution quality in these situations. This procedure assigns ranks, depending on whether a solution is dominated or not and successively removes individuals once they are ranked, while assigning subsequent non-dominated individuals a higher rank.

The focus of this work is on data-flow dominated systems modeled as sets of processes that operate concurrently and communicate with each other. Control-flow features are easier to implement if contained within a single operation in this framework. If spreading control-flow features across multiple operations, the user needs to be aware that all operations are pipelined as this introduces synchronization issues into the design work.

3.4. Motivating Examples

The primary task of a design optimization algorithm is to explore the trade-offs between resource costs and performance. The Self-Tuning Regulator example [**20**] and Extended Kalman Filter algorithm [**52**] are problems of sufficient complexity to determine an algorithm's success at this task. These two examples involve two types of resources (embedded multipliers and logic cells) and a sampling rate when implemented on FPGAs [**12**]. This allows a number of trade-offs between resource costs and performance. They also highlight that a design can be both smaller and faster than other alternatives. This is shown by the fact that the multiplexers used in sharing components are not negligible in resource costs especially as the number of operations sharing the component increases. This means that for large numbers of operations using the same component, the multiplexer costs can outweigh the component savings.

Most operations have many variants that can be grouped by differences in parameters. A common operation where this applies is digital filters. A large number of filters can be described that differ only in the number of taps, coefficients, and word length. It is not necessary or possible to have a library containing every possible combination. Therefore, operations that share such common elements should be grouped under parameterised operations. When supplied the appropriate set of parameters a parameterised operation can be instantiated. This reduces the size of the required libraries for optimisation algorithms.

Control-flow dominated systems are a sizeable proportion of design problems; however they have different requirements to data-flow dominated systems. A state register is included that allows Finite State Machines to be included as operations for limited control-flow support. The model described in 3.5 is capable of realising most basic control-flow applications.

3.5. Design Framework

This section gives an overview of the design framework, details on specification and specification interpretation, and examples of the three design case studies used in this work.

3.5.1 Overview

The proposed design tool consists of a system specification model, which is used as an input to an optimisation algorithm that performs allocation of resources and scheduling of the specification model, to produce the pareto-optimal set of solutions for the design problem and to generate VHDL code to realise any member of that set. Figure 2 shows the design flow. There are three entry methods shown in the figure; these are finite difference equation (FDE), direct net list entry, and by third party tool. FDE is a procedure that converts a set of finite difference equations to a net list. FDE is easier for a user to interpret than the net list format, but is not as general as the net list format. Direct entry allows a user to provide a text file that lists operations and connections in net list format. Direct entry is the most general format but is also the most difficult to use. Third party tools that produce text files following the net list format used can provide simpler methods for a user to specify a system that is then converted to net list format. The net list format fully specifies the system to be implemented.

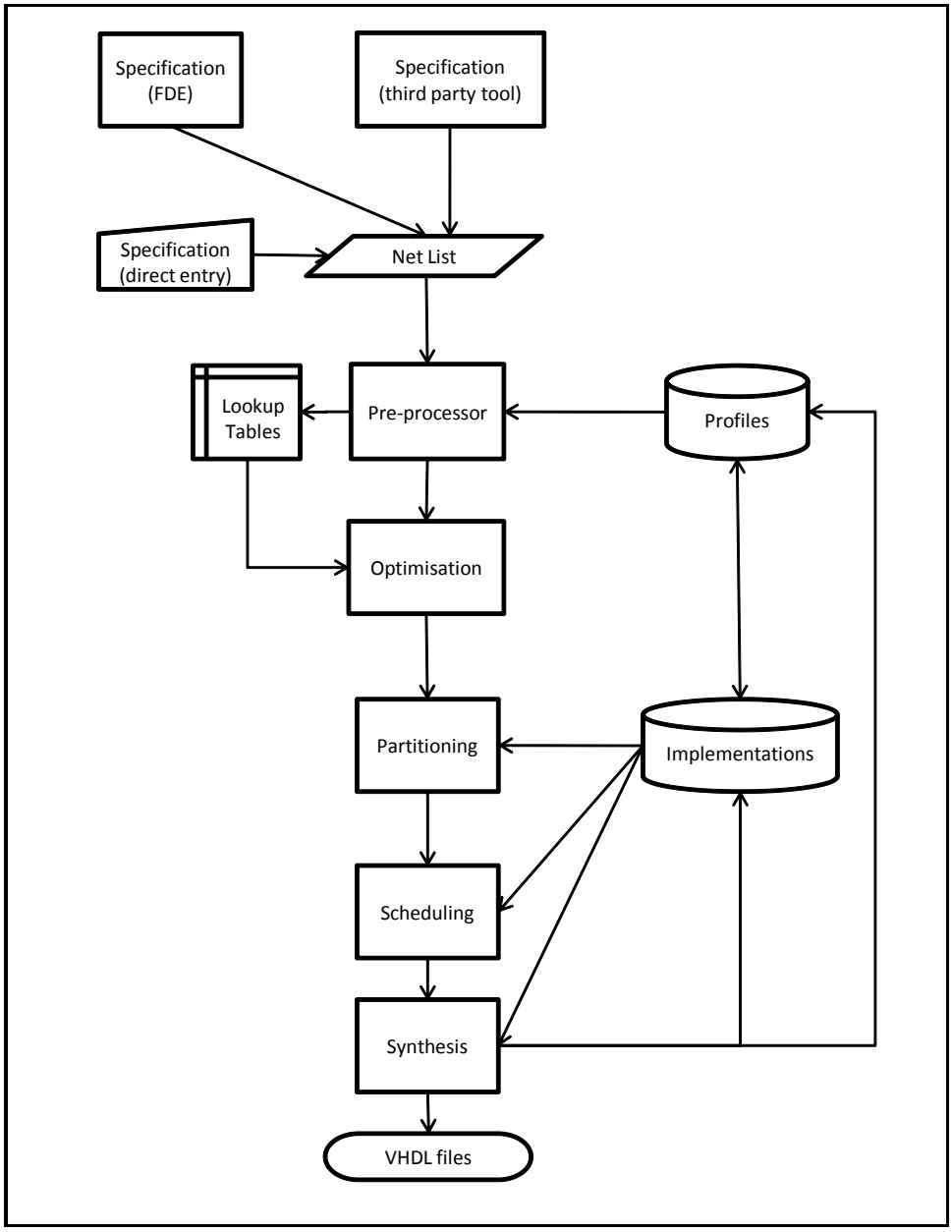

Figure 2 Design Flow

A pre-processor stage can be involved where parts of the net list are converted to sets of lookup tables and numbers of operations per component. Table 1 shows a typical lookup table calculated by the pre-processor stage. The first column, labelled "operations per component" indexes the table so that the implementation cost of a shared component can be directly looked up along with its sampling time allowed. For instance, binding 8 operations to a single 8-bit fixed-point divider requires 263 logic cells and means that the system sampling time can be no less than 9 cycles. These times include allowances for multiplexers used, storage of intermediate results, and storage of delayed inputs. Usually a pre-processor lookup table includes columns for embedded multipliers and other design criteria but for a

simple component these columns are not required; there are no embedded multipliers used in this component. For some components calculating each entry in the lookup table can be time consuming. By using a lookup table each possibility is calculated only once whereas without the lookup table each occurrence would be calculated every time it occurred.

Table 1 Pre-processor Lookup Table for 8-bit Fixed Point Divider

Operations per Component	Sampling Time	FPGA LUTs Used
1	1	99
2	3	129
3	4	151
4	5	173
5	6	197
6	7	219
7	8	241
8	9	263
9	10	287
10	11	309

These lookup tables are generated from a library of profiles of components. Components are indexed by function in these libraries. The optimisation stage uses the lookup tables to calculate the implementation cost of different operation to component bindings. The optimisation stage produces a set of architectures that designate how operations are to be allocated resources and scheduled. The allocation stage uses the number of resources required to implement each component involved. The scheduler then binds each operation to a component separated by time; the required multiplexers and storage registers are added automatically. Figure 3 gives a sample user specification. This specification requires that three inputs A, B, and C have function F() applied to them to produce outputs F(A), F(B), and F(C). If the allocation phase determined during optimisation provided only two F() components the design framework would add a multiplexer and two intermediate storage registers when implementing the system. There are three possibilities for how these would be added depending on whether, A and B share a component, B and C share a component, or A and C share a component. If A and B shared a component the resulting system would be given in Figure 4 where the added components are shown with dashed lines.

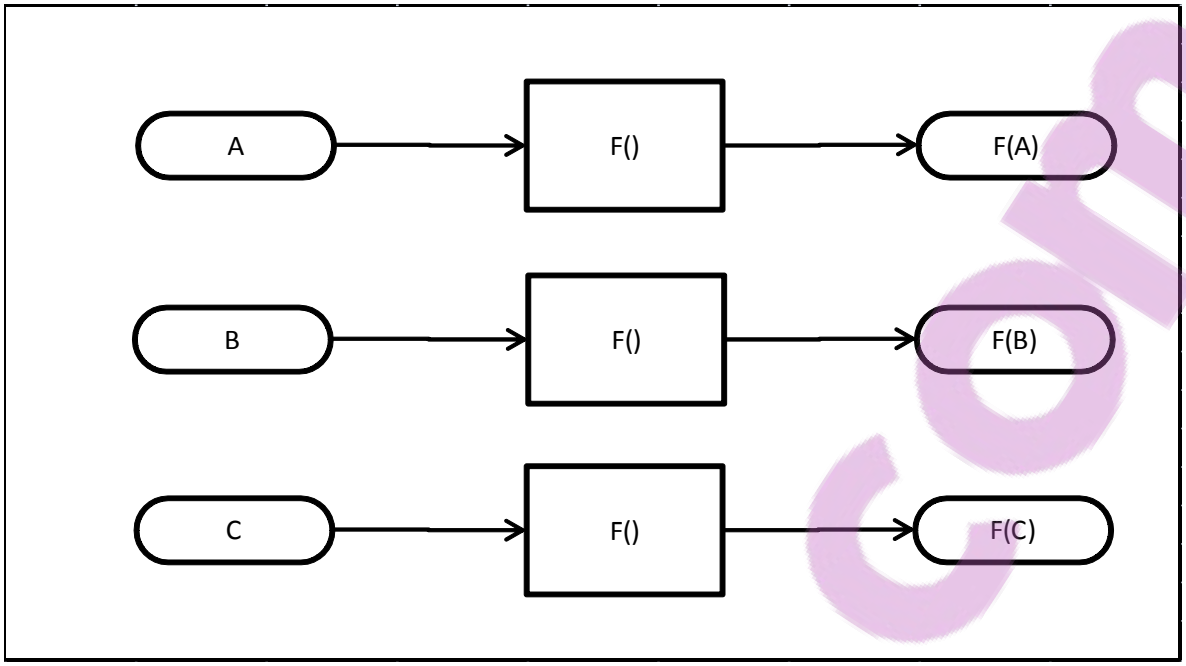

Figure 3 User Specification

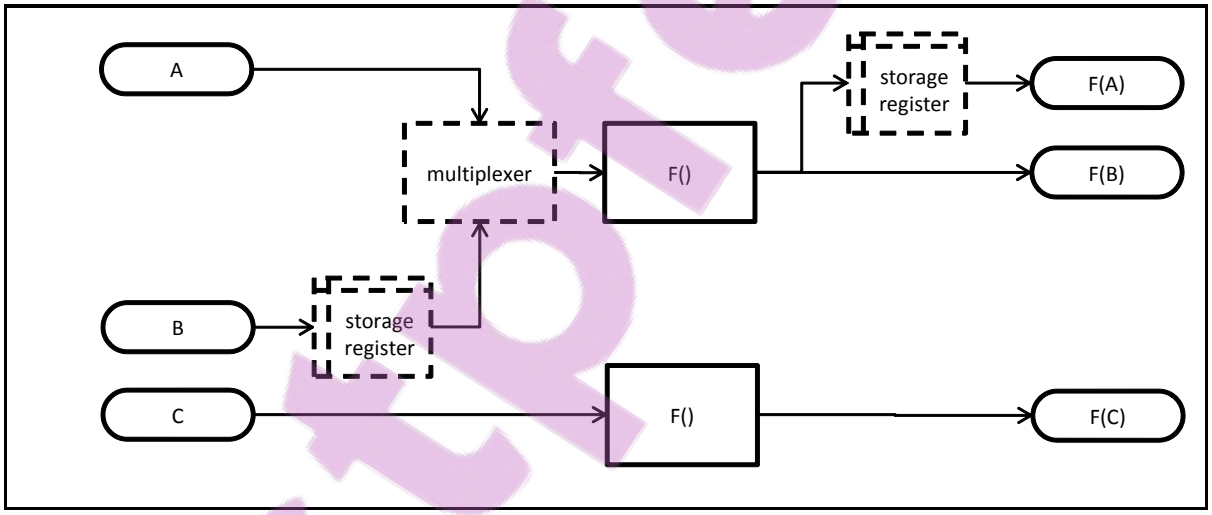

Figure 4 System Implementation

The final implementation stage then generates the code required to implement the proposed solutions. The pre-processor ensures that generated lookup tables account for multiplexers and storage registers added by the scheduler. This pre-processing stage adds no functionality and serves only to reduce the time taken by the optimisation stage; in practice the pre-processor reduces time taken for the optimisation stage of the design flow by three orders of magnitude on the SAD and EKF case studies described and evaluated in Chapter 7.

The design flow is dependent on either manual entry or other systems to produce the required net list, which is then interpreted with a user supplied library. This means the design flow is very flexible as to net list optimisations and target technologies. In many application domains there are optimisations made to designs that are common to all designs. These optimizations can be included into third party tools in their generation of the net list. The net list should then only include decisions to be made in testing the minimum set of design options required to implement a system. By changing the library, different technologies can be targeted by the design flow.

The Finite Difference Equation (FDE) extension produces a net-list from a set of finite difference equations for digital signal processing tasks. FDEs are easier for a reader to interpret than a net-list but are more restrictive in the types of systems that can be specified. The FDE entry method is included as an example of possible extensions that can generate net lists for this design framework. An example specification for an FDE specification is given below.

$$Y[n] = 34X[n] + 15X[n-1] + 2Y[n-1] - 3Y[n-2]$$

$$Z[n] = Z[n-1]/2 + X[n]/2$$

(1)

where Y and Z are outputs, X is an input, and [n] refers to the current sample while [n-1] is the previous sample and [n-2] is the sample prior to [n-1].

Components created for the user library using a net list are stored by making a copy of the specification and all design decisions made in producing the component, so that the component can be instantiated as necessary without requiring the user to write a specialised C++ description for the function of the component or its profile. All basic components in the library require a C++ description for the function of the component and a set of functions to generate its profile for any set of parameters. By storing the decisions made for components created from specifications, C++ descriptions are not necessary for components created from the specifications.

3.5.2 Design Specification

This section details the features of operations in this design framework, the details of channels for enhancing component reuse, the modelling of time in this framework, how to

construct consistent pipelines in this framework, how components are profiled and their relationship to operations in a specification, and finally how a specification's operations are scheduled onto a system.

Operations

The system specification model used here is a directed graph where edges represent communication channels for data transfer and nodes represent operations or already existing system specification models. When a system specification model is implemented it can be added to the library of existing components. Figure 5 gives the system specification model for a 2 by 2 SAD function as described by Equation (2). On the left are the system inputs $R_{1,1}$ to $R_{2,2}$, and $F_{1,1}$ to $F_{2,2}$. The inputs feed into a layer of absolute difference functions. These absolute difference values are then summed by two layers of adders before being supplied to the system output on the right. The functions called absolute difference and adder may either be operations in the library or other system specification models in a user directory. If they are operations then they are indivisible and found as a basic unit in the library. If they are system specification models then they must have been optimised previously and configurations saved in the library. In this case they are treated as operations with each configuration treated as a unique component. For instance, if the absolute difference function were a system specification model then Figure 6 would be one model of this function.

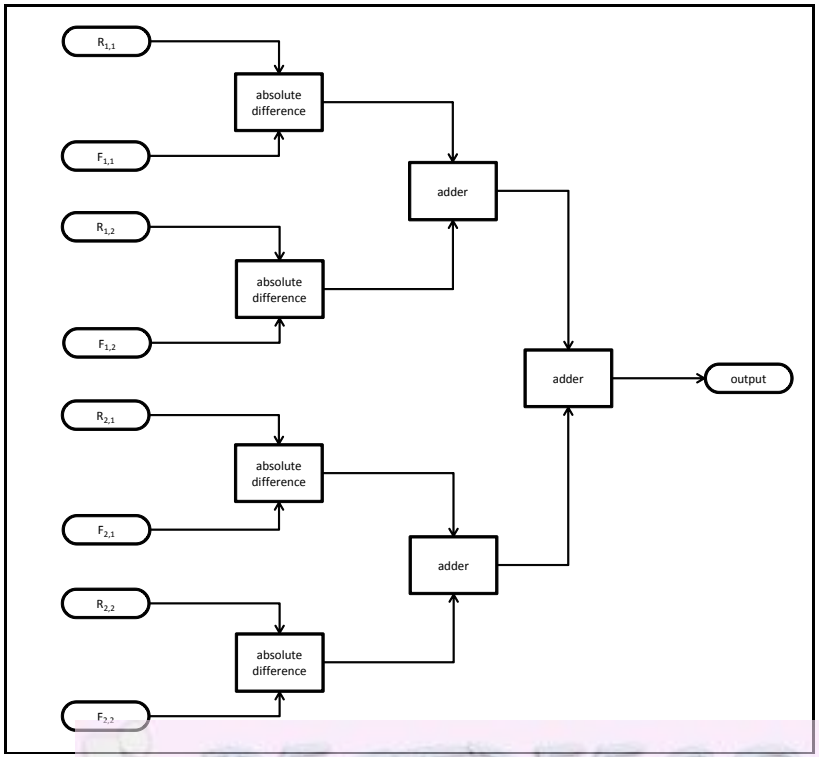

Figure 5 2 by 2 SAD Specification

$$output = \sum_{i=1}^{2}\sum_{j=1}^{2}\left|F_{i,j} - R_{i,j}\right| \quad \textbf{(2)}$$

In Figure 6 two inputs A and B are supplied to a difference function, the results of which are supplied to an absolute value function to produce output C.

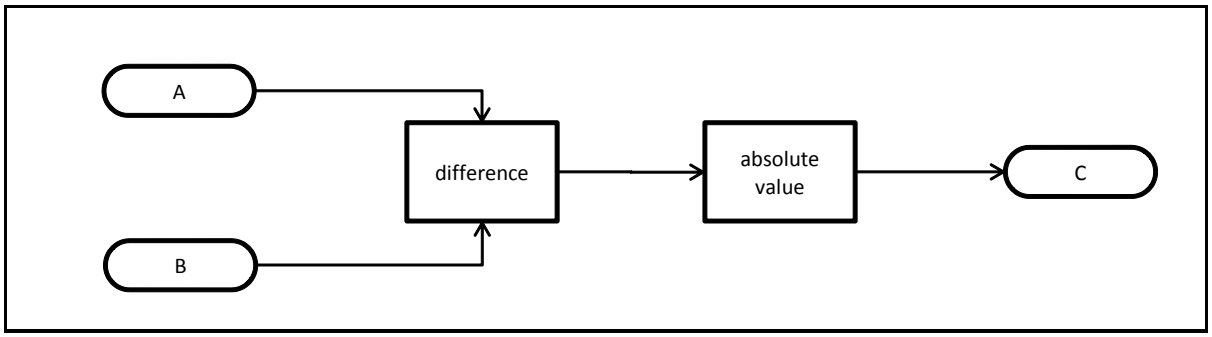

Figure 6 Absolute Difference Specification

Operations are black box functions with inputs, registered outputs, and an optional internal state register. These operations are indivisible and represent the basic unit with which more complex systems can be represented. A component is an operation with an attached profile (detailing one implementation of that operation) and also is used to reference a physical implementation on an FPGA device. A component may also take a set of control signals that help perform the operations bound to that component.

Figure 7 shows a basic model of an operation. The control signals of associated components are not shown in operation descriptions, but are part of the component description in the library. These consist of a clock, clock enable, asynchronous reset, and a set of component specific required control signals. These control signals are implemented as highly specialised inputs. The clock signal is required for synchronising the state and output registers in a system. The clock enable signal is used to place components in standby when not in use so that they do not change either their state or outputs. This is useful in stretching the sampling times of components when necessary. The asynchronous reset is used to reset the system to its default values. The component specific control signals are supplied by a global controller generated by the design framework. This allows common control signal patterns to be implemented using the same resource rather than a separate instance for each component.

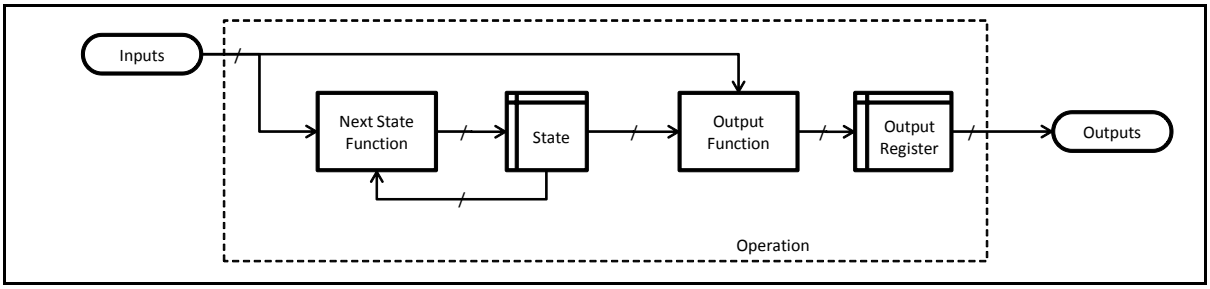

Figure 7 Generalised Operation

The dotted line in Figure 7 surrounds the contents of the operation which are hidden from the user. The operation uses a Mealy Finite State Machine architecture with the addition of an output register. The output register is used to implement pipelining for design solutions composed of many interlinked operations. During a sampling period an operation updates its state and outputs. Figure 8 shows a typical logic block in programmable devices. Inputs are supplied to a look-up table that produces a single output. This output is supplied to a flip-flop and directly to a programmable multiplexer. The multiplexer selects whether the output is supplied directly from the look-up table or from the flip-flop. The cost of the flip-flop is incurred when a logic block is used regardless of whether the flip-flop is actually utilised. Since designs are to be pipelined, and will be using this flip-flop, the operation model incorporates this as a registered output.

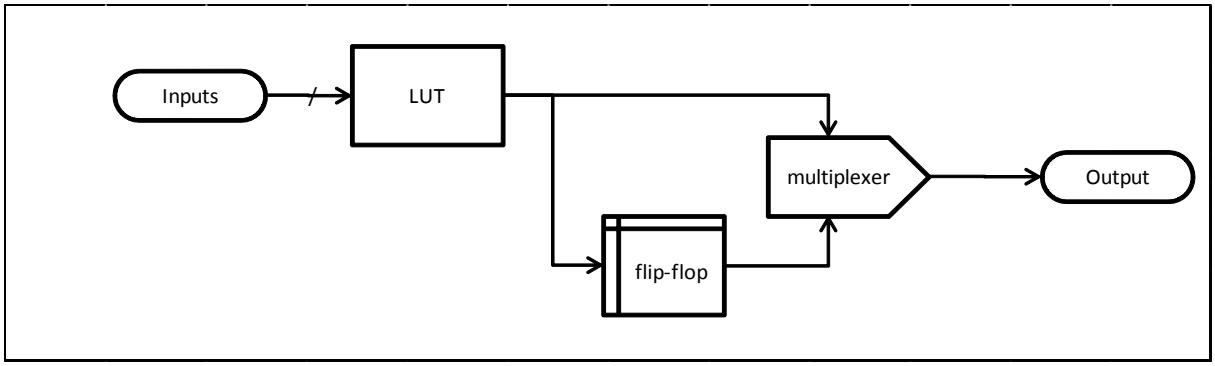

Figure 8 Typical Logic Block

The user library contains the next state function (if a state register is used) and the output function. These two functions must be purely combinational. Operations can have any number of inputs but must have at least one output. Operations may have an internal state register, in which case they must have a next state function which calculates the next value for the state register based on the current value of the state register and the inputs.

Every operation must have an output function which calculates the output based on any combination of the inputs and (if present) the state register. In combinational components there will be no state register and no next state function, although it should be noted that outputs will still be registered by the output register.

In practice there are three additional input/output lines to every component associated with an operation. These are required for a clock signal, a clock enable signal, and a reset signal. Each system has an asynchronous reset signal that resets the entire system to a default state as specified by the user. The clock signal is responsible for synchronising tasks in the system. The clock enable signal is used to switch sections of the system to standby when not in use. The necessary clock and clock enable signals are determined by the framework, while the reset signal is provided as an input line to the FPGA.

Channels

Channels are any method of transferring data between components. Channels can either be a direct connection between two components or may feature a channel component. Every input and output of a component has a type. Types are denoted by tags and can represent any type of signal such as bytes, words, integers, floating point numbers, strings, or analogue outputs for example. Only bytes and words have been implemented in the design framework considered by this work. Given that these types are only used in determining the channel components that are required, they can also be used to denote signal protocols. For instance if a component supplies an output using protocol A and another component requires inputs to follow protocol B then the output could be type A and the input type B. To connect the output to the input, a channel component that implements protocol conversion is required that converts A to B.

General channel components follow the same architecture as regular components but are a special class dedicated to transforming output types. The framework inserts channel components as required provided a component exists in the library that converts an output type directly to the required input type. The framework does not search for possible paths from one output type to the required input type using more than one converter. This would be too computationally expensive during optimisation as potentially every channel would require such a search. Channel components include two specialised components: multiplexers and storage registers. These are used in component sharing.

While only simple type converters, storage registers, and multiplexers are included in existing work the used system model allows for more advanced channels such as communication protocols for interfacing more advanced components, or any other mechanism that a Mealy Finite State Machine can implement. During optimisation and synthesis any necessary registers, multiplexers, or type converters are added. The costs of implementing these resources are accounted for during optimisation. Figure 9 shows a system with a channel conflict where component A and component B have conflicting type requirements. The dotted line divides the two pipeline stages to which the components belong. Figure 10 shows the same system after the framework has added a channel component. A type converter is added to the same pipeline stage as the component it provides an input for, and appears at the start of a pipeline stage. This is shown in Figure 10.

A multiplexer is added to the same stage as the component it supplies an input to and after any type converters present. Storage registers for outputs are added to the same stage as the component it takes an output from and after that component. Storage registers for inputs are added at the start of a pipeline stage on input lines that are multiplexed where that input is not used on the first cycle of the sampling interval and no type converter is present. Any type converter is capable of storing a value for later use and hence it is not necessary to have a type converter and a storage register on the same input. Shared components process their operations at different times within their pipeline stage. Channel components do not introduce additional pipeline stages but they do increase the system sampling period.

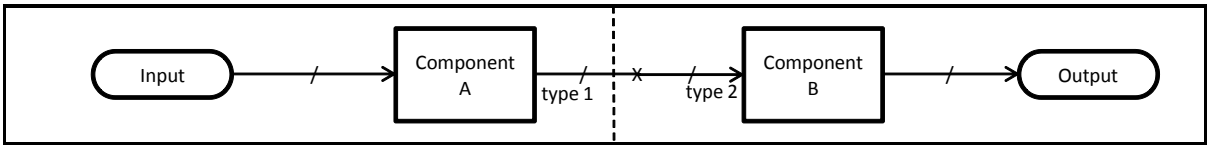

Figure 9 System with a channel conflict

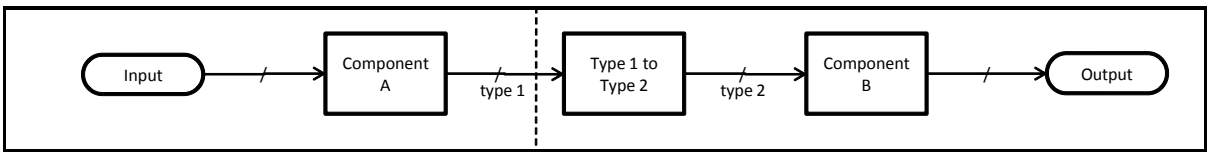

Figure 10 System with the channel conflict resolved

Figure 11 shows an implementation of the SAD function from Figure 5 with added channels. The multiplexer and storage register control signals are not shown. The multiplexers and storage registers are controlled by the same global system controller that generates all component control signals. The global system controller is generated during synthesis and

accounted for in terms of costs during optimisation. In this example two absolute difference operations are bound to each absolute difference component. Similarly two adder operations are bound to one of the adder components. The multiplexers are responsible for switching between sets of inputs to each absolute difference component. While the absolute difference component could have been directly connected to the leftmost adder this would require the design framework to test for such scenarios which are caused by the interactions of sharing different components. Since the framework is intended for fast prototyping as a design guide some system overhead was allowed for in order to reduce optimization times.

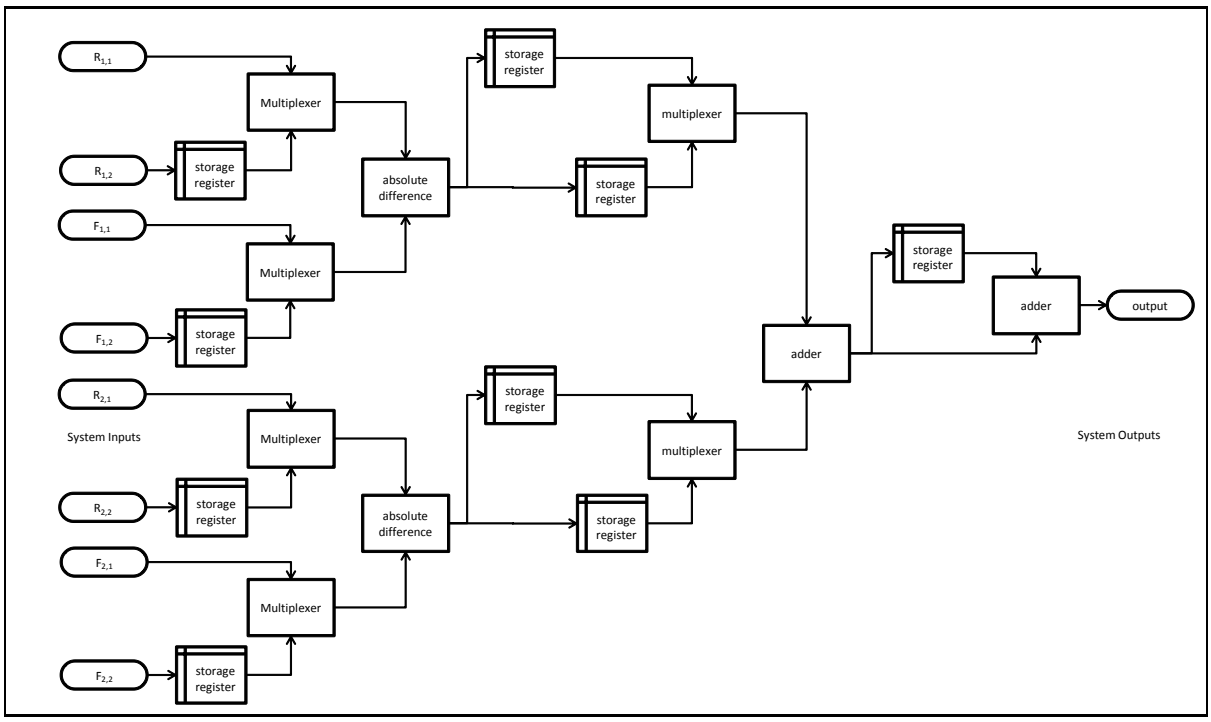

Figure 11 2 by 2 SAD function with channels

A special case should be noted in sharing components with internal pipelining. These components can process multiple inputs far more efficiently than their non-pipelined counterparts. This is reflected in terms of the relationship between the latency and sampling period of the component. Practically, for example, a component with three internal pipeline stages shared among three sets of inputs can process all three inputs in its pipeline at the same time. A non-pipelined counterpart must process them individually. Consider Figure 12 where the component has two implementations. In implementation 1 the component is internally pipelined, while in implementation 2 the component is not pipelined. If implementation 2 takes 9 cycles to complete its operation a typically latency could be 9

cycles before the output is available and 9 cycles before the implementation can accept a new input. Suppose implementation 1 is structurally similar but has been divided into three pipeline stages. The logic of each stage could take 3 cycles to complete if divided evenly with an additional 1 cycle per stage for pipeline registers. This gives a total latency of 12 cycles, and requires 4 cycles between new inputs. In this scenario without any type converters if the number of operations bound to the component is W then for a component with a sampling period of P and latency of M cycles where $P < M + 1$ the sampling period of the system is $P*(W - 1) + M + 1$. In a system implementation where three operations share the component ($W = 3$) the system sampling period using implementation 2 is $9*(3 - 1) + 9 + 1 = 28$ cycles. On the other hand, the system sampling period using implementation 1 is $4*(3 - 1) + 12 + 1 = 21$ cycles. Implementation 1 results in a system with a lower latency and higher sampling rate than implementation 2. However since W is determined during system implementation an optimisation algorithm is required to discover such scenarios.

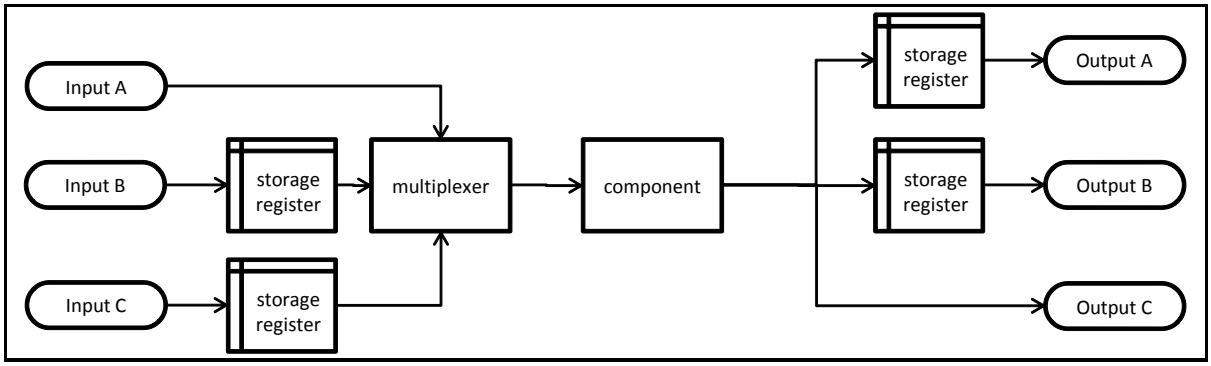

Figure 12 Example System

Figure 13 gives a situation where a type converter is present. A type converter is placed on the input path of a component prior to any multiplexer if present and replaces any storage registers at this point. If the type converter has a latency of Q and a minimum sampling period of R and the component has a latency of M and a minimum sampling period of P then the path from inputs to outputs has Q cycles of latency that may provide the necessary delay for the component to be ready to accept another input. The latency of the pipeline path from input A to output A is $1 + M + 1 = M + 2$ cycles. The latency of the pipeline path from input B to output B is $\max(P, Q) + M + 1$ cycles. The throughput cycle count is $\max(Q + M + 1, P + M + 1, 2*P, R)$ cycles.

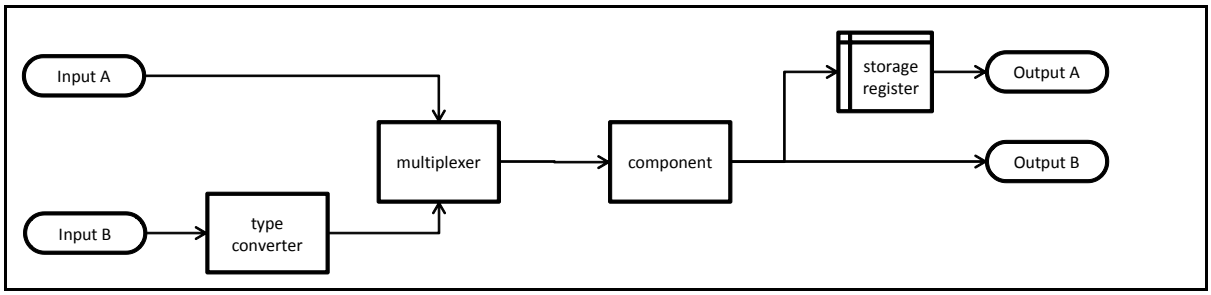

Figure 13 Example System with Type Converter

Ideally, to reduce the number of type converters, the numbers of components requiring an output to be a particular type and the number of components requiring an input to be a particular type should decide whether the type converter is included with the output or with the input (as in Figure 13). Since this decision affects the duration of pipeline stages in a system, as well as the resource cost of the system, this decision should be made in the optimisation stage.

Time

Time is modelled at two levels; sampling time and cycle time. Cycle time is the time taken for a complete clock cycle of the system global clock. Sampling period is the time taken between samples of the system inputs. Sampling period is an integer multiple of the cycle time. This integer, N is calculated for each system architecture and made available to each component should the relationship between sampling period and cycle time need be known for a given operation. The cycle time of a system is technology dependent and can be included in a component description. Equation (3) gives the relationship between sampling time and cycle time.

$$S = C \times N \tag{3}$$

Here, S is the sample time which depends on the system architecture, C is the cycle time which is technology dependent, and N is the number of clock cycles per sample which is calculated within the framework and dependent on system implementation. In some cases it may be necessary to adapt design coefficients dependent on the sampling period. The sampling period is a discrete measure of time. The sampling interval is an actual instance of receiving and processing a sample. The sampling interval occurs over the duration of a sampling period.

This framework assumes an input is available at the beginning of a sampling interval. An output becomes available at the end of a sampling interval. Figure 14 shows the case for N = 1 and the case for N > 1. The arrows mark the beginnings and ends of the sample interval. These are also the times at which the respective inputs and outputs for the sample interval must be ready. At any other time there is no guarantee that either inputs or outputs are available. Storage registers are therefore necessary to store the inputs and outputs if they are to be used outside these times.

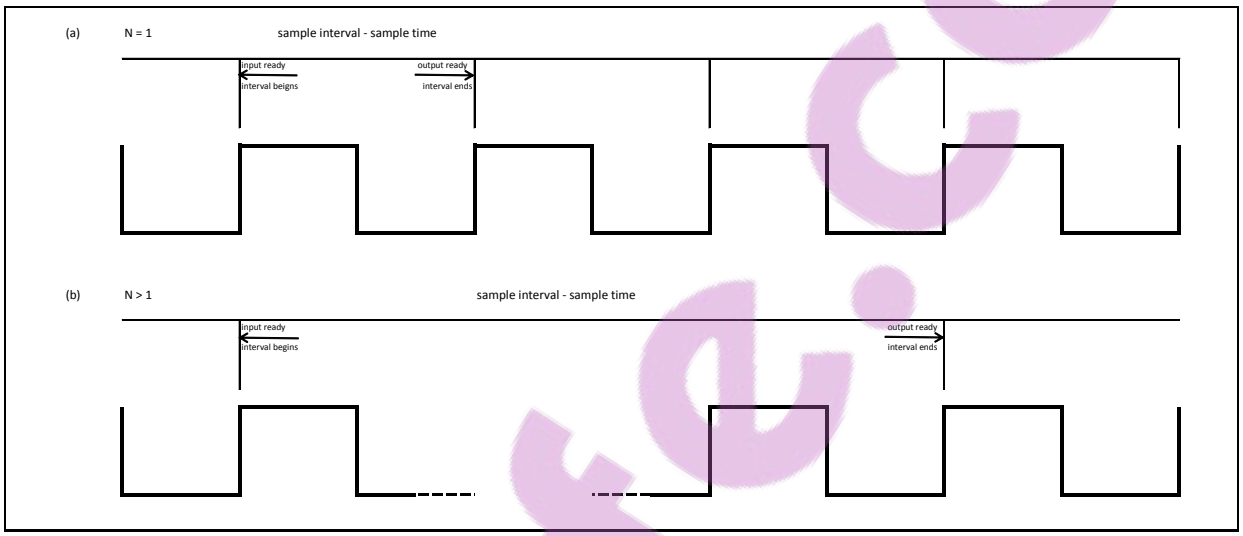

Figure 14 Data Availability

Many applications such as numerical integration (an inherent part of many control systems commonly appearing in the form of Euler approximations) and some filtering are less accurate with larger sampling times and sometimes not stable. If output error is used as a design criterion this requires the simulation producing these outputs to know the sampling time of the architecture used in order to calculate the variation from a reference output signal. For instance, given the function in Equation (4) where the system is integrating X with respect to time, by using a time step of h (where h is the time step between successive X values which by definition is the sampling time, S of a system) to produce Y then the approximation will incur higher errors as h increases. This is an Euler approximation method for integrating a function.

$$Y_{n+1} = Y_n + h \times X_n \tag{4}$$

This would be implemented as the system given in Figure 15.

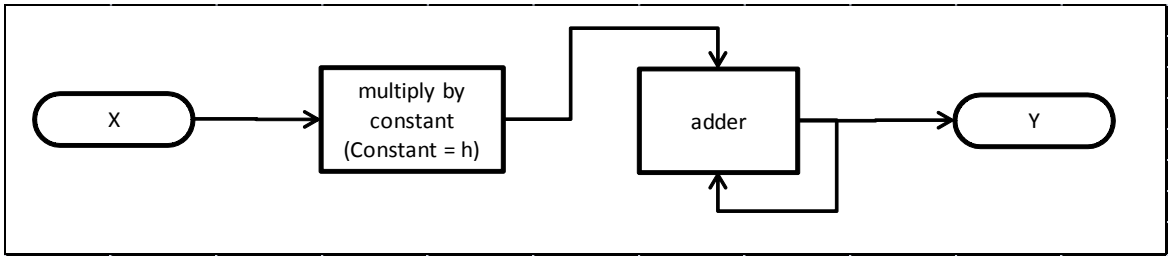

Figure 15 Equation 4 as a System

The component "multiply by constant" requires a parameter to multiply its input by. In this case that parameter is h. . As has been shown this depends on N and C which vary between different system architectures and technologies. Equation (4) is an integration using the Euler method where accuracy is known to be highly dependent on h. Therefore the accuracy of this system is highly dependent on the sampling time of the system.

When systems are simulated in this design framework they are 'cycle accurate'. In future work this could be reduced to 'sample accurate' to alleviate long computation times for the simulator. Figure 16 shows an example of sample and cycle time for N = 3.

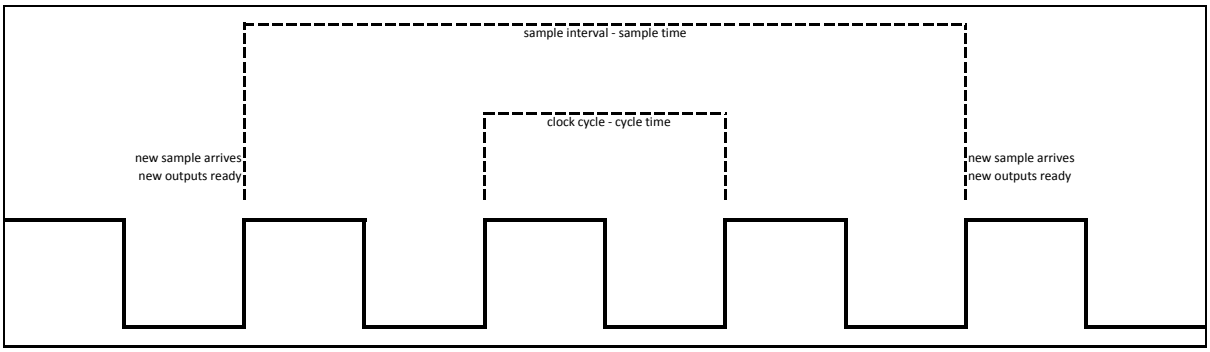

Figure 16 Sample and Cycle time

Pipelining

Generally systems are pipelined in this framework. The specification format assumes a pipeline register integrated in the output of each operation as shown previously in Figure 7. The length of a pipeline stage is determined by the largest number of cycles for any component to complete its assigned operations, including allowances for channel components associated with that component. This means scheduling in this framework involves finding the largest number of required cycles, N, and allocating this as a system wide constant for that architecture.

Channel components do not contribute to the number of pipeline stages in a system and internal pipelines in components are considered a property of those components, which is accounted for by the relationship between latency and throughput of the component. A system composed of many pipeline stages need only consider each pipeline stage independently when scheduling and may use the clock enable to place components that finish early in standby. Such a system might consist of many layers of different pipeline stages. This scheduling system enables the decomposition of an otherwise complex problem into a set of much simpler decisions although it introduces restrictions on when input and output signals must be available.

The number of clock cycles per sample, N is chosen such that every component within the architecture completes all its allocated operations in one sample interval. N is unique for the whole system. All components with a sample time that could be lower, process all their operations at the start of the system sample time and then are placed in standby until the end of the system sample time. Different architectures of a system may have different N values.

A system specification model assumes a pipeline register built into every operation (the output register of that operation). Every operation constitutes a pipeline stage. The user is responsible for ensuring that pipeline stages are consistent in their net list. Figure 17 shows a 2 by 2 SAD specification common in video motion detection applications with the pipeline stages separated by dashed lines. This model has three consistent pipeline stages.

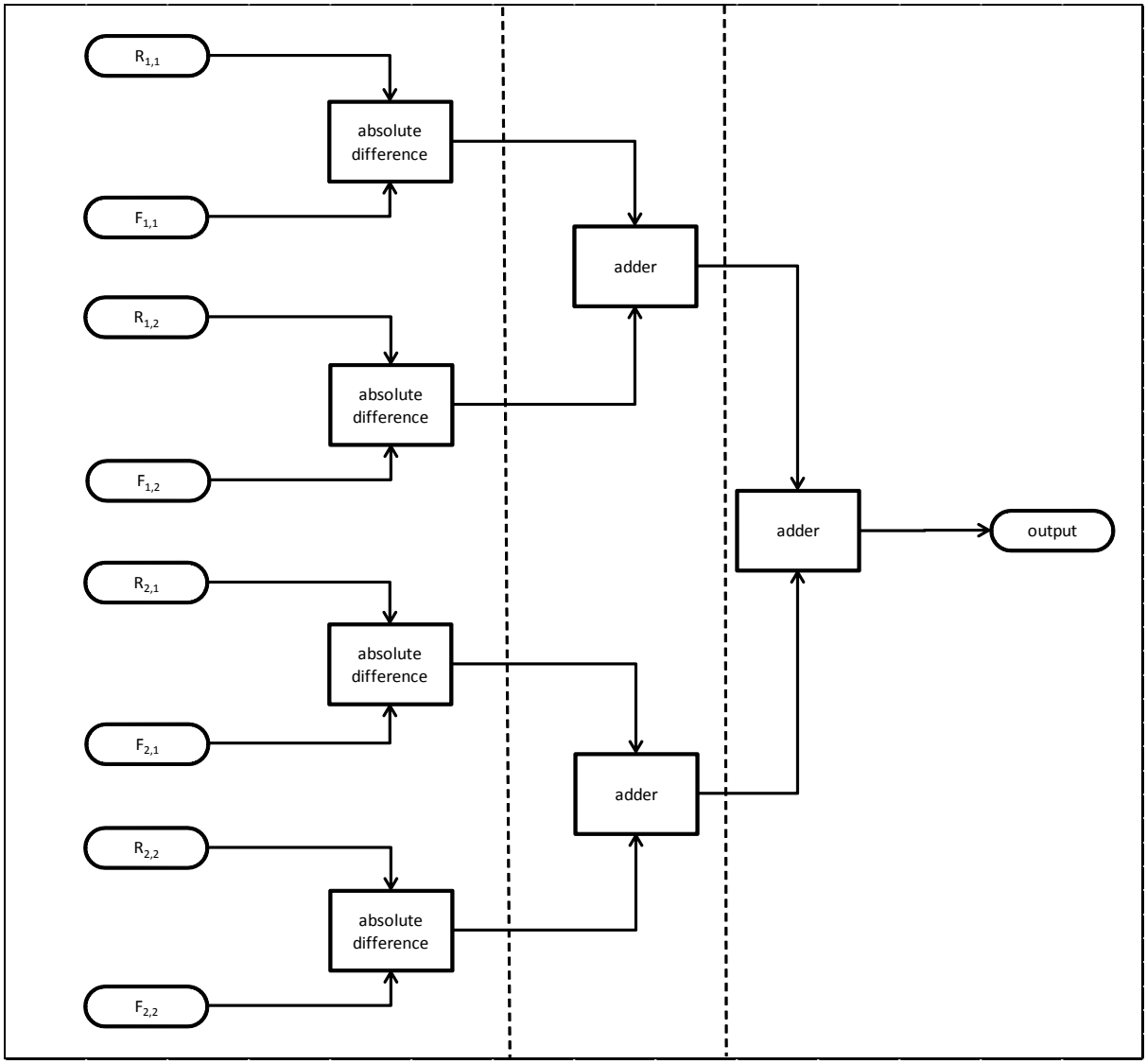

Figure 17 Pipeline Stages in 2 by 2 SAD Architecture

Figure 18 shows the same system as Figure 17 with all internal nodes annotated with their values. As can be seen the output of the absolute difference components relates to the previous values of the system inputs, while the first layer of the adders' outputs is two samples prior, and the final adder's output is three samples prior. The delaying effect in terms of samples must be accounted for in every system designed using this framework.

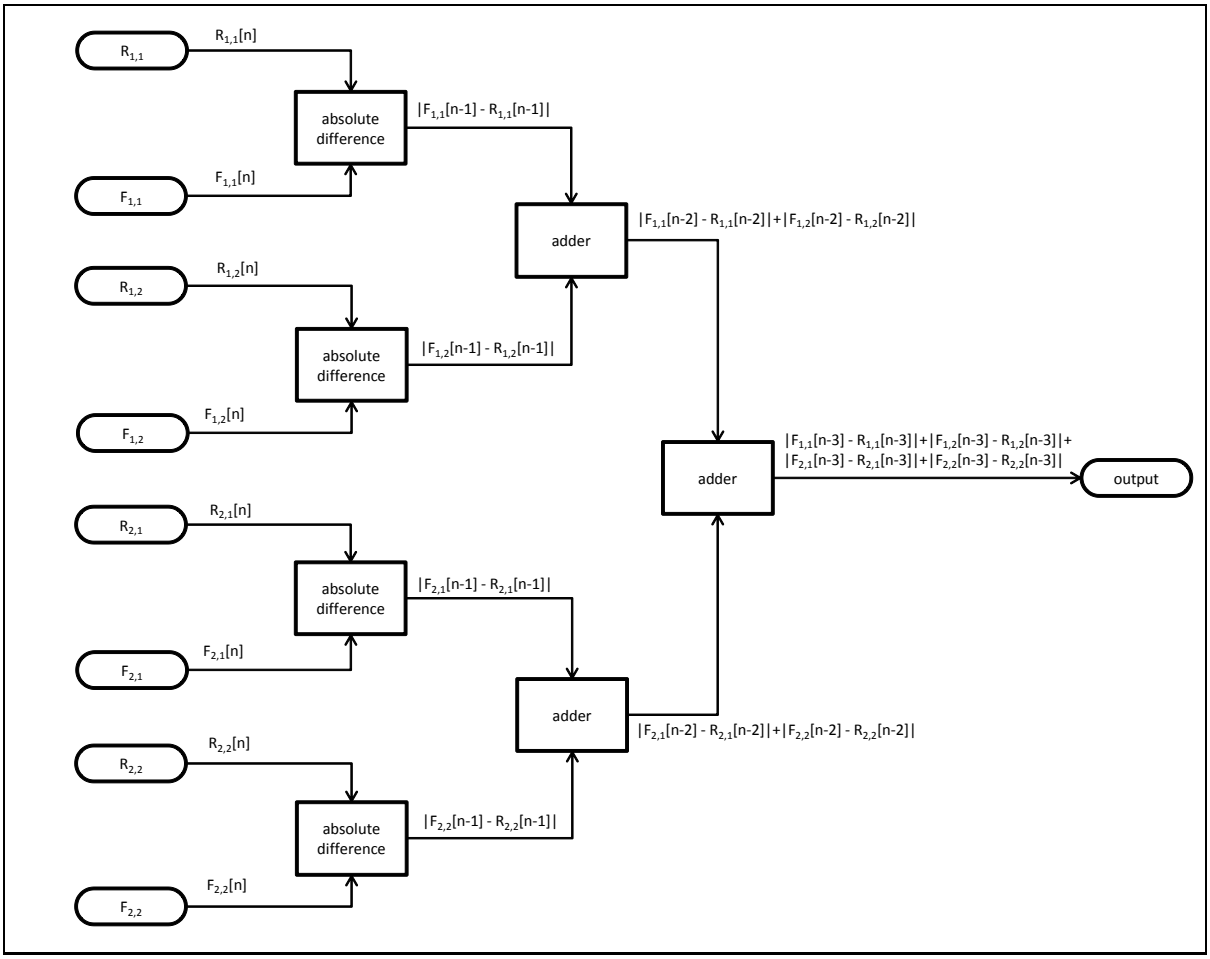

Figure 18 Annotated SAD Specification

A user may specify a pipelined design but is responsible for ensuring the consistency of the pipeline themselves.

Profiling

Every component has a profile which describes the characteristics of that component. Profiles provide details on cost and performance for each set of parameters that a component may have. These profiles are used to determine the system profile which includes the design criteria a designer is interested in. Profiles also contain two tags; one tag is common to all instances that perform the same function, the other is unique within this set to the particular component concerned. Restrictions on whether an instance can be multiplexed and input and output data types are stored here as well. The profile is a uniform interface between the library of components and the pre-processor so that unique components can be automatically incorporated into the optimisation stage following the pre-processor. Figure 19 shows a component with four inputs and two outputs. A typical profile for an FPGA implementation

of this component is also shown. Here, the term LUTs stands for a lookup table and an EM stands for an embedded multiplier both of which are types of resources present in FPGAs.

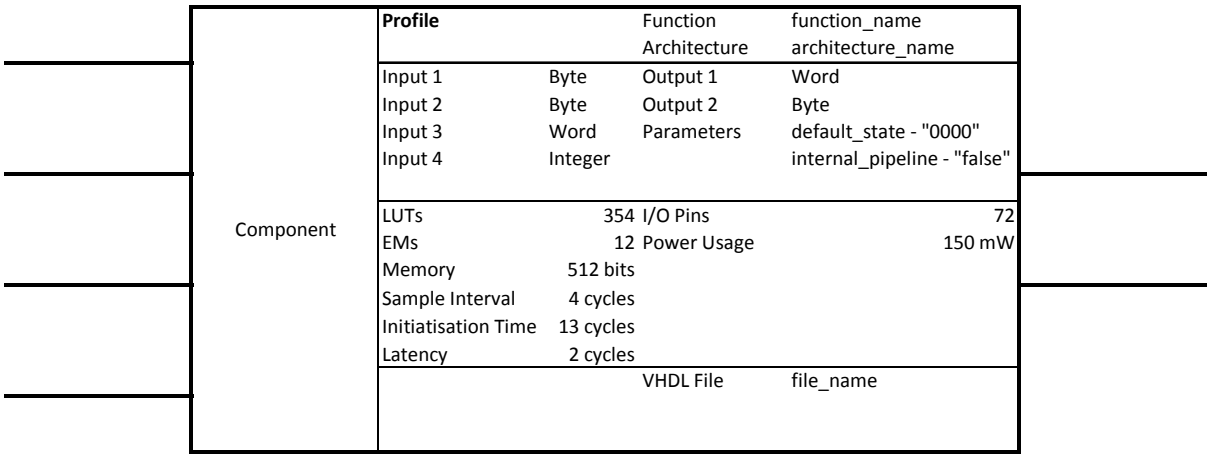

Figure 19 Example Component Profile

The function tag denotes either what a component does or its operation. This component is called "function_name" and all components with the same tag perform the same operation. The architecture tag denotes how a component performs its operation. Every architecture tag for a given function tag must be unique. The architecture tag here is given the description "architecture_name" but might also be labelled as "low-cost" or "high-performance" in a practical component to show the difference between a low-cost version of the operation and a high-performance version of the operation. These different architectures would have the same operation but different component profiles. "low-cost" if used must have only one occurrence in all components of function "function_name" but another function such as "divider" could also have an architecture called "low-cost". In this case, the "function_name" "low-cost" component would be a different component to the "divider" "low-cost" component. Each input and output has a type which is used to determine channel requirements for connecting components. These types are defined in the user library along with any type converters associated with them. Each component has a list of parameters it requires. These are a mechanism for supporting component templates or parameterised components. Each parameter has a name and value. The required parameters for each component vary between different components but a list of required parameters is available for each component. This can be obtained from the user library. In the example there are two parameters, a default state of "0000" and an internal_pipeline parameter that is set to false. The value associated with LUTs is the number of logic cells used by the architecture

for this component. The value associated with EMs shows the number of embedded multipliers. The memory value denotes how many bits of memory the architecture requires, and power value gives the power requirements of the architecture [**53**]. Switching power can be approximated using the simulator and input vectors or by utilising function stubs attached to components. Using switching power as a design criteria means a simulation is necessary and simulations are computationally expensive. I/O pins record how many inputs, outputs, and control pins are used by the component. Sample time is the number of cycles per sample this component takes to perform its function. The initialisation time is the number of cycles following a reset signal that must be waited before the component is ready to accept input samples. Using the clock enable signal allows the framework to place all other components on standby until every component is initialised. Latency is the delay in cycles between an input arriving and the component producing the associated output. A component has a VHDL file that describes the implementation of the component in a form that a third party compiler can use to program an FPGA device to perform this component's operation according to the component profile. A basic component is permitted to contain both behavioural and structural descriptions. A system prepared by the framework always consists of a number of connected components and is thus always described structurally. [**54**]

In the example the component finishes in 4 cycles, but by disabling the clock using the clock enable signal, the framework can increase this time. The example component takes 13 cycles to initialise following a reset signal. The example component takes 2 cycles to produce its output. Note also that although an output is produced in 2 cycles the example component must wait at least another 2 cycles before taking another input sample since the sample interval is 4 cycles.

System Scheduling

Each component calculates its output at a designated number of clock cycles after the sample interval begins and its output value is only known to be valid at the end of the sample interval.

When a reset signal is given, all components reset to their default states and then begin their initialisations. This is called the system's initialisation phase. When all components are initialised the initialisation phase ends. The length of the initialisation phase is determined by the component with the highest initialisation cycle requirement. If a component finishes before this time the component is placed in standby using the clock enable signal.

Once the initialisation phase completes, the operational phase begins. The operational phase lasts for N cycles. Once the operational phase ends a new operational phase begins. This process is repeated until a reset signal is received. During the operational phase each component completes all its assigned operations in a sequence, set during the optimisation stage of the design framework. Each operation is scheduled to be completed as soon as all the inputs are available and its assigned component is not performing another operation. These schedules are static and are determined at design optimisation time.

For scheduling purposes, at the start of the operational phase all component inputs must be ready, while at the end of the operational phase all component outputs must be ready. A component that does not have any channel components on its inputs will thus begin its operations on the first cycle of the operational phase. A component that has channel components on its inputs must wait until the channel components have prepared its inputs before beginning to process the associated operations. However, when the component is processing an operation the channel components can continue operating to ready the required inputs for the other operations in parallel with the component.

In Figure 20, a component is shown with a multiplexer and two storage registers. The multiplexer has a clock port, an output register, and a section of combinational logic which requires a latency of 1 cycle to be allowed for. The storage registers also have a latency of one cycle. Both the multiplexer and storage registers can accept inputs every cycle so they have a sample period of 1 cycle. For this example, the component has a latency of M cycles and a sample period of P cycles. Due to registering of outputs M is at least 1 cycle. Since two inputs will conflict if they arrive at the same time P must also be at least 1 cycle. The minimum latency of this pipeline stage path is M + P + 1 cycles. The sampling period of the system is P + max(M + 1, P). This is determined as follows.

The first input A takes one cycle to pass through the multiplexer, a further M cycles to be processed by the component and one cycle to pass through the storage register. This totals M + 2 cycles. Input B takes one cycle to pass through the storage register, and one cycle to pass through the multiplexer. B arrives at the component one cycle after A and must wait P – 1 cycles before the component is free. The component takes M cycles to process B. This totals M + P + 1 cycles. Since P is at least one and the latency is determined by the longest path, the latency of this system is M + P + 1 cycles. If the component must wait longer before

accepting another input (P > M + 1) then 2*P cycles must elapse before another input can be taken.

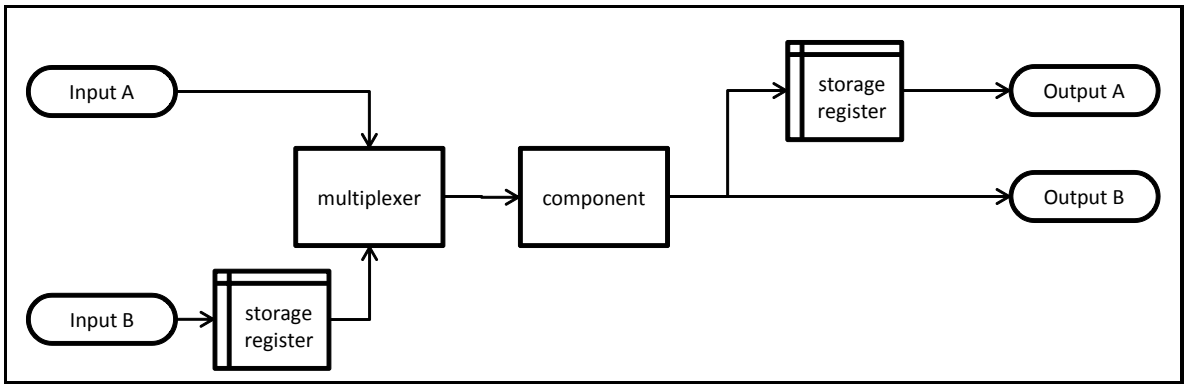

Figure 20 Example System

In a system with multiple paths the highest sampling interval determines the system sampling interval, X. Shorter paths are placed in standby using the clock enable signal. The latency of a system with Y pipeline stages is X*Y cycles. For a type converter with a latency of Q and a minimum sampling period of R and a component with a latency of M and a minimum sampling period of P then in components that are not shared, the latency of the component and its channel determines the latency of the pipeline stage path (Q + M), while the throughput cycle count is max(Q + M, P, R) cycles.

3.5.3 STR Example

The FDE extension represents a natural way of specifying the parameter estimation circuit of the self-tuning regulator [**20**]. The parameter estimation circuit is described in matrix form in Equation (5) as:

$$\begin{aligned}
&\varepsilon(t) = y(t) - \varphi^T(t-1)\hat{\theta}(t-1) \\
&K(t) = P(t-1)\varphi(t-1)\left(\lambda + \varphi^T(t-1)P(t-1)\varphi(t-1)\right)^{-1} \\
&\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\varepsilon(t) \\
&P(t) = \frac{\left(1 - K(t)\varphi^T(t-1)\right)P(t-1)}{\lambda}
\end{aligned} \tag{5}$$

where t is the current sample, $\hat{\theta}(t)$ is the estimated process parameters, λ is a forgetting factor, $K(t)$ is the Kalman gain, $\varepsilon(t)$ is the error in predicting the signal $y(t)$ one step ahead based on the estimate $\hat{\theta}(t)$ and measurements $\varphi(t-1)$, and $P(t)$ is the error covariance [**20**].

Four plant parameters are involved. This is represented in the FDE as the following set of finite difference equations shown in Equation (6). Here, y(t) is the plant output, u(t) are measurements made on the plant, and a1, a2, b0, and b1 are plant parameters.

```
//this line calculates ε(t) which is the first equation
error[t] = y[t] - (y[t-1]*a1[t-1] + y[t-2]*a2[t-1] + u[t]*b0[t-1] + u[t-1]*b1[t-1]);

//these lines calculate the second equation
KalmanPre1[t] = P11[t-1]*y[t-1] + P12[t-1]*y[t-2] + P13[t-1]*u[t] + P14[t-1]*u[t-1];
KalmanPre2[t] = P21[t-1]*y[t-1] + P22[t-1]*y[t-2] + P23[t-1]*u[t] + P24[t-1]*u[t-1];
KalmanPre3[t] = P31[t-1]*y[t-1] + P32[t-1]*y[t-2] + P33[t-1]*u[t] + P34[t-1]*u[t-1];
KalmanPre4[t] = P41[t-1]*y[t-1] + P42[t-1]*y[t-2] + P43[t-1]*u[t] + P44[t-1]*u[t-1];

A = forget + KalmanPre1[t]*y[t-1] + KalmanPre2[t]*y[t-2] + KalmanPre3[t]*u[t] + KalmanPre4[t]*u[t-1];

Kalman1[t] = KalmanPre1[t]/A;
Kalman2[t] = KalmanPre2[t]/A;
Kalman3[t] = KalmanPre3[t]/A;
Kalman4[t] = KalmanPre4[t]/A;

//this implements the third equation
a1[t] = a1[t-1] + Kalman1[t]*error[t];
a2[t] = a2[t-1] + Kalman2[t]*error[t];
b0[t] = b0[t-1] + Kalman3[t]*error[t];
b1[t] = b1[t-1] + Kalman4[t]*error[t];

//this implements the final equation, forget is used to reduce the effect of previous signals //over time
P11[t] = (P11[t-1] - Kalman1[t]*KalmanPre1[t])/forget;
P12[t] = (P12[t-1] - Kalman1[t]*KalmanPre2[t])/forget;
P13[t] = (P13[t-1] - Kalman1[t]*KalmanPre3[t])/forget;
P14[t] = (P14[t-1] - Kalman1[t]*KalmanPre4[t])/forget;

P21[t] = (P21[t-1] - Kalman2[t]*KalmanPre1[t])/forget;
P22[t] = (P22[t-1] - Kalman2[t]*KalmanPre2[t])/forget;
P23[t] = (P23[t-1] - Kalman2[t]*KalmanPre3[t])/forget;
P24[t] = (P24[t-1] - Kalman2[t]*KalmanPre4[t])/forget;

P31[t] = (P31[t-1] - Kalman3[t]*KalmanPre1[t])/forget;
P32[t] = (P32[t-1] - Kalman3[t]*KalmanPre2[t])/forget;
P33[t] = (P33[t-1] - Kalman3[t]*KalmanPre3[t])/forget;
P34[t] = (P34[t-1] - Kalman3[t]*KalmanPre4[t])/forget;

P41[t] = (P41[t-1] - Kalman4[t]*KalmanPre1[t])/forget;
P42[t] = (P42[t-1] - Kalman4[t]*KalmanPre2[t])/forget;
P43[t] = (P43[t-1] - Kalman4[t]*KalmanPre3[t])/forget;
P44[t] = (P44[t-1] - Kalman4[t]*KalmanPre4[t])/forget;
```

(6)

For the final equation that calculates P44[n] the system specification model of this section is described by the graph in Figure 21. This system was implemented without pipelining so that the entire system would compute its outputs before a new sample could be taken. This step was taken to make writing the equations for the FDE extension simpler without having to allow for the effect of pipelines introducing signal delays. In this framework pipelining is compulsory unless removed by altering the framework program code.

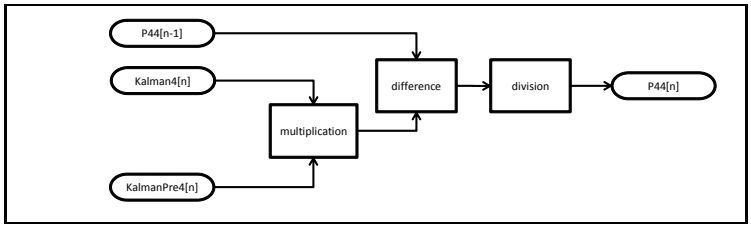

Figure 21 P44[n] Sub-circuit

Figure 22 shows the number of the clock cycle after the start of the sample interval in which the component begins its calculation. These clock cycles are in round brackets directly underneath each the component function description. Each component here takes one clock cycle to compute its result. Since the final division takes one cycle to complete, this section of the system takes a total of three cycles to complete.

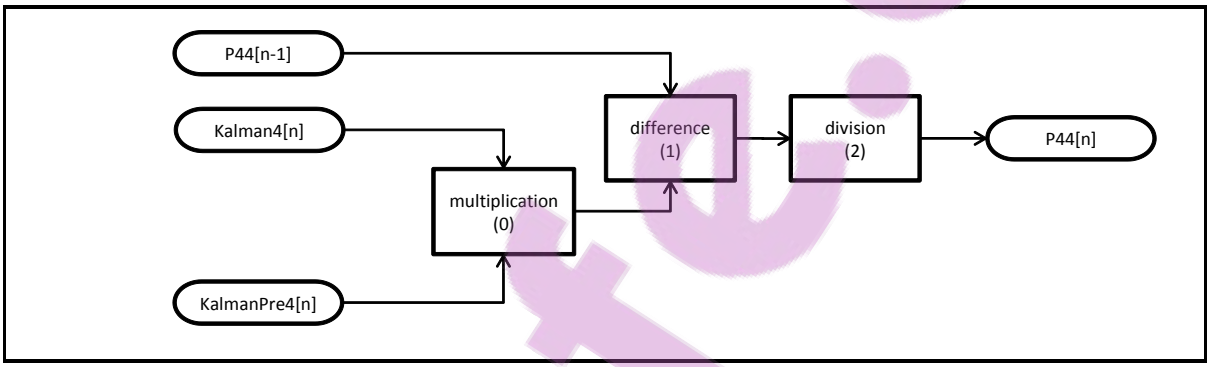

Figure 22 Time Annotated P44[n] Sub-circuit

This section when profiled by itself for a 16-bit implementation has the following properties as outlined in Figure 23 for an Altera FPGA device [**12**].

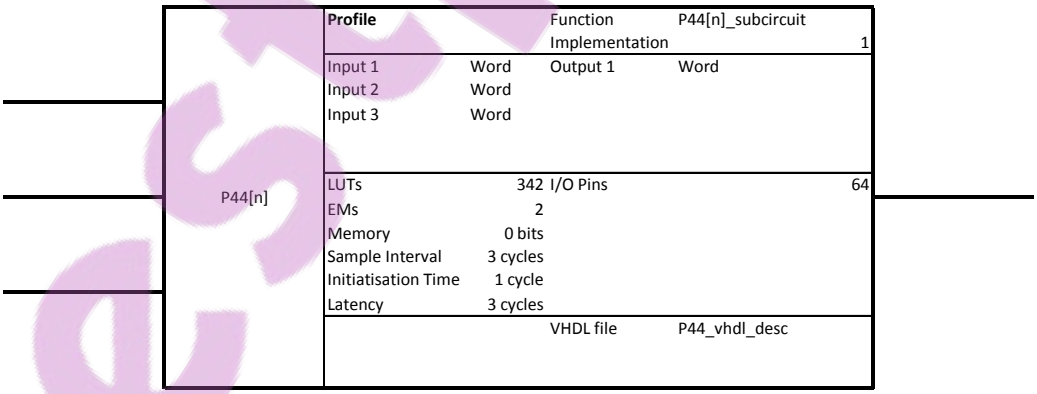

Figure 23 P44[n] Component

16-bit words are used here, which with three inputs and one output means a total of 64 input/output pins.

3.5.4 SAD Example

Figure 24 shows a system specification model for a SAD function with a window size of 2 by 2. This model has three pipeline stages, which are separated with the dashed lines. The output for the current sample will appear three sample periods after the arrival of the current sample.

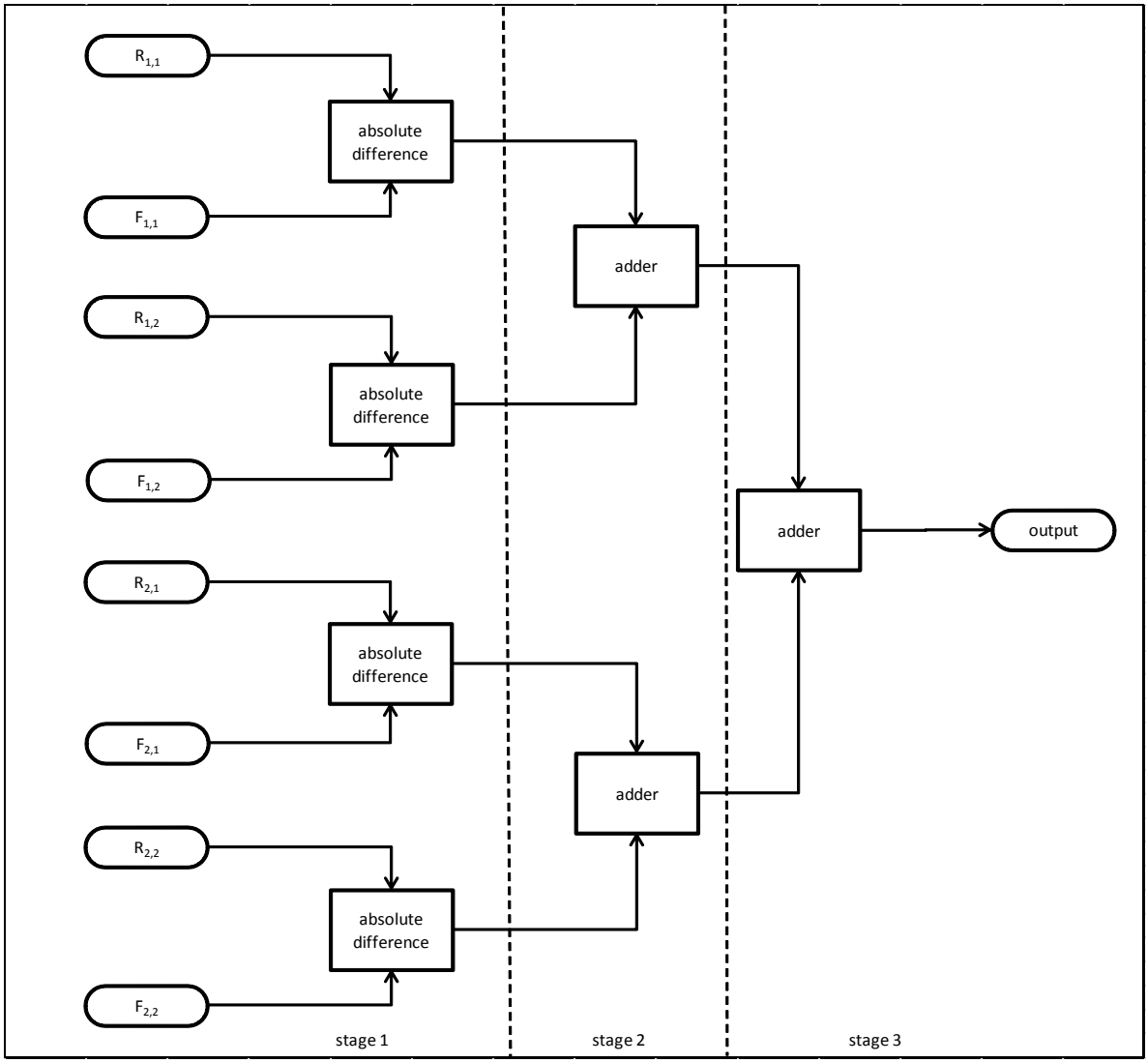

Figure 24 Pipeline Stages in 2 by 2 SAD function

There are a number of ways to allocate resources and partition this design; two possibilities are given in Figures 25 and 26. In Figure 25 every operation has its own component. In Figure 26 each absolute difference component has two operations bound to it. In Figure 26, as two operations are bound to each absolute difference component, multiplexers and storage registers are required to supply the inputs to each component and record intermediate outputs.

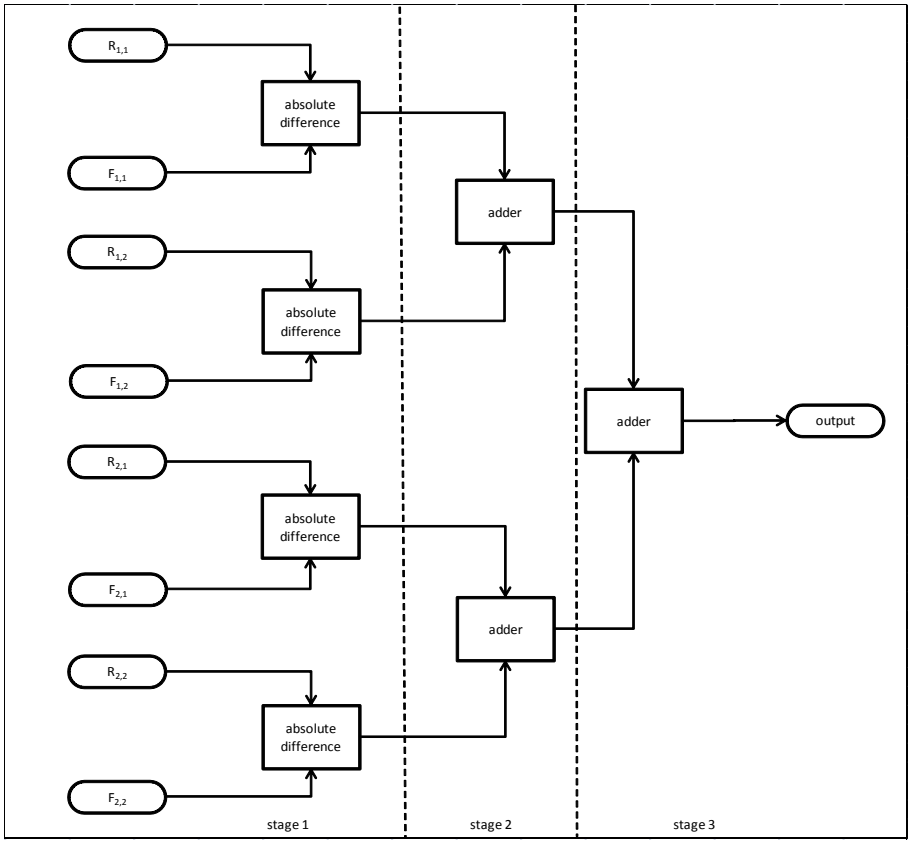

Figure 25 SAD function Implementation A

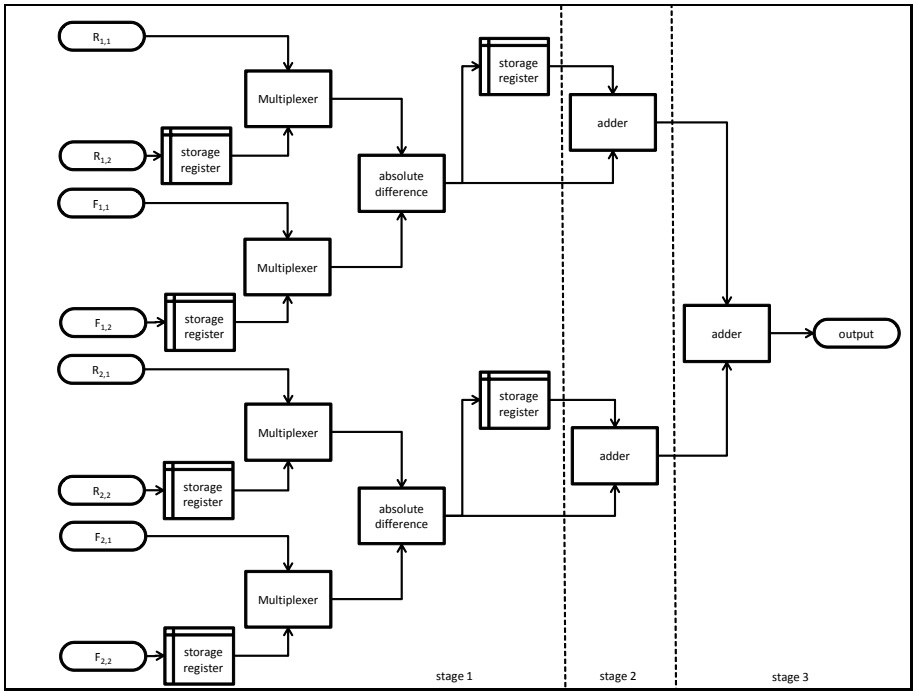

Figure 26 SAD function Implementation B

These implementations are functionally equivalent, but when they are scheduled they have different sampling rates. Implementation A and B still have three pipeline stages as before,

but the first stage has changed. Implementation A has four absolute difference functions in parallel in the first stage, two adders in parallel in the second stage, and one adder in the third stage. Each component takes one cycle to complete its operations so each pipeline stage takes one cycle to complete. The system can therefore receive a new sample every clock cycle with the results appearing three clock cycles later. Implementation B has an identical arrangement in the last two stages but differs in the first stage. In the first stage samples must pass through a multiplexer, an absolute difference component and a storage register. This means the first stage takes three cycles to complete. This limits the system in Figure 25 to receiving new samples only every three clock cycles with a latency of nine clock cycles (3 pipeline stages at a system setting of 3 clock cycles per stage).

When profiled for a monochromatic system with 8-bits per pixel using an Altera FPGA [**12**], the implementation from Figure 25 becomes a new component as shown in Figure 27 for the user library, while the implementation from Figure 26 becomes the component shown in Figure 28.

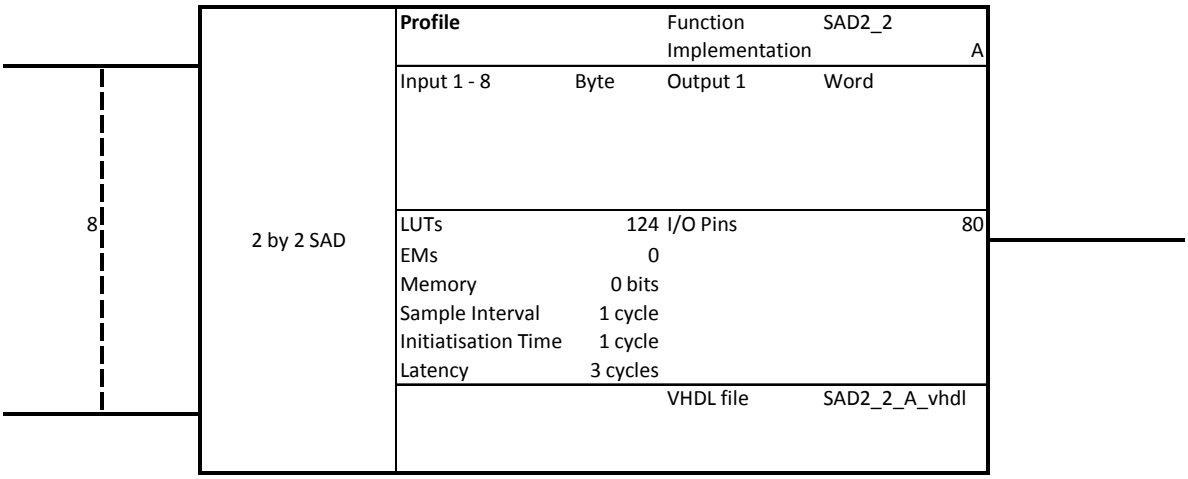

Figure 27 SAD Component A for Implementation A

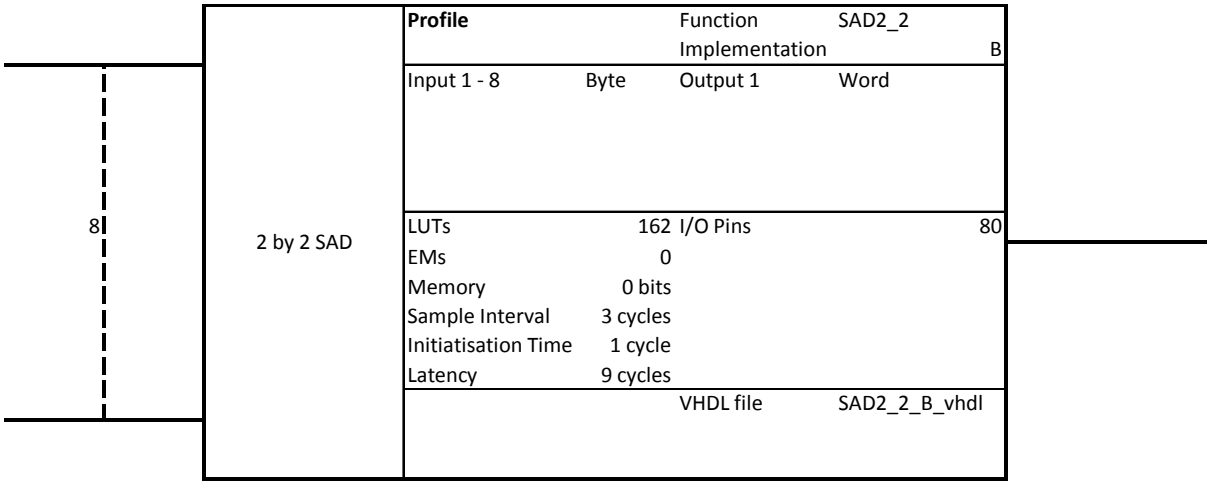

Figure 28 SAD Component B for Implementation B

As can be seen from the component profiles from Figure 27 and Figure 28, component B is a poor choice. The cost in terms of logic cells (LUTs) and the latency are both higher and the sampling rate (clock frequency/sample interval) is lower. In practice, component B would not be added to the user library. Component A can also be considered a system. As a system, A has the same profile as its component version.

3.5.5 EKF Example

The Extended Kalman Filter is used in estimating the plant parameters of non-linear systems. For a plant where the plant state is x, the process noise is w, the measurements of plant outputs are y, and the measurement noise is v, then the system is given in Equation (7) as:

$$\begin{aligned} x_{k+1} &= f_k(x_k) + w_k \\ y_k &= h_k(x_k) + v_k \end{aligned} \tag{7}$$

For the Extended Kalman Filter, F, H, R and Q are as described as:

$$\begin{aligned} F(k) &= J_{f_k}\left(x(k \mid k)\right) \\ H(k+1) &= J_h\left(x(k+1 \mid k)\right) \\ R_k &= E\left(v_k v_k^T\right) \\ Q_k &= E\left(w_k w_k^T\right) \end{aligned} \tag{8}$$

As such, the Extended Kalman Filter is described in as two stages below in Equation (9) labelled; Prediction, and Filtering [**55**] [**56**].

Prediction

$$\hat{x}(k+1|k) = f_k(\hat{x}(k|k))$$
$$P(k+1|k) = F(k)P(k|k)F^T(k) + Q(k)$$

Filtering

$$\hat{x}(k+1|k+1) = \hat{x}(k+1|k) + K(k+1)[y_{k+1} - h_{k+1}(\hat{x}(k+1|k))]$$
$$K(k+1) = P(k+1|k)H^T(k+1)[H(k+1)P(k+1|k)H^T(k+1) + R(k+1)]^{-1}$$
$$P(k+1|k+1) = [I - K(k+1)H(k+1)]P(k+1|k)$$

(9)

This algorithm features a number of matrix multiplications. For the matrices in Equation (10) the system specification for a_{11} is given in Figure 29.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \quad \textbf{(10)}$$

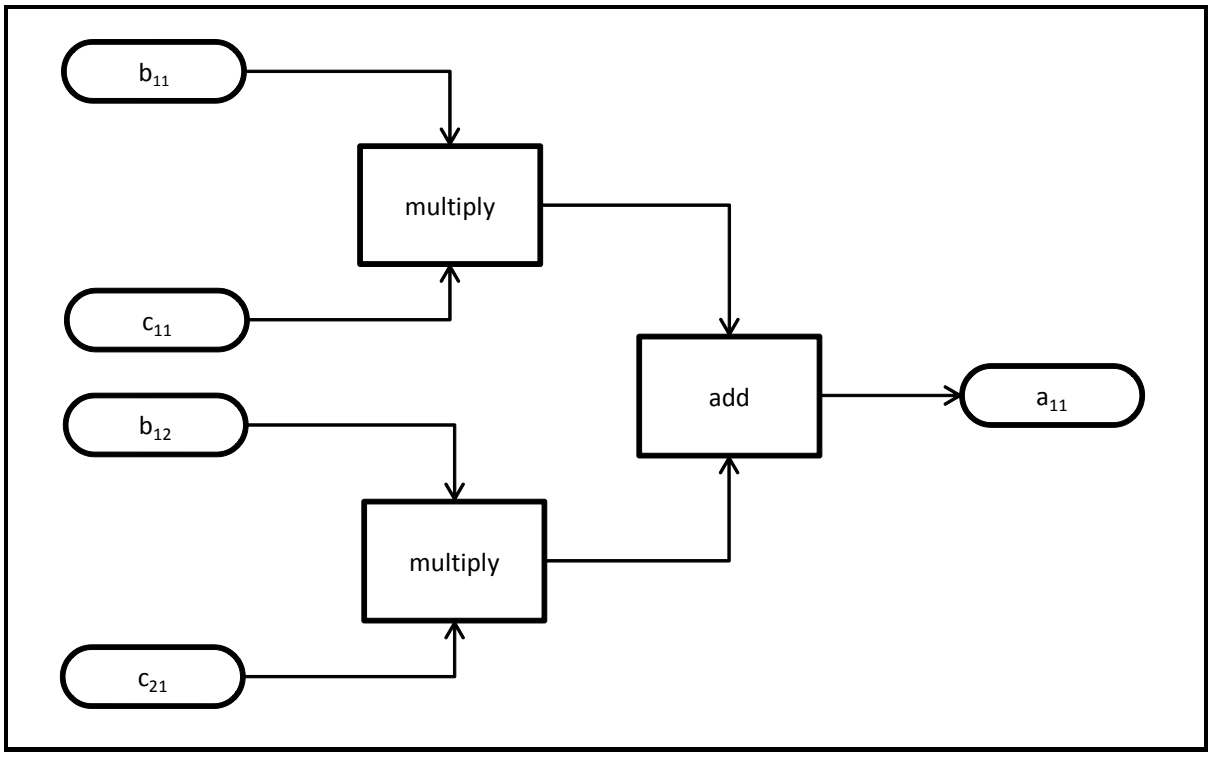

Figure 29 a_{11}

An example component that implements a_{11} is given in Figure 30.

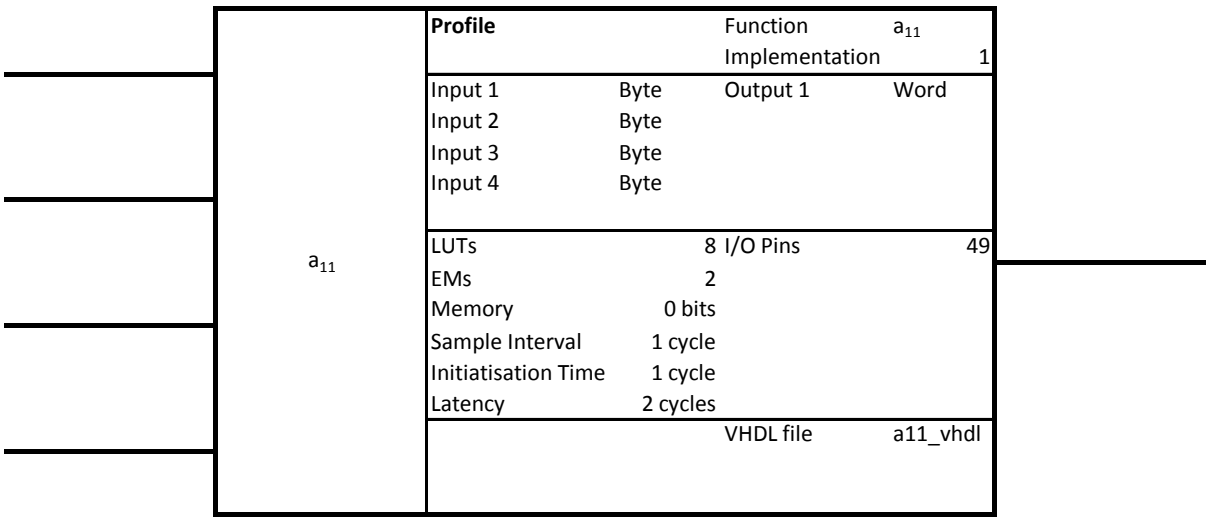

Figure 30 Component for a_{11}

3.6. Chapter Summary

Optimisation of designs is one of the key issues of this research and a limitation on the power of any automated design tool. This chapter has discussed the design framework and what optimisation is required. The framework addresses what can be represented and how it can be represented in terms of both a specification and an implementation. Chapters 4 - 6 address the algorithm that produces an implementation from a specification, with Chapter 7 providing some case studies of practical significance.

Producing such an implementation, requires a measure of the quality of an implementation in order to pick the best and a degree of certainty that proposed implementations are good solutions, compared to what other design tools can produce. The framework discussed in this chapter is used to produce a profile that describes the physical characteristics of a design implementation. These physical characteristics are used in measuring the quality of a design implementation. Chapter 7 discusses how to pick the best solutions and compares different algorithms used in multi-objective problems with reference to the parameterised nature typical of design problems.

Chapter 4. Genetic Algorithms

This chapter presents a general account of genetic algorithms. Firstly their algorithm flow is given, followed by their principal operators and example implementations of those operators. Lastly, the issues of mate selection, selection of search parameters, and balancing exploration and exploitation in search algorithms are discussed.

4.1. Overview

Genetic algorithms are effective and robust for searching trade-offs in multi-objective combinational problems [**15**][**57**]. They possess an inherent parallelism in their search that allows multiple patterns to be processed for each individual considered [**15**]. The majority of the time taken by a genetic algorithm is in evaluation, which may be performed in parallel, since the evaluation of one individual is independent of any other individual. Results from previous evaluations of individuals may be used to avoid re-evaluating previously tested individuals. Genetic algorithms utilise payoff functions called either fitness or cost functions which guide the search process. As payoff functions are used, the objective functions need not be differentiable but preferably good solutions should exist in clusters.

Genetic algorithms are used to perform the optimisation stage of the proposed design framework from Chapter 3 because of their robustness and the ability to make modifications to operators and solution encoding, without requiring problem domain knowledge. Currently genetic algorithms are commonly used in design optimisation work over a range of different applications. These different applications have differing requirements which genetic algorithms are able to meet because of their abstraction of problem representation [**15**].

This chapter presents an overview of genetic algorithms (indicating where the proposed algorithm differs), a brief historical account of common methods and some issues, specific to genetic algorithms. A flow diagram is given for a typical genetic algorithm. The three most commonly identified processes are given, with some historical accounts of their implementation. This chapter also gives a selection of issues in genetic algorithm usage that are important in the development of the operators for the proposed algorithm given in Chapter 5.

Many genetic algorithms that are improvements on the SGA exist for solving complex problems. There are coarse-grained genetic algorithms where subpopulations evolve on

separate computers [**58**], combinations of simulated annealing and genetic algorithms [**59**], fuzzy logic control of genetic algorithms [**60**], combinations of linear programming and genetic algorithms [**61**], quantum genetic algorithms [**62**], compact genetic algorithms for embedded applications [**63**], species selection mechanisms [**64**], and multi-population co-evolutionary algorithms [**65**].

4.2. Algorithm Flow

Figure 31 presents the algorithm flow for a typical Genetic Algorithm. The process begins with a problem which is supplied to the genetic algorithm. The genetic algorithm is initialised. This usually involves randomly generating the first population. A genetic algorithm at any time has a set of solutions which is called a population. Each solution is called an individual. During initialisation the first population is prepared.

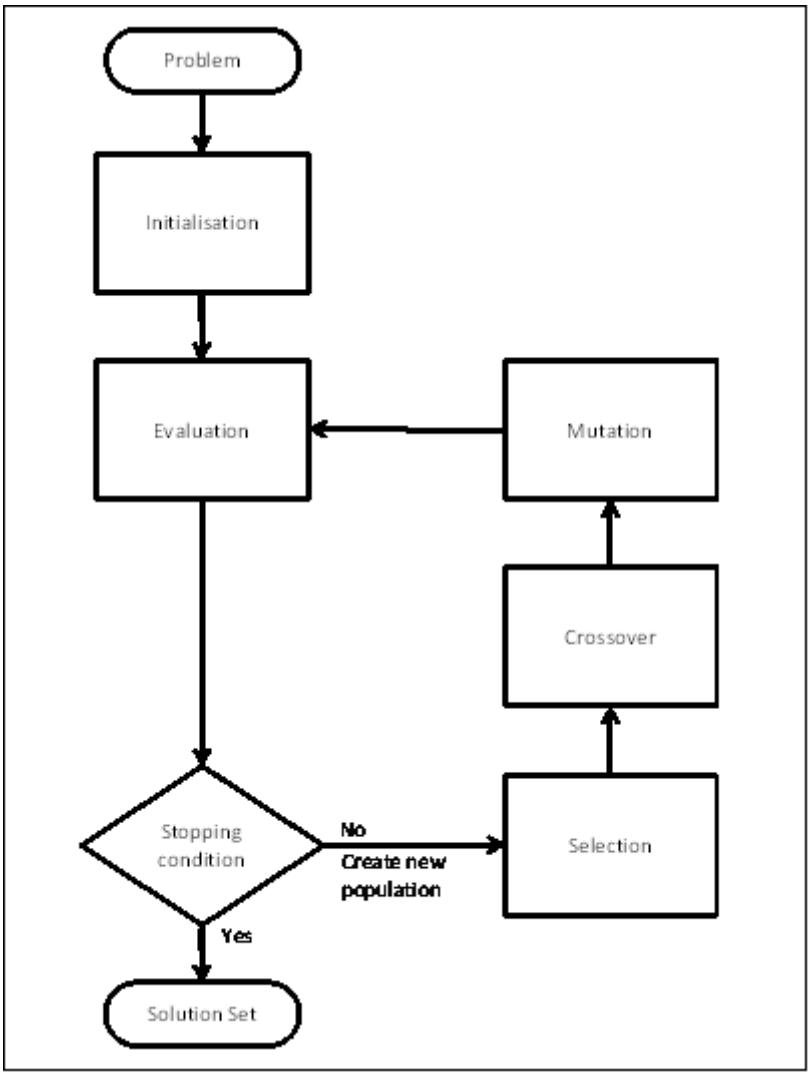

Figure 31 Genetic Algorithm

An individual or solution is represented by both a genotype and a phenotype. A genome is the complete genotype of an individual. Figure 32 shows the genome with its parts labelled. A genome is a set of chromosomes. A gene is a location on a chromosome. An allele is the value held in a gene. Each gene may have different limitations on the alleles it may take.

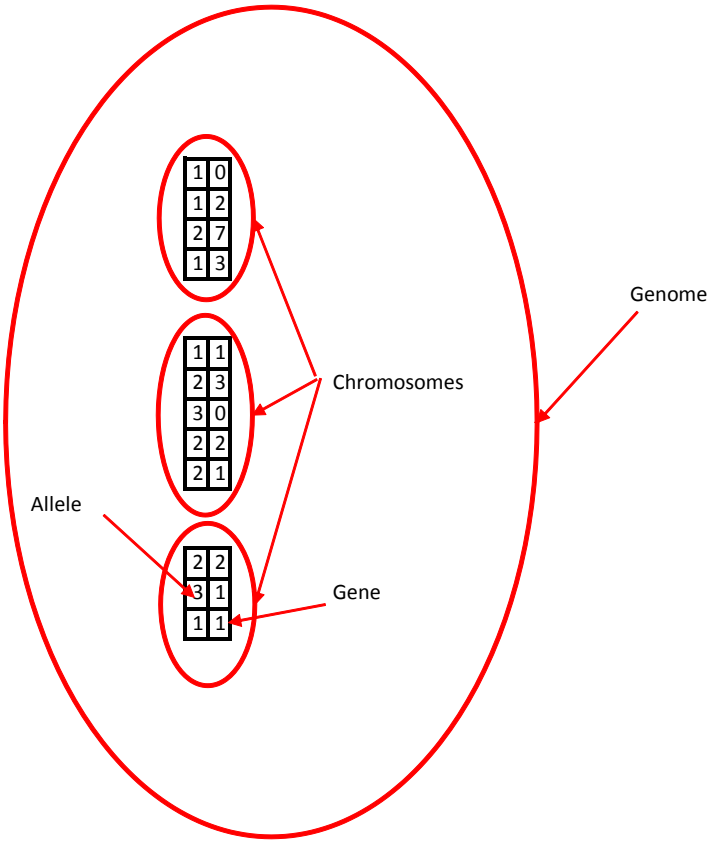

Figure 32 Parts of a Genome

A phenotype however is the overall functioning ability of an individual. In a genetic algorithm this is the individual's set of objective values. A phenotype is determined by the genotype of an individual. The mapping from phenotype to genotype is not unique as many genotypes can produce the same phenotype. The mapping from genotype to phenotype is however unique as a given genotype can have only one associated phenotype. Table 2 shows an example of a "Punnett square" for a non-unique mapping due to allele dominance. In genetic algorithms there can be other causes for such non-unique mappings. Capital letters show dominant alleles while lower case letters show recessive alleles. The genotypes presented are RR, Rr, rR, and rr. If R is the dominant allele for red and r is the recessive allele for white then the phenotypes are red and white. The red boxes show red phenotypes while the white boxes show white phenotypes. The Punnett square indicates that the red

phenotype can be caused by multiple genotypes while the white phenotype can be caused by only one genotype. Also given is that a genotype has only one associated phenotype.

Table 2 Punnett Square

	R	r
R	RR	Rr
r	rR	rr

During evaluation all the phenotypes for each individual in the current population are determined from their genotypes. This mapping relation is treated as a "black box" function by the genetic algorithm so there are no requirements on function type.

Typically a genetic algorithm has a number of stopping conditions. These stopping conditions can be: a given number of generations, evaluating a given number of individuals, a given time, when no significant progress has been made for a given number of generations, or when the algorithm has converged. Once this stopping point is reached the genetic algorithm supplies the set of the best solutions it has found.

If the stopping condition has not been met then a new population must be created. New populations are made from the current population via selection, crossover, and mutation operators. The selection operator selects individuals from the current population and copies them to an intermediate population. In order to assist selection a payoff function called a fitness function is calculated based on the individual's phenotypes. Selection gives preference to individuals that are fitter and more diverse than their rivals. The crossover or mating operator is where parents are selected from the intermediate population and replaced with their children. Their children are individuals made of genes from their parents. Generally a child inherits equal quantities of genes from each parent. The selection of these genes from parents is usually random. The mutation operator is where every gene in the intermediate population has a chance to mutate or change its current allele. The new allele is chosen randomly from the other possibilities. The mutation operator is applied to the intermediate population to create a new population. When the algorithm progresses to evaluation this new population becomes the current population.

In this thesis, new selection, crossover, and mutation operators for use in genetic algorithms are proposed. The proposed selection operator attempts to estimate the probability an allele is present in a solution in the pareto-optimal set and allocates time proportional to this probability to pursuing individuals with that allele. The proposed crossover operator

incorporates species into the genetic algorithm in order to identify key genetic sequences present in subsets of the population. This operator then focuses on refining these key sequences. The proposed mutation operator allows the chance of mutation to be adaptively controlled during evolution and is tied to the other two operators through an individual's genome.

These changes have been made to (where possible) eliminate the need for search parameters and mitigate their sensitivity. When all three operators are used, the only remaining sensitive parameter is the population size. Other search parameters may be left at their default states for relatively little loss in performance. As with most genetic algorithms, performance increases with population size and so the largest feasible choice is usually the best.

4.3. Genetic Operators in Genetic Algorithms

In this section the features of genetic algorithm operators used in existing genetic algorithms are presented. These are divided into selection, crossover, and mutation.

4.3.1 Selection

Selection is responsible for the replication of desirable genes in a population. This is usually accomplished by selecting fit individuals more often. Fit individuals have a higher chance of being copied to the next generation. This higher chance means that successive generations tend to contain more copies of fitter individuals. The copying of these individuals increases the proportion of genes in the population containing their alleles.

Provided the epistasis of genes is not too significant by increasing the proportion of these alleles, the chance of finding better solutions is also increased. In the case of extreme epistasis there is little that can be done to guide search processes. These situations involve finding a small number of isolated peaks. An example is factorising large semi-prime numbers. Semi-prime numbers are numbers with exactly four factors. These factors are 1, itself, and two prime numbers. Large semi-prime numbers are used in cryptography because factorising them is difficult.

How individuals are determined to be fit for this purpose and how they are replicated determines the selection scheme used. Individuals closer to the pareto-optimal front are fitter. The best cases are when the individual has no other individuals that dominate it. In this case it is on the pareto-optimal front. Because an individual is close to the front may not

lead to the individual having a higher chance of selection. A front often contains many solutions. If only fitness were considered, then stochastic variations would make the likelihood of convergence on a small region of the front too significant. Diversity is also considered in how individuals are replicated. After checking an individual for fitness an individual is also checked for how different it is from other individuals found. This comparison helps spread individuals out so that a larger area of the pareto-optimal front is covered.

Objective functions are the performance characteristics for measuring fitness. When there is more than one characteristic, then other mechanisms are employed to determine a fitness value. Several techniques use the pareto-dominance rank [**15**] or a derivative thereof. The pareto-dominance rank is determined as follows. All the non-dominated solutions are assigned a rank of 1. These are then removed from the population. Of the remaining individuals all the non-dominated solutions are assigned a rank of 2. These are then removed. Of the remaining individuals, the next set of non-dominated solutions are assigned a rank of 3 and the process is repeated, assigning successively higher ranks until every individual has been assigned a rank. Consequently all rank 1 individuals are better than rank 2. All rank 2 individuals are better than rank 3 and so forth [**15**]. NSGA-II [**16**] uses the pareto-dominance rank in deciding which individuals to copy to the next generation.

During replication, an individual with a higher fitness has a higher chance of being copied to the next generation. Two established techniques for deciding which individuals to copy are Roulette-Wheel Selection [**15**] and Tournament Selection [**66**]. Roulette-Wheel Selection assigns a probability of selection proportional to the contribution the individual makes to the total sum of fitness values from the population. The name derives from the process being similar to the spinning of a Roulette-Wheel. However for a genetic algorithm the areas of the wheel are divided into different sizes. The probability of stopping in one region depends on how much of the total area that region represents. The regions are made proportional to the amount of the total fitness of the population that each individual contributes. In Tournament Selection N individuals are randomly chosen from the population and the best M of these N individuals are selected to be copied to the next generation. If a tie occurs then one of the individuals is chosen randomly and that individual is copied to the next generation.

It is not uncommon to copy the best X individuals into the next generation automatically. This is called Elitism [**15**]. Elitism preserves the best known solutions so that they are not

lost in subsequent generations and has been implicated in improvements in evolutionary multi-objective optimization [**67**]. This benefit however has its costs. The faster convergence means a lower running time but can also mean premature convergence to a sub-optimal solution set.

Some methods rely solely on Elitism in selection but still prove to be highly successful. NSGA-II [**16**] uses only Elitism in its selection. It copies the best ranks from its parent and child populations into its next generation. In the case of a tie for a rank it copies those individuals most distant from any other individuals while giving favour to those individuals at the extremities of the phenotypic space.

4.3.2 Crossover

The common characteristics of any crossover method are that they attempt to combine the qualities of two or more parents to create children with better qualities. Crossover methods differ in how they select parents, how many parents they select, how genes are recombined to create children, how many children are created, and how parents and children are incorporated into the next population. One simple and common approach is that two parents are selected using the same process for selection as in the selection operator. Half the genes from each parent are then used to create one child that is added to the next population. Any vacant places in the population are filled with individuals from the same source as the parents.

Two example crossover operators are K-Point crossover and uniform crossover. K-point crossover is used where there are clearly identifiable building blocks in a problem. Uniform crossover is used for large problems where no easily identifiable ordering for genes is apparent. Large problems can prohibit the use of inversion due to increases in problem complexity. These factors are further explained in the following subsections.

K-Point Crossover

K-point crossover divides individuals at k points along the chromosome and rearranges segments among the individuals to create children. An important aspect of K-Point Crossover is that adjacency of genes is more likely to be preserved than in Uniform Crossover. It is used when a problem is known to have substructures or “Building Blocks” [**15**] involving genes that can be identified as related or in cases when Inversion [**15**] is implemented. Inversion reorders genes on chromosomes without changing their function.

Inversion increases problem complexity by requiring the genetic algorithm to both search for good solutions and good gene orderings. In larger problems with high numbers of genes the additional search requirements can prove taxing. While it is common for K-Point Crossover to be applied to two individuals, there is no restriction that more than two individuals could not be used.

To illustrate this procedure consider two individuals A and B (given in Table 3).

Table 3: Parents A and B

A	1	1	1	1	1	1	1	1
B	2	2	2	2	2	2	2	2

Individual A and Individual B crossed under 2-Point Crossover might yield the following in Table 4.

Table 4: Children A and B after 2-Point Crossover of Parents A and B

A	1	1	1	2	2	2	2	1
B	2	2	2	1	1	1	1	2

Here individuals A and B have been divided between locus 3 and 4, and locus 7 and 8. The segments between these divisions have been interchanged to generate child A and B.

Uniform Crossover

Uniform crossover swaps equal proportions of genes amongst parents to create children. If for example two parents with eight genes produce children then each child will get 4 (8/2) genes from each parent but those genes do not have to be adjacent as in K-Point Crossover. For example, given the parents A and B in Table 3 then after uniform crossover one possible outcome is given in Table 5. Each child has four genes from each parent but those genes are not adjacent.

Table 5: Children A and B after Uniform Crossover of Parents A and B

A	1	2	1	2	2	1	2	1
B	2	1	2	1	1	2	1	2

Uniform crossover is not subject to poor gene orderings. Given that the problems considered in this work are not readily divisible, it is difficult to determine how genes are related. While this remains true, K-point crossover requires inversion as the user does not have the required information to define gene order. Inversion requires additional modifications to allow for the use of species in the proposed crossover operator. These complications can be avoided by using uniform crossover.

To set the position of a gene on a chromosome requires familiarity with both the problem and the internal operations of evolutionary algorithms. In the problem domains considered in this thesis, there is little prior knowledge about the problem domain, meaning that position information is unavailable. This means that an algorithm must either adapt on the fly or ignore position. Given that for some of the problem domains for this work, where position yields no advantages, position is ignored. In design problems using the design framework presented in chapter 3 different designs have different problem structures. These structures are highly problem dependent. To take advantage of gene position would require each possible structure to be accounted for. This can be implemented adaptively but at the expense of increased problem complexity since the algorithm must both find a suitable ordering and good solutions to the problem presented. Since the design problem complexity is already high and advantages of gene position are limited to only some design problems, gene position is not utilised.

K-parent uniform crossover is one extension of uniform crossover for K parents. In this thesis this is implemented as follows. If A is a vector of natural numbers from 1 to N – 1 but in random order and A_i is the value in the ith position of this vector, $C_{i,j}$ is the jth gene on the ith child, and $P_{i,j}$ is the jth gene on the ith parent then C_{i,A_j} is given the value of $P_{(i+j) \bmod k, A_j}$. The modulus addition generates a unique shift for each gene within the new set of children with respect to the parents. This operation ensures that alleles are switched between parents as evenly as possible in order to generate the children. This procedure ensures bounds are

kept so that the procedure produces K complete children without any missing genes. When K is 2 this is identical to ordinary uniform crossover.

If K were 3 and three parents as given in each row of Table 6 were used, then

Table 6: 3 Parents

A	A	A	A	A	A	A	A
B	B	B	B	B	B	B	B
C	C	C	C	C	C	C	C

a possible outcome of 3-Parent Uniform Crossover is given in Table 7.

Table 7: 3 Children

A	B	C	C	B	B	A	A
C	A	B	B	A	A	C	C
B	C	A	A	C	C	B	B

4.3.3 Mutation

The common characteristics of any mutation method are that one or more alleles are randomly assigned a new value. This random assignment may be non-uniform in some cases such as evolutionary strategies that use Gaussian noise to mutate alleles. Two simple cases of different types of mutation are given. The first is a fixed rate mutation while the latter is an adaptive mutation operator.

A fixed rate mutation operator requires a search parameter that states the likelihood of a gene mutating. This likelihood can range from 0 to 100% but is usually between 3 and 5%. This operator is applied to every gene. A gene may either mutate or remain the same depending on chance and the mutation rate. If a gene is to mutate it takes a new allele from the other possible alleles for that gene.

An adaptive mutation operator may not require any search parameters but commonly requires additional genes to be added to an individual to encode the mutation rate. The alleles of these genes when decoded tell a genetic algorithm what mutation rate to apply to the individual. Since these alleles are also subject to the genetic algorithms operators, the mutation rate per individual will change over time. Again, depending on the mutation rate a gene may mutate

or remain the same. If the gene mutates it takes on a new allele. Typically in earlier generations mutation rates can climb to large values possibly approaching 100% but in latter generations these rates usually drop to about 1%.

4.4. Issues in Genetic Algorithms

This section discusses three important factors in the development of the proposed optimisation algorithm. Mate Selection determines how individuals in a population select mates. Search parameters are necessary for many search algorithms, but determining how to set these parameters is not necessarily trivial. Search algorithms must balance exploration with exploitation in searching a space of possible solutions.

4.4.1 Mating Selection

A common but limiting implementation is to use the selection operator to pick parents for crossover. Doing this however restricts the ability to separate these processes. A selection operator uses the phenotype to pick individuals. A crossover operator manipulates the genotype. When the relationship of phenotype to genotype is not unique individuals picked using the selection operator can have a significant chance of including conflicting allele patterns. When the crossover operator is applied to them they can yield sub-optimal results.

Active mating selection is where the chances of mating different individuals are not the same. Active mating selection includes assortative mating and inbreeding or outbreeding. Assortative mating can be positive assortative mating, or negative assortative mating. In positive assortative mating, individuals are more likely to mate if they share similar phenotypes for example, tall people choosing tall partners. Negative assortative mating is the opposite where individuals are more likely to mate if they have differing phenotypes for example, tall people choosing short partners. Inbreeding is where individuals are more likely to mate if they are related. Outbreeding is where individuals are more likely to mate if they are not related. [**68**]

Active mating selection is employed to select mates more appropriately to either promote diversity or refine an existing pattern. When promoting diversity negative assortative mating or outbreeding is used. This can disrupt key genetic sequences but has increased resistance to premature convergence. When refining an existing pattern positive assortative mating or

inbreeding is used. These tend to preserve key genetic sequences and make minor changes to assess their effect but can result in premature convergence.

4.4.2 Selecting Search Parameters

Typically, genetic algorithms require search parameters that determine how recombination is performed. In the case of a Simple Genetic Algorithm (SGA) [**15**] which has the same algorithm flow as in Figure 31 and several of its derivatives, a crossover rate and mutation rate must be given in advance. These control the probability of a pair of individuals being crossed or a gene changed.

The ideal values for search parameters are determined by stochastic processes and the problem the algorithm is applied to. This situation is not ideal as it requires knowledge about the solutions in order to find them. This knowledge is usually substituted by an estimate or replaced with adaptive schemes.

Adaptive schemes can avoid the need for tuning search parameters especially in cases where a priori information is insufficient. It is difficult for adaptive schemes to perform as well as ideally tuned conventional schemes. Ideal tuning is hard to achieve though. It is preferable in cases where an algorithm is to be applied to many different problems to use a scheme that does not require re-tuning for each problem. Given the different requirements of different problems it is also unlikely that an optimal tuning exists for all problems. Because of these factors the proposed operators were designed to be adaptive.

4.4.3 Controlling the Balance of Exploration and Exploitation

Exploration and exploitation are the two primary processes in search and optimization algorithms [**68**]. Exploration explores new regions of the solution space to find better solutions. Exploitation attempts to accelerate the search process using knowledge gained from exploration. The balance of exploration and exploitation is crucial to the success of a search or optimisation algorithm [**69**]. If too much time is spent in exploration the algorithm becomes too time consuming. If too much time is spent on exploitation then an algorithm is unlikely to cover many solution options and yields a poor result.

Determining the appropriate balance of exploration and exploitation requires the properties of the search space to be approximated based on incomplete knowledge. This approximation determines how exploitation is implemented. The implementation of exploitation governs its

interaction with exploration. This relationship determines the balance between exploration and exploitation. A better approximation yields a better balance. For example, the Ones problem is to maximise the number of ones in a binary string. Knowing that the function has little epistasis means that an exploitation process can be used where each position on the binary string is tested individually. An exploration process then tests a one or a zero at each location. An approximation that does not make use of the low epistasis in this problem will result in a longer running time and possibly poorer solutions.

When little information is present exploration is best. When significant information is present exploitation is best. However, determining how much information is present is often not possible. This information can be absent because the size of the solution space is unknown or the information held by each solution is not the same. Two possible scenarios are: two similar individuals that perform poorly, or two different individuals that perform poorly. Determining which scenario contains the most information may not be trivial. The two similar individuals indicate a common poor pattern with verification of two occurrences. The two different individuals indicate two poor patterns but with no verification.

One means of controlling the balance of exploration and exploitation is to use an adaptive heuristic function. Under an adaptive heuristic, an estimation of how much knowledge of the search space is made based on an assumed model. The heuristic then sets the balance of exploration and exploitation. A heuristic can also be evolved for difficult problems. The proposed approach was to add a species tag to individuals to control the crossover process, and additional genes to control the mutation process. These genes completely define how the proposed crossover and mutation operators are performed. The heuristic is that species contain one key genetic sequence, which is tagged by a species tag and used in the refinement process. For mutation rates, a high mutation rate is only sustainable when there is relatively little difference in fitness values in the population.

4.5. Chapter Summary

This chapter gives the algorithm flow of a general genetic algorithm. Typically, genetic algorithms consist of initialising the first population and then repeat a sequence of evaluating individuals, selection, crossover, and mutation until a stopping condition is achieved.

A brief historical account of existing genetic operators is given for common selection, crossover, and mutation operators. This includes pareto-dominance ranks, roulette-wheel

selection, tournament selection, elitism, K-point crossover, uniform crossover, K-parent uniform crossover, fixed rate mutation, and adaptive mutation. The proposed selection operator uses roulette-wheel selection and elitism. The proposed crossover operator is an extension of K-parent uniform crossover. A common adaptive mutation strategy of encoding the mutation rate into individuals is followed. These are given in Chapter 5.

Issues involved in genetic algorithms are given covering mate selection, search parameters, and the balance of exploration and exploitation. These issues are key concepts to the design of genetic operators which are given in Chapter 5. The proposed selection operator was developed because design problems frequently do not have fixed objective goals. For example, it is not necessarily known that the best design has X logic cells. The proposed crossover operator is designed to fix an issue with the identification of key genetic sequences and their combination by ensuring that each species is refining a single sequence. Traditional crossover operators can attempt to combine conflicting gene sequences yielding suboptimal results. The proposed approach first identifies a single sequence disturbed by some random variations and iteratively removes those variations. The proposed mutation operator was necessary because the optimal mutation rate to use during searching a space is dependent on search progress. A mutation rate tends to start small, increase for a limited time and then decay as the search progresses. This allows a search algorithm time to identify good values, search for better values, and then refine what has been found.

Chapter 5. Adaptive Speciation Genetic Algorithm (ASGA)

This chapter presents the proposed ASGA for use in the optimisation stage of the design framework given in Chapter 3. The ASGA is a type of genetic algorithm. Fundamental information on genetic algorithms may be found in Chapter 4. These changes were introduced because of the problem requirements of system design optimisation. Typically a designer must investigate what target costs and performance are feasible in large complex designs. A single design problem can have many niche designs each fulfilling different aspects of the problem to different degrees. In addition, computational resources are typically very limited with respect to the size of the design spaces involved.

5.1. Overview

Additional properties are proposed for the three genetic operators, to extend the functionality of a genetic algorithm. This leads to the ASGA model proposed in this thesis. The Selection operator uses the population average for each objective, to define what constitutes a good objective value. This information is then used to estimate the likelihood that alleles are present in the pareto-optimal front. The resources dedicated to pursuing each allele are determined by this likelihood. The Crossover operator attaches a tag gene to each individual that determines whom the individual mates with during crossover. This restricts mating to individuals with similar characteristics and is called positive assortative mating. If individuals with similar characteristics persist in a population for any length of time then there is a high degree of confidence that they possess desirable genetic sequences. The proposed operator is designed to filter out random variations in these sequences to determine the gene sequence having the most desirable performance. The proposed mutation operator encodes how subject to mutation that individual is onto the DNA of the individual. The proposed operator uses gray-code numbers to encode the number of genes to mutate. As will be seen, this is how an adaptive mutation rate is implemented in this work.

The proposed ASGA is easier to apply to problems as only the population size is required from a user. The population size should be set to the largest value feasible. In Chapter 6 different designs of ASGA are tested and a final implementation selected which is subsequently referred to as ASGA. This leaves only the initial mutation rate which if poorly chosen is adaptive and will correct itself. This means that the ASGA may be readily applied to problems that the user is unfamiliar with, without requiring many pilot runs to determine

how to set the algorithm search parameters. This ease of use is beneficial in design problems where there is a large variation in the types of problems encountered and each problem may require different design strategies.

The work in this thesis draws upon existing material and extends the genetic algorithm through three proposed operators that are evaluated in Chapter 7. Experiments conducted during the design of these three operators may be found in Chapter 6. The motivation for producing these three operators was to remove the need for search parameters and to mitigate the effects of poor choices on search results. In this work search parameters refer to the population size, crossover rate, mutation rate, number of generations, and scaling of fitness or cost functions required to implement a genetic optimisation process. The expected benefits of this work are a set of operators that produce a genetic algorithm with little or no search parameters required that still performs as well as other leading algorithms.

The proposed operators are less subject to poor selection of search parameter values and fewer parameters are required to be specified. The only required search parameter is the population size. This influences how much processing effort is to be spent on the problem. Because the adaptive operators will eventually converge, the population size should be set higher than for a typical genetic algorithm. The proposed operators converge when the search reaches the point where further improvement is extremely difficult to obtain. This is the point where substantial increases in processing time occur for relatively little gain in solution quality. For the proposed algorithm, to ensure it does not prematurely converge additional resources must be allocated each generation by using a higher population size for the search process. The proposed algorithm will complete optimisation in fewer generations. Other search parameters may be left at their default values for relatively no change over a variety of different problems.

Since the pareto-optimal front is unknown an adaptive scheme was chosen to control the balance of exploration and exploitation. There were three main goals in the development of the proposed algorithm. These were to determine which solutions were good, to find the key genetic sequences in these good individuals, and to explore sufficient solutions to effectively cover a problem. Selection was used to determine which solutions were good. Crossover was used to rearrange individuals to find out which alleles composed their key genetic sequences. Mutation was used to widen the coverage of solutions considered.

Speciation has been identified as a process of allowing genetic algorithms to more readily find multiple optima in multi-modal problems [**70**]. Each species tends to focus on different optima. Design work typically has multiple optima that are of interest. Two methods for using species in a genetic algorithm are the "tag" and the "island" models. In the Simple Subpopulation Scheme by Spears [**70**], a tag was used on individuals to denote which species they belonged to. During evolution this tag could be mutated to allow individuals to branch into new species or cross to another species allowing a certain level of adaptation in the operator. During crossover, individuals could only mate if they belonged to the same species. In Island Model Genetic Algorithms each island is essentially a species. Migrations allow for transfers between islands or species, but usually an island or species size is fixed. The migration policy is also usually fixed.

5.2. Selection

5.2.1 Development Goals

A new selection strategy uses averages to determine whether an objective value is desirable and to estimate the probability that the alleles are desirable. Under this method any above average fitness objective, or below average cost objective, is flagged as desirable. Basing decisions upon an average gives a higher resistance to stochastic variation at a fraction of the computational cost of comparing every solution with every other solution ($O(n)$compared to $O(n^2)$) [1]. This also allows the algorithm to iteratively adjust averages as more information becomes available on what constitutes a good decision value. However in multi-objective cases this can lead to average performance rather than finding the pareto-optimal front when objectives conflict. This strategy was proposed because the goal objective values in design problems are not typically known. These goals depend on problem complexity and are difficult to determine in advance. In addition, finding the extreme values of each objective can also be difficult which causes problems with objective scaling. For instance while it is relatively easy to determine the fastest and slowest system designs (a fully parallel design is the fastest, a fully serial design is the slowest), the largest and smallest system designs are more difficult to determine as these depend on the growth rates of multiplexer costs.

[1] O(function) is the symbol for the order of a function. This order refers to the term with the highest rate of increase in a function. This shorthand is used when describing computational complexity which is the time taken to perform a task or the memory requirements of an algorithm. $O(N^C)$ where C is a constant is a tractable problem of order polynomial. O(N!) however is an intractable problem of factorial increase.

In multi-objective cases, the number of times objectives were better than average was used as a fitness for proportional selection. Proportional selection introduces a random element while preserving the property that time spent analysing each allele, is proportional to the estimated probability that the allele is a desirable value. An extension was also developed that allowed different weightings for objectives. This was used to meet constraints for constrained systems.

The primary applications for this approach are systems where good values for each objective are unknown so there is no specific target to aim for. The algorithm must perform as well as possible in the time given. In outlining the proposed method a single objective case will be considered. This will then be extended to the multi-objective case. Further extensions allow objectives to be of differing importance.

The proposed selection operator bases selection on the probability of an allele being desirable in an individual. The population's genetic content is re-sampled in proportion to the number of times an allele is associated with a better than average objective. For example, if in 70% of cases with good performance gene 3 had allele A then this allele is re-sampled 70% of the time to generate the next population. The operator thus apportions processing efforts based on these values as probabilities that the allele is in fact beneficial.

Scaling is not necessary, nor is knowledge of the range of objective values. The proposed selection operator depends only on the difference between an objective and the average value for that objective. As the algorithm progresses these averages move toward desirable regions in the solution space. The expectation is that regions that allow multiple objectives to be better than average will be sampled more frequently. This means that computational effort will be mostly expended in these areas. Regions with fewer objectives that are good will receive fewer samples. Regions where all objectives are poor will be avoided.

5.2.2 Implementation

A fitness function is proposed for analysis that takes fitness as a function of the individual and average prevalent conditions of other individuals at the time. The basis for this approach is that by making fitness relative to other individuals, this enhances competition amongst individuals and so this should drive shifts in the average prevalent conditions[2]. This then

[2] This decision is also followed in regard to the hypothesis that an objective value better than the average is a result of desirable alleles.

feeds back into the Selection mechanism making selections progressively more restricted. When combined with a species system, this should mean that species evolve in parallel, causing other species to become extinct by outperforming them and taking their place in the limited population space.

For a single objective case, an individual is considered to be in the available pool for reproduction if and only if it is better than the average objective value for the current population. The next population is made from a random selection, with replacement from the pool available for reproduction. Using the average value as a cut-off, introduces the concept that the fitness of an individual is relative to all other individuals in the population. Because every individual selected for the next generation is better than average, the average values should improve. There is a small possibility of crossover and mutation resulting in average values that are worse but such a scenario is unlikely to persist. As the average value improves, the search becomes more restrictive as it is more difficult to excel the average. This has the effect of increasing selective pressure with successive generations. The rate at which selective pressure increases is governed by the evolutionary process and the problem presented. If better solutions are found more rapidly the average value increases more rapidly making the search more selective earlier. If however better solutions are difficult to find, changes in selective pressure may be slower or even a reduction in selective pressure may occur. The degree of selection should therefore evolve according to the problem presented.

The selection function chosen is called the Adaptive Goal Selection operator (AGS). The name is derived from the goal or target values for objectives being found adaptively during the search process. This has a specialised fitness function given in Equation (11) but otherwise uses Roulette-Wheel Selection. Equation (11) assumes maximisation problems although swapping the > and < operations in $\mu(n)$ would allow minimisation problems.

$$F(x) = \sum_{i=1}^{N} \mu\left(O_i^x - \overline{O_i}\right)$$

where **(11)**

$$\mu(n) = \begin{cases} 1 & if\ n > 0 \\ 0.5 & if\ n = 0 \\ 0 & if\ n < 0 \end{cases}$$

where $F(x)$ is the fitness of individual x, N is the number of objectives for the problem, O_i^x is the i[th] objective value of individual x and $\overline{O}_i$ is the average value of the i[th] objective values in the population. For example, if a problem had 5 objective functions (N = 5) to be maximised, and an individual had 3 objectives above average, 1 average, and 1 below average then $F(x)$ would be 3.5 (= 1 + 1+ 1+ 0.5 + 0). The fitness of this individual would be 3.5. It would be 3.5 times more likely to be selected than an individual with a $F(x) = 1$.

It is possible that during evolution individuals will evolve whose objective values equal the population average. Therefore it may be necessary to assign a non-zero value for $\mu(n)$ to use when n is equal to zero. This prevents a converged population from re-initialising. However re-initialising the population in such a case may be desirable. A converged population means the evolutionary process will find no more new solutions, re-initialising at this point provides the opportunity that a new search will find more solutions.

The proposed evaluation process is O(NQ) in computational complexity compared to O(NQ^2) [**16**] for pareto-dominance ranking of individuals, where N is the number of objectives and Q is the population size. The lower order of O(NQ) compared to O(NQ^2) means the proposed evaluation process is faster than pareto-dominance ranking.

5.2.3 Extensions

Here an extension to $F(x)$ is presented for the case when objectives differ in importance. Such is the case, for example, when a design for a handheld application is to be optimised for power, cost, and performance. Here, power may be assumed to be the most important objective as this determines the battery life of the product. Equation (12) presents the modified function.

$$F'(x) = \sum_{i=1}^{N} \mu_i'\left(O_i^x - \overline{O}_i\right)$$

$$\textit{where} \qquad \textbf{(12)}$$

$$\mu_i'(n) = \begin{cases} w_i^{high} & \textit{if } n > 0 \\ w_i^{med} & \textit{if } n = 0 \\ w_i^{low} & \textit{if } n < 0 \end{cases}$$

where $F'(x)$ is the new fitness, w_i^{high}, w_i^{med}, and w_i^{low} are weights supplied by the user. The higher the weight the more importance will be placed on making the objective value higher or lower.

While up to three weights may be required for each objective, it should be noted that these weights are auxiliary and are not, in general, required. In most cases $F(x)$ is sufficient. The Simple Genetic Algorithm and several other algorithms require weights to be given in order to assess multi-objective problems. Because weights are not compulsory this gives the proposed approach the advantage of requiring fewer parameters to be given by the user. Even when weights are used, the weights are simpler to interpret and require much less knowledge of the search space than traditional weights. Traditional weights must be chosen to allow for the range of objective values present. The weights for the proposed algorithm do not affect, nor are they affected, by the range of objective values present. They represent only the amount of importance or effort to be spent on a particular goal.

The priority values would not need to be an integer, but integers have the advantage that random number generators in computers generate discrete random numbers, thus priorities that are integers could be scaled to prevent the possibility of overflow or underflow. Overflow is where the sum of fitness values exceeds the maximum integer allowable on the computer. Underflow is where the fitness value is smaller than the minimum value that can be represented on the computer. When overflow occurs an exception may be generated. When underflow occurs, the result is treated as zero. The latter generally does not generate an exception but has pronounced effects on results. This means that some individuals, due to underflow, have a fitness of zero and cannot be selected.

When an individual is worse than the average the user may wish to still allow a small chance of reproducing. This is useful for escaping local optima by allowing steps away from the local optima similar to simulated annealing. In this case weights for poor objective values should be non-zero.

A variant where fitness values were shared amongst members of each species was tested. This divided the fitness function by the number of members in an individual's species and has the effect of implementing fitness sharing at a species level. This type of operation can reduce the effect of super-fit individuals dominating a population and causing premature convergence. Super-fit individuals when reproduced also reproduce the same species tag.

Thus super-fit individuals for the same solution are more likely to belong to the same species. Because their fitness values are divided by the size of their species, super-fit individuals have their fitness values divided by an increasing number. This means the chance of selecting a super-fit individual becomes independent of the number of copies of the individual in the population. This serves to constrain the effect of their higher fitness values.

5.2.4 Constraint Handling

The removal method is a simple extension for selection operators that allow constrained problems to be optimised using a genetic algorithm. This method removes any individuals that violate constraints from the population. This method also has amongst the lowest performance due to the large loss of information caused. Every individual removed no longer contributes information about its good or poor aspects. If an individual were very close to satisfying the given constraints then it is lost using this method and this may not be a good outcome.

The penalty method is the most common extension for selection operators in constrained problems. Under this strategy the fitness of individuals may be reduced if constraints are violated. This requires a user to determine a scheme for applying penalties. It involves issues such as; is a penalty to be applied dependent on whether a constraint is violated, or dependent on the amount of violation? If the amount of violation is used then how is this to be measured? Will penalties differ with each constraint, or will violation of any constraint have the same effect?

In early work, a tuning of the proposed selection operator's weights using the proposed extended fitness function, $F'(x)$, was found to be able to be used to meet constraints. Tuning required large amounts of experimental data on the effects of different values, which then have to be iteratively modified and retested. It was found that unless the tuning obtained could be applied to other similar problems this method was too time consuming for practical use.

In more recent work, from the literature, constraints are introduced as additional objectives. Rather than include a special operator for handling constraints, some researchers [**71**][**72**][**73**][**74**] have included them as additional objectives to a problem. A simple example is to add the distance to the nearest feasible region as an objective and then to specify that objective to be minimized. When this distance reaches zero, its minimum value,

then the constraints have been met. This method's main disadvantage is that it increases the number of objective functions. In problems with many objectives this can greatly increase the size of the solution space making exploration difficult or in some cases too time consuming to undertake [**75**]. The increased number of dimensions can also change the topology of the solution space.

5.3. Crossover

In system design problems typically there are multiple design solutions for each problem. It is favourable to find as many of these solutions as possible to present the designer with the most options possible. Later in development these may lead to different product lines for different target consumers but at this stage it is important to find these alternatives. These different solutions will each occupy separate niches from each other. The subdivision of a population according to different niches is a typical strategy that allows different sections of the population to pursue different niches. The proposed crossover operator is designed to identify, isolate, and refine these different design niches more efficiently than a traditional crossover operator. This is achieved by restricting which individuals may mate and how this is implemented.

5.3.1 Tagging Individuals

The task of crossover is to sift out the similarities in good individuals. The emphasis in the proposed crossover is shifted from combining good patterns in separate individuals to refining a single pattern present in many individuals. Selection ensures individuals in a population are fit. An individual that is fit has a good genetic pattern causing the fit phenotype. This good genetic pattern may be considered an ideal genetic pattern with a few random variations. To obtain the ideal pattern requires the removal of the random variations or "genetic noise". The more individuals with this noise disturbed ideal pattern the easier it is to remove the noise and recover the ideal pattern. This requires identification of individuals that share a common ideal pattern. By tagging an individual with a species tag common ancestors are identified. Individuals that share common ancestors are more likely to share common patterns. To refine the pattern, genes are shuffled in individuals of a species. Those alleles that belong to the ideal pattern occur the most often and therefore the ideal pattern is least likely to be disturbed by this process. Those alleles that belong to the noise occur least often and are most likely to be disturbed breaking any false associations with the ideal

pattern. These false associations can occur where a poor allele rides on the performance of other alleles to remain in the population. Breaking these associations makes the removal of these alleles easier.

This process is an extension of the K-parent uniform crossover from Chapter 4. This extension covers the selection of parents for crossover. A Species Tag gene is appended to each individual. The Species Tag gene has the same number of alleles as members in the population. Initially the Species Tag genes are set randomly. Each species is then treated as a set of parents using K-Parent Uniform Crossover.

In addition to the traditional DNA for an individual in a simple genetic algorithm, genes are added to implement the species tag. The following shows an example of an individual with labelled alleles.

$$F_1F_2F_3F_4F_5F_6F_7F_8M_1M_2M_3M_4C$$

where F_i represents traditional DNA, M_i are the genes for the adaptive mutation operator discussed later in section 5.4, and C is the crossover species tag. C has the same number of alleles as the population size being used.

Individuals are tagged with a species gene which is the mechanism for recording genetic histories [**70**]. Mutation on the species gene is the only method for changing the species gene allele but such an occurrence is procedurally rare. A mutation of the species gene is common shortly after the beginning of an evolutionary run and persists for only a short duration. For most of the evolutionary process mutations do not occur on the species gene. These mutations play a significant role in terms of solutions which are discussed later. Individuals that belong to the same species are most likely, however, to have inherited their species gene from a common ancestor. This means that the species gene acts as a record of inheritance or genetic history. Sharing a common ancestor also means that individuals of the same species are likely to share many genes in common with each other. This means that the species gene identifies a common gene sequence present in the individuals of the species.

The species tag is encoded on a single gene. During crossover, individuals with the same crossover species tag are all crossed uniformly within their species. This allows the number of parents used, and which parents to cross, to be determined via evolution. Thus if a population of individuals exists such that:

A A A A A A A A A A A A 1
B B B B B B B B B B B B 1
C C C C C C C C C C C C 2
D D D D D D D D D D D D 2
E E E E E E E E E E E E 1

where the numbers correspond to the crossover species tag and letters are other genes in the individual. From this, three individuals have crossover species tag 1, and two individuals have crossover species tag 2, then following crossover a possible outcome is:

A B A B E A B E E A E B 1
B A E A A E E B B E B A 1
C D C D C D D C D C D C 2
D C D C D C C D C D C D 2
E E B E B B A A A B A E 1

where the individuals with crossover species tag 1 have been uniformly crossed, and likewise the individuals with crossover species tag 2 have been uniformly crossed.

Tags in the literature are implemented on multiple genes, but here are encoded on a single gene [**70**]. The proposed method uses a single gene to encode the tag, thus the likelihood of mutation on the tag is greatly reduced. This is important when considering genetic histories, discussed in this chapter, which are important for this operator.

5.3.2 Mate Selection

The proposed crossover operator is known as a non-panmictic operator, meaning interactions between individuals are restricted. Non-panmictic population structures have more complex Markov Process models, as the reductions possible when selection probabilities are the same, cannot be applied [**51**]. This approach primarily addresses the selection of parents and the number of parents involved. The species tag gene has been added to track genetic histories and implement crossover using this tag. Tags have been proposed before in the literature for this purpose [**70**], however the proposed implementation has several important differences in implementation and in the intended result.

The proposed operator is a positive assortative mating operator. Assortative mating is where the likelihood of individuals mating is influenced by the similarity or dissimilarity of the

individuals involved. Positive assortative mating is where individuals are more likely to mate if they are similar. Negative assortative mating is where individuals are more likely to mate if they are dissimilar. The goal of this operator is to identify key genetic sequences that lead to desirable phenotypic characteristics. The species tag is a method of annotating individuals with similar genetic sequences. Should a species survive and multiply, then the species must have a desirable genetic sequence present. Random variations within the species prevent the taking of a gene sequence of any member of the species as a solution. Instead the proposed crossover operator filters out the random variations to obtain the key genetic sequence. This sequence is one of the solutions for the problem's solution set. Other solutions will typically be present in other species.

Every member of a species is involved in crossover when generating a set of children for a species. This maximises the chance of disrupting random variations but should have minimal effect on the key genetic sequence due to the prevalence of alleles from that sequence. The key sequence will be disrupted on some members of the species but others are likely to retain intact versions of the key sequence and only one is required to find that sequence. Those individuals with disrupted sequences will be iteratively removed by the selection process in later generations.

Using species as proposed, allows the evolution of a parent selection mechanism determining both how many parents there are and which individuals are parents. Higher numbers of parents are used in niche problems to refine specific patterns. In this case every species is effectively competing for different niches in a single environment. Deciding which individuals can mate together is important in this case for preserving the key genetic sequences. The proposed mutation operator is responsible for finding these sequences, but the proposed crossover operator is responsible for refining them.

Crossover is achieved with the addition of a gene to each individual. This "Crossover" gene or species tag identifies species within the population. Only individuals of the same species can be crossed with each other. During crossover, every individual in a species acts as a parent for crossover and produces a set of children equal in size to the set of parents. The children then replace the parents to form the next generation for the species. This is done for every species in the population. This allows the evolution of the number of parents in crossover while also accomplishing selection of parents for crossover.

Usually the same selection mechanism used in reproduction is used for selecting parents but this can have drawbacks. This is due to the fact that selection mechanisms usually use the phenotype for selecting individuals but crossover operators manipulate the genotype. The problem occurs where the mapping from phenotype to genotype is not unique. This means that individuals selected by a selection operator may be fit for different reasons. This leads to conflicting sets of alleles in their key genetic sequences. When crossed, such individuals lose fitness rather than gain fitness.

5.3.3 Ancestry

Tagging has been proposed in the literature [**70**] as a means of recording ancestry and isolating individual patterns present in sets of individuals. In non-panmictic populations there is a high degree of control over which individuals may mate. The same theory is applied here. However in the literature [**70**], implementations typically involve representations where the tag is spread over multiple genes and used in the selection of pairs of parents. Here the same theory is used to create a pattern filter using a single gene tag.

When the species tag is inherited it usually denotes a common ancestor. Individuals of the same species are also likely to have other genetic qualities in common. This is why they are viewed as a species. If a species is successful for any duration then its common genetic structures are more likely to be fitter than its competitors. While in nature many instances of mating occur and each mating is usually restricted to a pair, this procedure takes millions of years and has a very large population. As time frames and computational resources are limited the process is accelerated by mating all individuals in a species at the same time which is more akin to a pattern filter. What this process effectively does is sift the genetic sequences to find the common genes in a species, which is the intended result.

The ASGA algorithm refines a single genetic sequence in crossover as opposed to combining multiple good sequences in other crossover operators. There is no guarantee that combining two sequences will yield a better sequence. There is however a degree of certainty that a successful species has a common genetic sequence that is good, although in each member of the species it may be disturbed by “genetic noise”.

5.3.4 Species Management

The species tag is, in the majority of cases, inherited from previous generations, although during early generations high mutation rates can mean the species tag was produced by mutation. For the majority of an evolutionary run, species tags are most likely inherited from a common ancestor. During the first evolutionary steps species are establishing themselves and have little meaning in that most species have very few members that share no common elements. A species tag has greatest effect when the species have established themselves and some rudimentary genetic sequences have been identified.

When the species tag is mutated an individual may either branch into a subspecies, or jump species and join another existing species. If the new species tag for the individual is not in use already then a new species evolves or branches from an existing species. The new species is then free to advance in different directions to its original species. This allows optima near each other to more readily be found. If the new species tag is already in use by another species then the individual jumps species and causes a small disruption to the other species. This disruption can help avoid local optima and due to the relative rarity of such an event during the entire evolutionary process, will not become a dominate factor.

Figures 33 to 36 show the numbers of species, extinctions, subspecies evolving, and interspecies jumps obtained from 100 ASGA optimisation runs on a Sum-of-Absolute-Difference function for a window size of 8 by 8 pixels implemented in an FPGA. These evolutionary characteristics were recorded accessing the internal data structures of the ASGA during each run and are not normally an output of the ASGA. Figure 34 shows a peak in extinctions as the initial population is screened for poor individuals resulting from random initialization. The numbers of new species evolving and interspecies breeding do not exceed 25% of the number of species at their peak. The trends in these values are similar to the mutation rate trends which could be expected, as mutation is the sole cause for evolving new species and interspecies breeding in the ASGA. When the mutation rate is high the occurrence of these events is also high and reduces as the mutation rate also reduces. Interspecies breeding reduces at a faster rate because both the mutation rate and the number of species present are factors in determining the chance of a mutation in the species tag, yielding a value already present in the population. Both these values are reducing with successive generations yielding a high rate of decay in interspecies breeding.

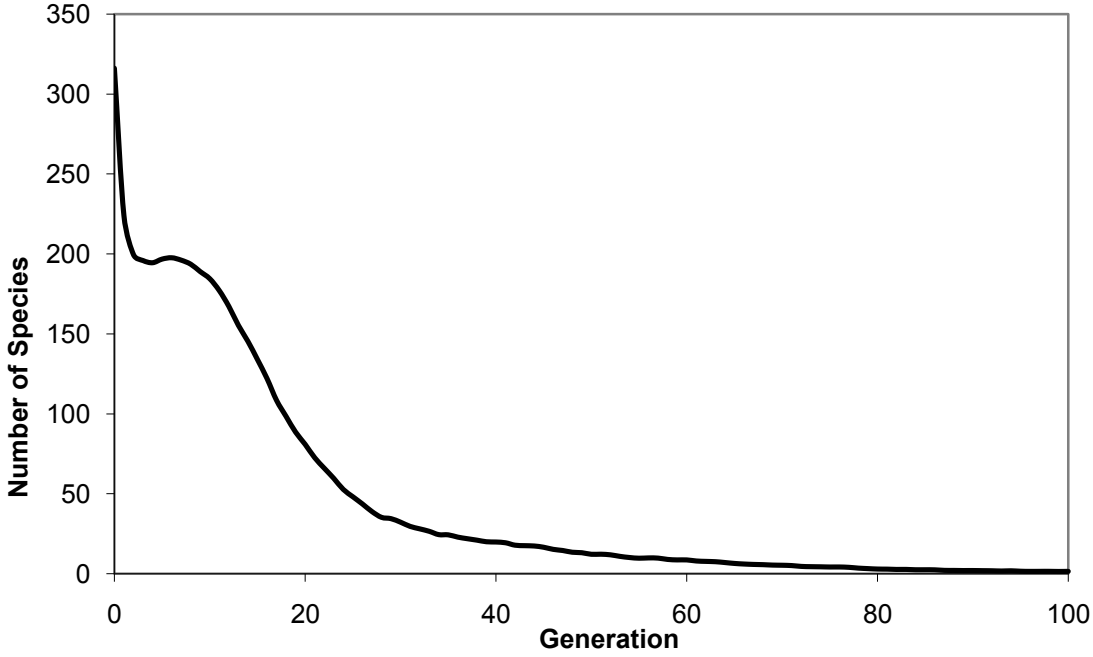

Figure 33: Number of Species

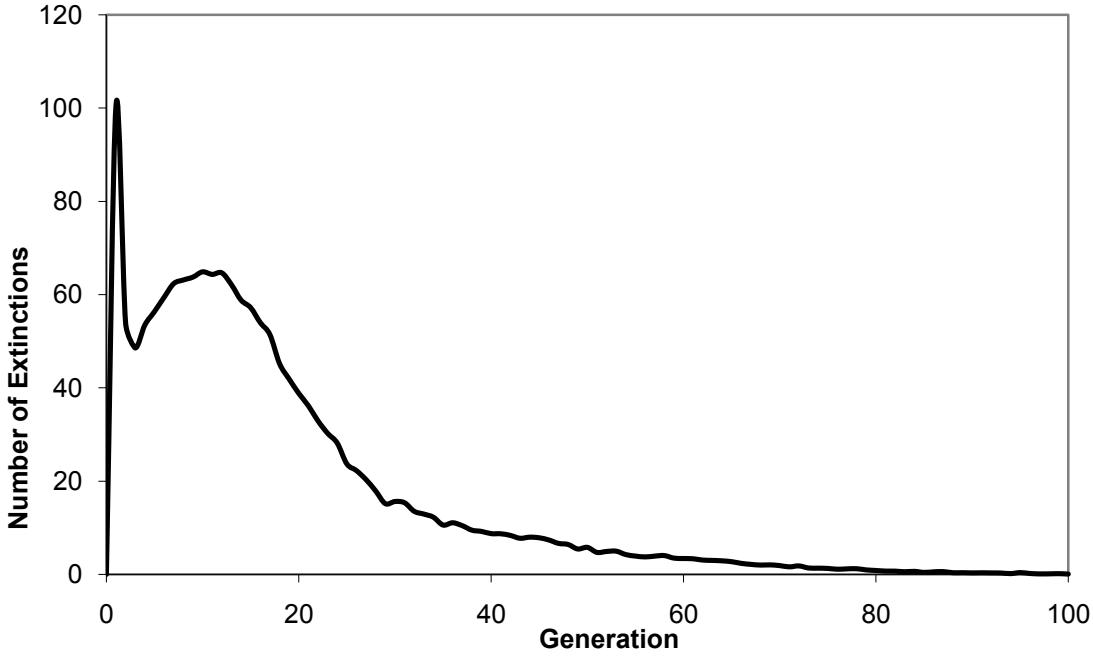

Figure 34: Extinctions

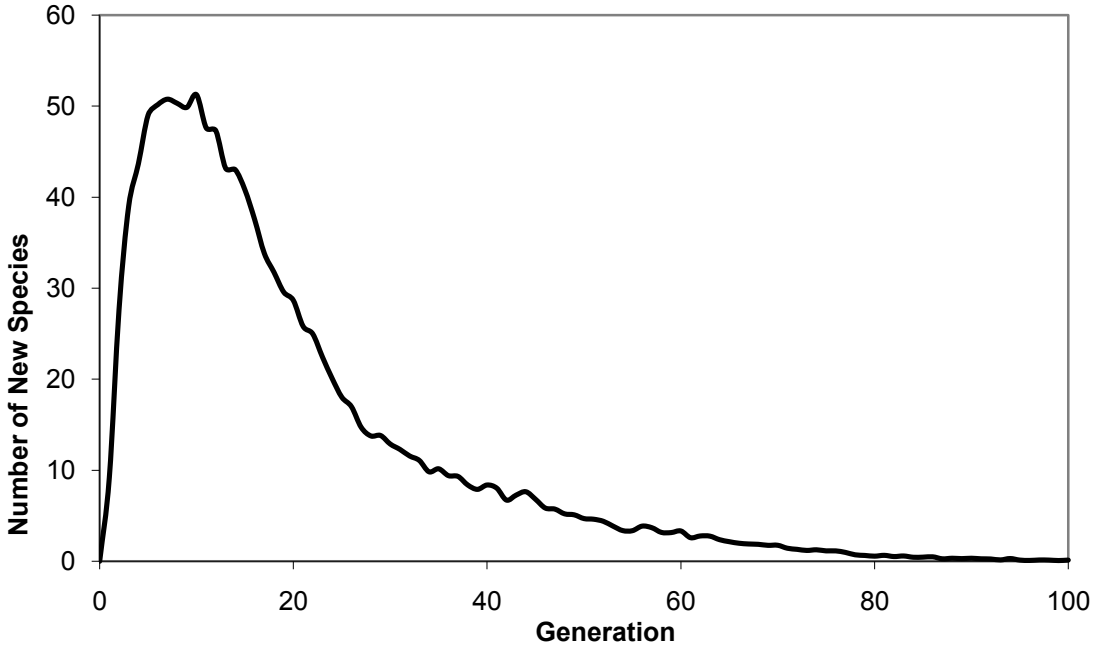

Figure 35: Evolution of Species

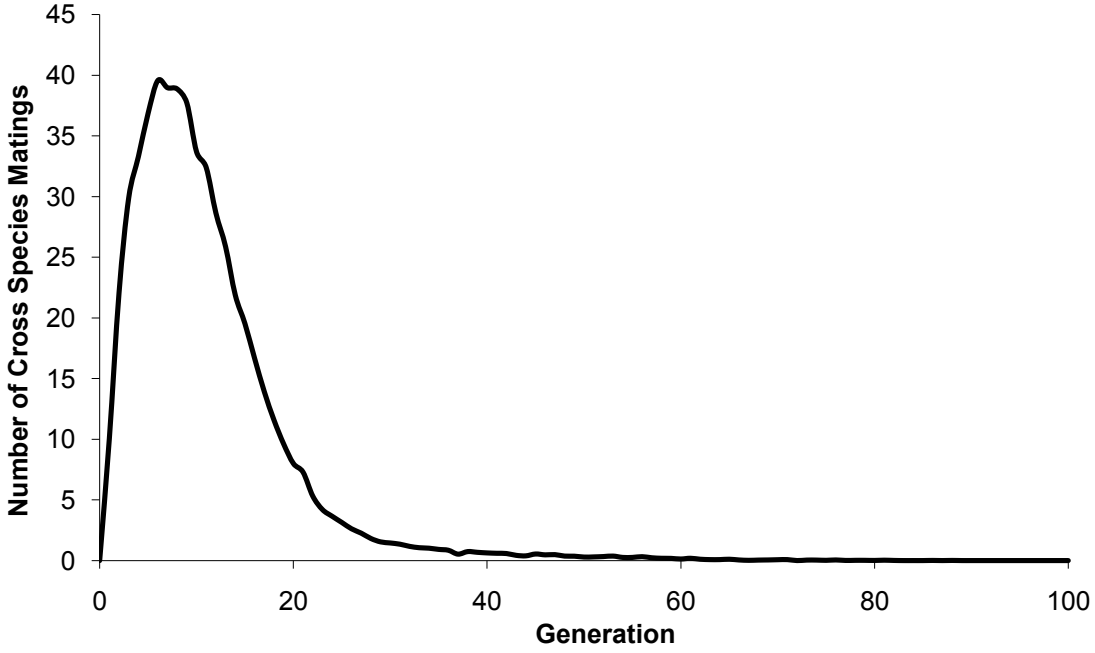

Figure 36: Interspecies Breeding

5.3.5 Implementation

All individuals in a species are uniformly crossed to accelerate evolution. This is done because the process is sifting the patterns present in the species to determine the key genetic sequence of the species. The key genetic sequence will have a disproportionately high number of its alleles present in genes throughout the species. An equal portion of genes from each individual are randomly reallocated to every other individual in the species. The genetic sequence most likely to survive this process is the sequence whose alleles are most common in the species. The sequence most likely to survive is thus the key genetic sequence of the species. Random variations in genes of lesser importance are subject to random reallocation by this process. The resulting phenotypic changes allow the variations in phenotype introduced by these genes to be determined. This process can be implemented by applying uniform crossover to each species with every member of a species acting as parent for a single crossover operation. Equation (13) presents this process, where S_i is a set where each row of the matrix is an individual, and each element along a row a gene within the individual from a total population P. The number of genes in an individual is n, "random_reorder" is a function that randomly shuffles the elements of a vector, and S_i' is the set after the proposed crossover operator has finished.

For

$$S_i = \begin{bmatrix} i_{1,1} & i_{1,2} & \cdots & i_{1,n-1} & i_{1,n} \\ i_{2,1} & i_{2,2} & \cdots & i_{2,n-1} & i_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ i_{m_i,1} & i_{m_i,2} & \cdots & i_{m_i,n-1} & i_{m_i,n} \end{bmatrix}$$

where

$$S_i \subseteq P$$

and

$$m_i = n(S_i)$$

and

$$\forall k \in [1, m_i], l \in [1, m_i] : i_{k,n} = i_{l,n} \wedge \neg \exists S_q, r \in [1, m_q], t \in [1, m_i] : q_{r,n} = i_{t,n}$$

then let

$$B = [1 \quad 2 \quad \cdots \quad n-1]$$

and

$$B' = random\_reorder(B) \wedge B' = [b_1 \quad b_2 \quad \cdots \quad b_{n-1}]$$

then

$$S_i' = \begin{bmatrix} i'_{1,1} & i'_{1,2} & \cdots & i'_{1,n-1} & i'_{1,n} \\ i'_{2,1} & i'_{2,2} & \cdots & i'_{2,n-1} & i'_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ i'_{m_i,1} & i'_{m_i,2} & \cdots & i'_{m_i,n-1} & i'_{m_i,n} \end{bmatrix} \quad \textbf{(13)}$$

where

$$\forall i, S_i \subseteq P, u \in [1, m_i], v \in [1, n-1] : i'_{u,b_v} = i_{(u+v-2) \bmod m_i + 1, b_v} \wedge i'_{u,n} = i_{u,n}$$

Table 8 shows a before and after example of a population P of size 6 with three sets S_1, S_2, and S_3. The last column contains the species tags. S_1 refers to individuals with a tag of 1. S_2 refers to individuals with a tag of 2. S_3 refers to individuals with a tag of 3. The population has six members numbered 1 to 6 (these labels should not to be confused with the species tag). In the before case every individual is completely homogenous which means every gene has the same allele. This is so that the source of each allele can be readily identified in the after case. In practice the occurrence of completely homogenous individuals is rare unless such an individual is on the pareto-optimal front.

Table 8 Crossover Example

	Individual	Genes							
Before	1	A	A	A	A	A	A	A	1
	2	B	B	B	B	B	B	B	1
	3	C	C	C	C	C	C	C	2
	4	D	D	D	D	D	D	D	1
	5	E	E	E	E	E	E	E	3
	6	F	F	F	F	F	F	F	3
After	1	A	B	D	D	B	A	A	1
	2	B	D	A	A	D	B	B	1
	3	C	C	C	C	C	C	C	2
	4	D	A	B	B	A	D	D	1
	5	E	E	F	E	F	F	E	3
	6	F	F	E	F	E	E	F	3

In Table 8 species 1 could be generated by having a B' vector of [1 2 3 6 5 4 7]. If the fourth and sixth columns are swapped then reading across individual 1 after crossover the genome would be ABDABDA. This shows the effect of the modulus addition used in determining where to obtain alleles from. U is 1 for individual 1, v increases from 1 to 7, and m_i is 3. The modulus addition of (u+v-2) mod m_i + 1 gives the sequence 0, 1, 2, 0, 1, 2, 0. A 0 means the allele is sourced from the first individual in species 1 which is individual 1 and has an allele of A. A 1 means the source is the second individual in species 1 which is individual 2 and has an allele of B. A 2 means the source is the third individual in species 1 which is individual 4 and has an allele of D. Altogether this gives individual 1 the sequence ABDDBAA1 following crossover.

5.4. Mutation

The ideal mutation rate to use is dependent on the degree of progress at a problem. This progress is problem dependent, niche dependent, and generally improves with subsequent

generations. The rate of this improvement would be difficult to infer given that the problems encountered are NP-Complete and this would require leveraging additional information from an NP-Complete problem. An adaptive strategy of encoding the mutation rate on the DNA allows the mutation rate to change with successive generations at a rate determined by the problem and the species of an individual because that species is linked through the individual's DNA. The fitness feedback from the selection operator controls the adaptive process to fine tune the mutation rate.

5.4.1 Adaptive Strategies

Mutation is used to try new combinations and restore lost alleles to a population. It is possible that an initial population may not contain a significant allele or that an allele is lost during evolution. Mutation is used to introduce the possibility of such alleles back into the population, and is employed to try new combinations or restore old combinations. The number of genes mutated should be linked to the fitness of the individual, the species it belongs to, and the extent of generations tried by the algorithm.

Goldberg proposed in [**15**] that adaptive mutation schemes could be implemented by encoding the mutation rate in the genome. Three different schemes of encoding the mutation rate in the genome were tested. These were using binary numbers, gray-code numbers and integers as the representation. In a binary or gray-code representation the mutation rate is represented by a set of genes each of which may hold an allele of '0' or '1'. For instance in binary representation to represent the number 8 at least four genes are required. Three of these genes would hold the value '0' and one gene would hold the value '1'. When interpreted they would be arranged in a string as "1000" where each character in the string represents the allele held by one of the genes. In gray-code representation at least four genes are required to represent the number 8 but in this case two genes would hold '0' and two genes would hold '1' as gray-code for 8 is "1100". In an integer representation only one gene is used but that gene may instead hold '0', '1', '2', '3', or higher depending on the total number of genes in the genome. To represent 8 in the integer representation the gene would hold the value '8'. Binary numbers encountered problems with the Hamming Cliff problem. The Hamming Cliff problem is where in order to reach the next binary number several genes must be changed. For instance "0111" (or 7) to "1000" (or 8). 7 and 8 are adjacent but "0111" and "1000" have a hamming distance of 4 which means they are distant from each other. Gray-code and integers performed equally well after convergence. Gray-code

numbers gave a faster convergence. Gray-code numbers achieve this because, unlike binary numbers, adjacent gray-code numbers differ by only one place. Continuing the example of 7 and 8 the respective codes are "0100" and "1100". The operator used here uses gray-code numbers to represent the number of genes to mutate in each individual. This number is allowed to mutate anywhere from no genes to all the genes in an individual. This operator is the main means of preserving diversity in a population in this work. This consists of a set of genes appended to the genome which represents a gray-coded number that when decoded gives the number of genes to mutate on that genome.

Encoding the mutation rate in an individual involves adding additional genes to the individual's genome that do not contribute to the phenotypic characteristics of that individual. Rather, these genes determine how subject to change the genome of the individual is. Introduction of this method relies on the evolutionary process to evolve mutation rates suitable for population exploration and refinement of individuals. Typically, high mutation rates lead to unstable phenotypes which mean that eventually, during a period of poor performance, the individual is removed from the population. This leads to a reduction in the overall mutation rate. Low mutation rates lead to stable phenotypes. However, if the phenotypes are all poor then the likelihood of a high mutation rate being removed during a period of poor performance is reduced. This is because all solutions are of poor quality. This allows individuals with higher mutation rates to evolve and remain in the population longer. This effect increases the overall mutation rate. This higher mutation rate moves the search to other regions where fitness values may be higher. When a higher set of fitness values is found stability becomes more important and mutation rates start to decline again.

By encoding the mutation rate on an individual the mutation rate is linked to the individual's fitness and species through its genome. The iterative improvement of fitness caused by selection is most easily maintained with a low mutation rate, though better individuals can be found faster with higher mutation rates over the first few generations. This factor ties the extent of generations to the mutation rate.

5.5. Encoding

The mutation rate is encoded as the number of loci to change during mutation. Three scenarios were tried in this regard: binary encoding on $\log_2 N$ genes, gray-code encoding on $\log_2 N$ genes, and integer encoding on a single gene. It was found that binary encoding gave

the worst results, both in terms of quality of the final result and performance over time. Integer and gray-coding both gave equal final results, but gray-coding had better performance over time. The mutation rate was thus encoded on multiple genes using gray code.

The mutation rate for an individual is permitted to range from 0 to 100%. This is to accommodate the range of values used during an evolutionary run. For a limited duration the highest mutation rate for any individual in a population can approach 100%. These individuals serve to create an influx of new alleles for other individuals. They are however only present during a small number of generations just following the beginning of the evolutionary run.

Two different methods were evaluated in initialising the genes used to determine the mutation rate. Random initialisation and seeding with a fixed value were tried. In random initialisation each gene of the mutation code is randomly set to either 1 or 0. In seeding, the mutation code of every individual is set to mutate a specified of genes. It was found that appropriate seeding of genes gave the best performance with the best seed being 5% of the genes but ensuring that at least 1 gene would be mutated in short genotypes.

5.6. Chapter Summary

In this chapter the three operators of the ASGA have been presented. The experiments conducted during their design are given in Chapter 6. Chapter 7 analyses their performance for a test bench of applications against other leading genetic algorithms.

The proposed selection operator uses the objective averages for the population to iteratively estimate the likelihood an allele is present in the pareto-optimal front. This is then used in allotting optimisation time to different alleles.

The proposed crossover operator utilises species to identify and refine key genetic sequences. This relies on the selection operator to ensure fitness and mutation to create more species. The crossover operator randomly rearranges equal proportions of each member of a species to every other member in the species. This disrupts undesirable patterns while preserving desirable patterns.

The proposed mutation operator is a specialised encoding of the mutation rate onto the genotype using gray-code numbers. Gray-code numbers have the benefit of a low hamming distance between adjacent numbers which facilitates progression through different levels of

mutation. Initial seeding with a 5% mutation rate is used based on experiments given in Chapter 6.

Chapter 6. Algorithm Evaluation

In this chapter the ASGA proposed in Chapter 4 and detailed in Chapter 5 is evaluated against an SGA using a number of experiments to determine how to most effectively implement each genetic operator. The analysis uses selected functions with different topologies to evaluate performance. The results are intended for qualitative analysis to determine which alternatives are better rather than by how much as the latter can be changed by scaling and shifting functions. This chapter details the selection of the remaining design choices in ASGA's operators. These include the how the implementation of the species tag was selected and the implementation of an adaptive mutation rate. ASGA is evaluated with these design decisions against other algorithms in Chapter 7.

6.1. Overview

Ten test functions were used to assess the impact of different implementations of the proposed operators in Chapter 5 against a standard Simple Genetic Algorithm (SGA) as a benchmark. Since an ASGA is based on the linking of genetic operators through data held in the population, the ASGA's operators were tested in all the different combinations possible. The three operators evaluated were Adaptive Goal Selection (AGS), Adaptive Speciation Crossover (ASC), and Adaptive Gray-code Mutation (AGM). AGS adaptively refines an estimate after each generation of the likelihood an allele is present on the pareto-optimal front. This is used to allot processing time to alleles. ASC employs speciation to identify key genetic sequences which it refines during crossover. AGM modifies the mutation rate for use at each stage of the evolutionary process.

Throughout this analysis, a standard SGA [**15**] was used to benchmark the proposed algorithm. Due to the large number of combinations of the ASGA's operators and the influence of search parameters on genetic algorithms, the SGA was selected as a single point of reference for this process. Despite this reduction there were still 549 different search algorithm configurations to be tried. This includes a range of search parameter settings for the SGA and every combination of the ASGA's operators to verify the basic functionality, concept of operation, and whether the ASGA's operators are functioning as expected.

In the following sections the various test functions are described, followed by the experimental method used in analysis. Lastly, the results of the analysis are evaluated and summarised.

6.2. Test Problems

The first nine test problems are from the literature and may be found in Goldberg [**15**] and Michalewicz [**66**]. These problems have been used to evaluate genetic algorithms on different types of problems in the literature. The significant aspects of each are their differing topologies and the level of epistasis involved. They represent a wide range of different optimisation problems which so that choices made reflect a wide range of different scenarios both in design problems and other optimisation problems. Epistasis is the degree of interdependence amongst genes. A high degree means that many genes do not show their function unless combined with several other genes. A low degree means that genes' function can be assessed by changing each gene in turn and assessing its effect These problems have been chosen to assess different elements of the proposed operators against a standard Simple Genetic Algorithm (SGA) [**15**]. The last problem is a multi-objective quadratic problem specially designed for this work to test the proposed selection operator on a problem where it is possible to simultaneously increase all objectives up to a set limit. Each of the test problems are described in the following subsections.

6.2.1 70 Ones Problem

The simplest type of test function is a binary string where the objective is to maximise the number of ones in the string. This test function is used for checking the "building block" functionality [**15**] of genetic algorithms. To solve this problem an algorithm must be able to combine several smaller sets of ones to make larger sets of ones. Each set of 1's may be considered a "building block" for an even larger "building block". This type of problem has a very low epistasis as each gene's effect can be determined gene by gene. This problem is only suitable for testing the ability of an algorithm to combine simple "building blocks".

For this test function a string length of 70 was used. Although this is a simple problem to optimise given that each string position is independent of any other string position, the problem becomes difficult when this information is not supplied. There are approximately 1.18×10^{21} points in the solution space. For good results an algorithm must determine a strategy that involves combining sets.

6.2.2 30 Ones Problem

The second test was another binary string where the objective was to maximise the number of 1s but this time the string length was reduced to 30. The objective of this test function was to determine how successful the ASGA was at finding the true optimum solution. The true optimum is 30 1's. This test differs from the first in that the algorithms are tested to find not just good solutions but the best solution possible. This type of test determines the ability of an algorithm to make fine adjustments in the later stages of optimisation to improve from good solutions to the best solution.

6.2.3 Allele Alphabet Size Problem

A test of the ASGA on genomes with higher numbers of alleles per gene was also conducted. This time instead of a binary allele alphabet, 5 chromosomes each with 14 genes were used. Table 9 gives the alleles each gene may take dependent on which chromosome the gene is on. The objective of this test function was to maximise the sum of all the alleles on each chromosome. This has a maximum of 210. The objective was to test the ASGA on genotypes that had more than two alleles. This was done because having more than two alleles impacts upon the "implicit parallelism" [**15**] present in genetic algorithms. "Implicit parallelism" is the process by which an individual in a population not only represents itself but also a set of patterns of important genes within itself. For instance the individual 111011 contains the pattern 111011, 1x1011, xx1011 and so forth where x is a "don't care" state that indicates the value of the gene at that position is not important. "Implicit parallelism" means that when an individual is evaluated then at the same time all its contained patterns are also evaluated. Increasing the number of alleles a gene may take detrimentally affects "implicit parallelism" by reducing its effectiveness, as outlined by Goldberg [**15**]. The design concept for the ASGA differs from a typical genetic algorithm and so this test is conducted to see if the difference is sufficient to avoid problems caused by allowing genes to have a larger alphabet of alleles.

Table 9 Genome

Chromosome	Alleles
1	0, 1
2	0, 1, 2
3	0, 1, 2, 3
4	0, 1, 2, 3, 4
5	0, 1, 2, 3, 4, 5

6.2.4 Iterated Prisoner's Dilemma Problem

The ASGA was also tested on the Iterated Prisoner's Dilemma problem [**15**]. In this well known problem two prisoners can either defect or co-operate with each other. At any stage they do not know what the other prisoner is going to do but they do know what the other prisoner has done in the past. If they co-operate with each other both score 3 points. If one defects the defector scores 5 points while the prisoner that co-operated scores 0 points. If they both defect then they only score 1 point each. The purpose of this problem is to determine a strategy that maximises a prisoner's score. For this exercise the algorithms were not told how many stages were to be involved. This means that co-operating until the last stage and then defecting when the other prisoner cannot retaliate is not possible as the algorithms are not informed which stage is the last stage. For evaluation purposes 100 stages were given. The maximum score possible is 500. However, if both strategies are good then each prisoner can be expected to score 300 with a total score between them of 600. A hand coded reference strategy was used that would co-operate provided it was not betrayed, but if it was betrayed then it would retaliate until it had matched its score with the other prisoner's score. Consequently the maximum possible score is only 300.

This strategy weakens the requirements of the Iterated Prisoner's Dilemma problem as now it is possible to score well with weak strategies that favour co-operation even in the potential case of betrayal. To avoid this, the individuals would have to compete with each other rather than use a fixed strategy. This would mean a competitive co-evolutionary algorithm would have to be used. This would require a different algorithm design compared to our other test functions and so a fixed strategy was used.

Co-evolutionary algorithms tend to have strengths in different types of problems to evolutionary algorithms [**76**]. Co-evolutionary algorithms are employed where individuals must either compete or cooperate with each other to solve a problem. For example, in game strategy designs, individuals may compete against each other to establish the best overall strategy. This means that fitness is far more relative to other individuals than typical evolutionary algorithms. The design work in this thesis is not relative. For example, 100 logic cells is a concrete cost. As the problems encountered have concrete objectives an evolutionary algorithm was selected. As evolutionary algorithms are being tested a fixed strategy was used in the Prisoner's Dilemma problem to convert the problem to an evolutionary problem.

This problem tests the ability of an algorithm to evolve a strategy or in this case a very simple program that determines what to do next based on previous experience. As a simple program is evolved changes in genes can have a far larger effect than might be expected. For example, by changing a gene from cooperate to defect the path of states encountered could be changed. This might mean other sections of the program become active as now a new path of states is possible. Problems were encoded as a set of '1's for cooperate, and '0's for defect. The first six genes defined the 'initial state' for the history of each player's first three moves. For instance "11,00,01" indicated that each player first cooperated ("11"), then both defected "00", then the first player cooperated and the second defected ("01"). An additional 64 (= 2^6 possibilities) genes supplied the response that player 1 was to give based on the last three moves using the first six genes as an initial state. This is the same encoding given in Goldberg [**15**].

6.2.5 De Jong's F1 Function

The ASGA was tested on De Jong's F1 function [**66**] given in Equation (14). The task is unconstrained and requires minimization of the following function.

$$\sum_{i=1}^{3} x_i^2 \quad where\ x_i \in [-5.12, 5.12] \tag{14}$$

The function has no local optima and a single global optima of 0 at (0, 0, 0). This function has a very low epistasis. This function consists of three chromosomes which could be optimised independently. Each chromosome decodes to one of the x_i variables. Each chromosome had 10 genes that could be either '1' or '0'. These genes collectively represented a gray-code number that was then scaled to the range [-5.12, 5.12]. This function has a higher degree of epistasis compared to the ones problem but is still overall relatively low. This function is used to evaluate the effect of increasing epistasis.

6.2.6 De Jong's F4 Function

The ASGA was also tested on De Jong's F4 function [**66**]. This function features disturbance by Gaussian noise and serves to test the resistance of ASGA to noise in an objective function. The De Jong's F4 function is given by Equation (15) which must be minimised.

$$\sum_{i=1}^{30} i \times x_i^4 + guass(0,1) \quad where\ x_i \in [-1.28, 1.28] \tag{15}$$

The global minimum of De Jong's F4 function is 0 at (0,0,0,…..,0). This function incorporates noise and is used here to test the resistance of ASGA to noise in an objective function. An optimiser should still be able to find the optimum value to a problem in the presence of noise. This can be done by calculating an objective function more than once to average out the effects of noise.

This problem was encoded onto 30 chromosomes, one for each x_i. Each chromosome had 8 genes. Each gene could be either '1' or '0'. Each chromosome was interpreted as a gray-code number that was then scaled to the range [-1.28, 1.28].

6.2.7 Branin RCOS Function

The Branin RCOS function [**66**] was used as a test function for testing a polynomial function such as that in the De Jong F1 function but with oscillations caused by a cosine term. This extends the previous De Jong F1 function by including a fixed oscillation causing ripples in the function topology. Unlike noise, these ripples are a fixed part of the function and an optimisation algorithm must allow for these. The Branin RCOS function is defined as to minimise the function in Equation (16).

$$\left(X_2 - \frac{5.1}{4\times\pi^2}\times X_1^2 + \frac{5}{\pi}\times X_1 - 6\right)^2 + 10\times\left(1-\frac{1}{8\times\pi}\right)\times\cos(X_1)+10 \qquad \textbf{(16)}$$
$$where\ X_1 \in [-5,10]\ and\ X_2 \in [0,15]$$

This function has three global minima at function value of 0.397887 for the points (π, 2.275), ($-\pi$, 12.275), and (9.42578, 2.475). Figure 37 shows a three dimensional plot of the Branin RCOS function.

This problem was encoded as one chromosome for X_1 and one chromosome for X_2. Each chromosome had 16 genes. These genes could be either '1' or '0'. Each chromosome was interpreted as a gray-code number and scaled to the appropriate range, [-5, 10] for X_1, and [0, 15] for X_2.

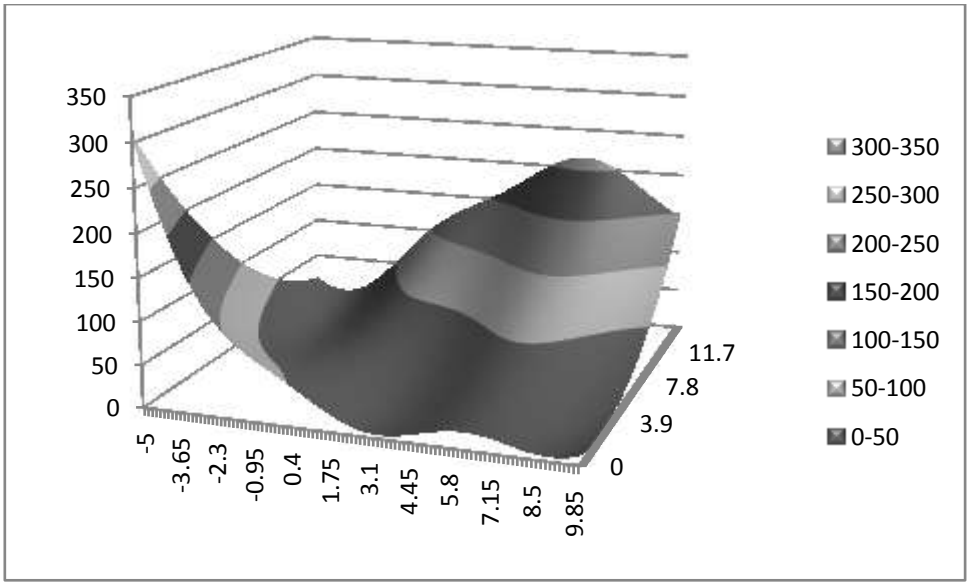

Figure 37: Branin RCOS function

6.2.8 Shubert

The Shubert function as detailed in [**66**] was used to test the ASGA because this function consists of a very high number of local minima with a select few global optima and very little information between optima to guide search algorithms. The Shubert function [**66**] requires the minimisation of Equation (17).

$$\left(\sum_{i=1}^{5} i \times \cos[(i+1)\times x_1 + i]\right) \times \left(\sum_{i=1}^{5} i \times \cos[(i+1)\times x_2 + i]\right) \quad \textbf{(17)}$$
$$where\ x_i \in [-10, 10]$$

This function has 760 minima of which only 18 are global minimum. The function's value at each of these global minima is –186.73. Figure 38 shows a three dimensional plot of the Shubert function. This function represents a very difficult case for optimisation due to the small amount of information available to guide the algorithms between peaks, and the large number of local minima present. This scenario is unlikely to occur and represents an extreme case. The ASGA's performance is evaluated on this extreme case.

X_1 and X_2 were implemented each on a separate chromosome. Each chromosome had 11 genes. Each gene could be either '1' or '0'. Each chromosome represented a gray-code number that was then scaled to the range [-10, 10].

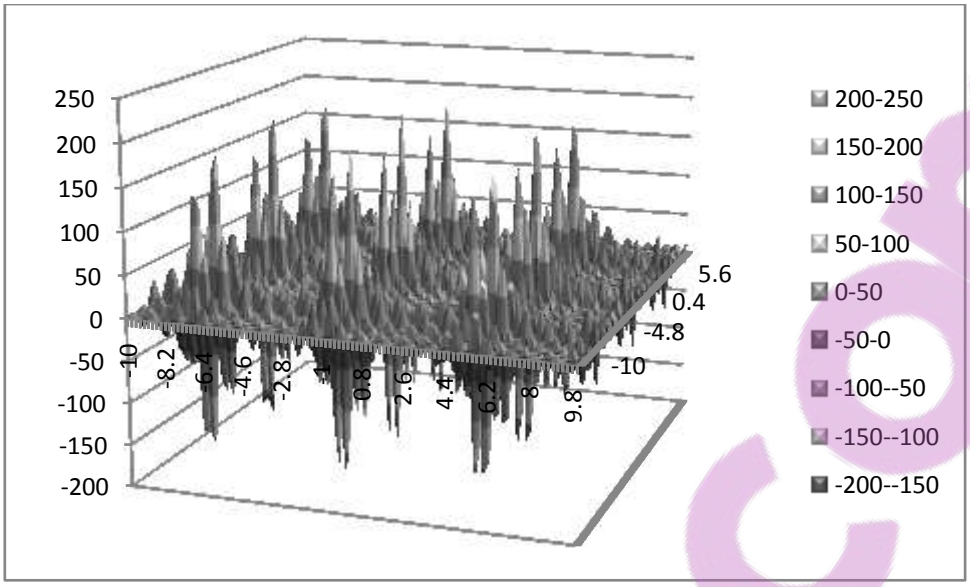

Figure 38: Shubert Function

6.2.9 Michalewicz Sine Function

A sine function was used to test the ASGA [**66**] to test the ability to overcome a series of troughs guided by a sequence of every increasing peaks. This problem attempts to maximise the expression in Equation (18) and tests the ASGA.

$$x \times \sin(10 \times \pi \times x) + 1 \quad \textit{where } x \in [-1, 2] \tag{18}$$

The maximum value is 2.85. Figure 39 shows a plot of Equation (18) over the selected range. This function is characterised by a set of increasing peaks in each direction. To reach the next peak a trough must be overcome. Finally, toward the end of the optimisation of the function it is necessary to discard one of the search directions as the other becomes a better option. The series of increasing peaks serves to guide the search and this function is not as difficult as the Shubert function to optimise. This function is used to check both the ability to overcome troughs and the ability to discard unsuitable search directions.

X was encoded using 9 genes. Each gene could be either a '1' or a '0'. These genes represented a gray-code number that was then scaled to the range [-1, 2].

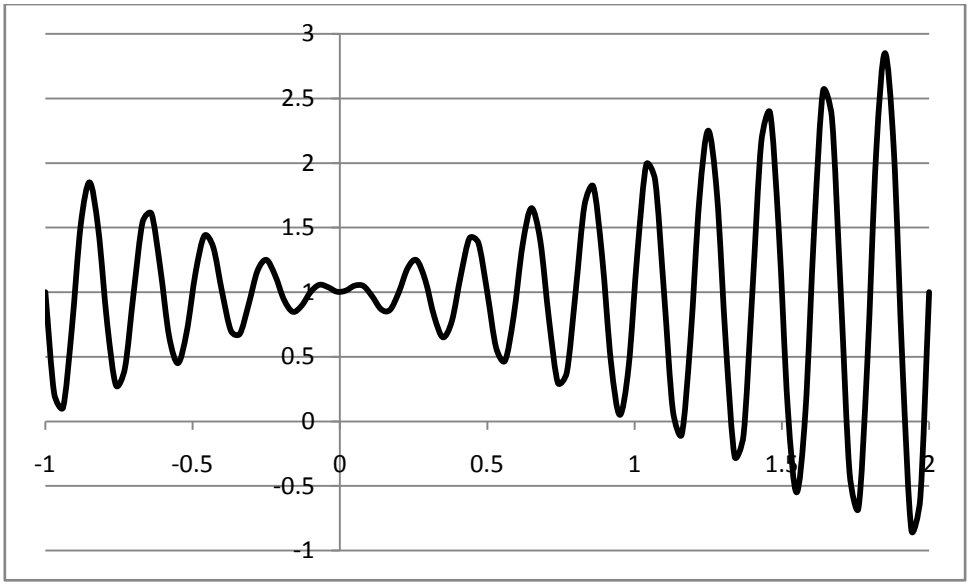

Figure 39: Michalewicz Sine Function

6.2.10 Multi-objective Quadratic Problem

Finally, a multi-objective problem to maximise four quadratic equations was also used for testing. The four functions are listed in Equation (19) as a set. Here it is possible to have higher values in most but not all of the quadratic equations.

$$\begin{aligned} &100 - x^2 \\ &121 - x^2 + 2 \times x - 1 \\ &108 - x^2 - x + 2 \\ &130 - x^2 + 3 \times x \\ &where\ x \in [-10, 10] \end{aligned} \quad \textbf{(19)}$$

An estimate that a good trade-off would be to maximise the sum of the four functions was made. This point occurs when x = 0.5, where the four functions would have the values 99.75, 120.75, 109.25 and 131.25 respectively. For comparison, the SGA considered was adapted for a multi-objective problem by considering the equations in random order.

X was encoded using 30 genes. Each gene could be either a '1' or a '0'. These genes represented a gray code number that was scaled to the range [-10, 10].

This problem tests the ability of an algorithm to incrementally improve several equations without decrementing any other equations. For example moving x from -2 to -1 improves every equation but moving from 1.5 to 1 improves the first three equations at the expense of the last equation. This problem is to test the selection operator that was designed for this type

of scenario. The selection operator will increase as many objectives as possible over a population.

6.3. Experimental Design

The goal of these experiments was to compare the ASGA against the best performance and reliability that could be obtained from an SGA. However in order to compare different configurations of the ASGA and the SGA, performance and reliability must be combined in a single metric. Performance is measured using the average values ($\overline{X}$) of objectives from trials on each problem from section 6.2. Reliability is measured by calculating the standard deviation (σ) of objective values from trials on each problem from section 6.2. The test statistic for determining which configuration of the ASGA and the SGA was best was $\overline{X} + 2.326 \times \sigma$ for minimisation problems and $\overline{X} - 2.326 \times \sigma$ for maximisation problems. This performance measure is based on a 99% single ended confidence interval with a normal distribution approximation and is used to estimate the likely average results an algorithm will generate.

The best performance for an SGA is obtained by the best selection of crossover and mutation rates for the population size used and the number of generations evaluated and a good scaling of objective functions. To find the best crossover and mutation rates requires an exhaustive search of each combination, which is a time consuming process and only attempted here because the algorithms are being tested rather than intending to solve the individual problems.

For the SGA, a good scaling of objective functions requires both knowledge of the objective values found in a starting population and the best objective values possible. The starting population values require a pilot population. The best objective values require a problem where the best values are known, which limits the effects of most scaling procedures. This additional information supplied to the SGA gives the SGA a considerable advantage over the ASGA. The ASGA is required to evolve suitable crossover and mutation rates as well as evolving its selection scheme.

To find the best crossover rate and mutation rate to use for the SGA each problem was tested with different combinations of crossover rates and mutation rates. The crossover rates were varied from 0 to 100% in 5% increments. The mutation rates were varied from 0 to 50% in 2.5% increments. This gives a total of 441 different combinations (21 crossover rates and 21

mutation rates). Fortunately the best values for each problem in Section 6.2 is known so a linear scaling was adopted to give the best objective values the highest fitness and objective values in an initial population were given a marginal fitness.

For the last test problem the SGA was extended to a multi-objective problem by considering objectives one at a time in random order as outlined in Goldberg [**15**]. In this procedure objectives are presented in random order one per generation. When each objective has been presented a new random order is determined and the process repeated. Uniform crossover of two parents was also chosen for the SGA. This was to avoid dependencies created by gene orderings as discussed in Chapter 5. A fixed mutation rate was used. It should be emphasised that the ASGA is being compared against the best obtainable from an SGA. Typically, however an ASGA will give far superior performance to an SGA. In addition, problems had to be specially prepared with scaling implemented in order to be able to use the SGA effectively on these problems. The ASGA requires no such modifications to these problems and can be applied in its most basic form to yield results comparable to the best that can be expected from an SGA. The ASGA must evolve its own scaling function for fitness values, and must also evolve its own crossover and mutation rates. The SGA is supplied with the best values to use for each of these.

To finalise the development of the ASGA it was necessary to determine if the ASGA operators would yield improvements as well as how to implement and initialise the species tag and mutation rates used. A multi-gene encoding and single gene encoding of the species tag were tested. The species tag was tested with random initialisation, assigning each individual a separate species in the initial population, and pairing individuals in species in the initial population. These options allow a random option which can avoid any pitfalls a deterministic algorithm might have. Initialisation of 1 member per species means that interactions between members of a population are delayed as each member belongs to a different species. This allows additional time before crossing of individuals occurs. Initialisation of pairs of members per species creates a crossover system that is very similar to standard uniform crossover involving two parents. Because each species contains two members initially, the chance of performing crossover with two parents is increased. This is still not guaranteed as crossover is performed after selection.

The mutation operator was tested with mutation rates represented using gray-code numbers, binary numbers, and a single gene containing an integer. The mutation operator was tested

with random initialisation, seeding with a 3% mutation rate, and seeding with a 50% mutation rate. To seed a mutation rate each mutation encoding gene is assigned a fixed value that when decoded yields the percentage of genes to mutate in an individual. Seeding with 3% means that 3% of genes will mutate. Seeding with 50% means that half of the genes will mutate. Random initialisation introduces a greater variety of mutation rates rapidly. Seeding allows a user to take advantage of the commonly used mutation rates that have the most support from empirical experiments, for instance a mutation rate of 3 to 5% is commonly used in most problems. These represent 108 different combinations to test (2 selection choices, 2 species tag encodings, 3 species tag initialisations, 3 mutation rate encodings, and 3 mutation rate initialisations). Combined with the 441 combinations for the SGA this yields a total of 549 combinations to test.

The test statistic was calculated using a sample size of 50 and plotted against the number of generations tried in order to evaluate performance over time. This allows a better comparison of the convergence characteristics of an algorithm rather than only considering the converged characteristics. This helps improve the analysis of each algorithm's performance by extending the information gained using transient as well as steady state characteristics of the algorithm and helps to highlight when convergence is achieved.

549 different plots either on separate graphs or a single graph would not be practical or interpretable. As such, due to this large number of configurations, only the best results are shown in the graphs of Section 6.4. The significant aspects of this choice are not so much the trends in the plots but rather which data sets are included as these indicate the best choices for setting up the ASGA. In the case of the SGA, these should be interpreted as the best obtainable from the SGA but the crossover and mutation rates are not necessary the best general values to use, since due to the large number of SGA trials (441) outliers can occur where seemingly poor choices have by chance yielded high performance. Comment is given where outliers are suspected of occurring.

The ASGA is adaptive and capable of determining its own search parameters; consequently it is not necessary to provide in advance, optimal search parameters for the ASGA for any of the chosen tests. The only requirement is that an identical population size to the SGA is used and furthermore when seeding our mutation rate, a value of 3% is selected. This is common in the literature for genetic algorithm applications.

A moderately sized population of 128 was chosen to allow sufficient processing power for the algorithm to function well but at the same time limiting population size so that the time taken to conduct these experiments was practical.

It should also be noted that because the SGA was effectively run more often, it is more likely for high performing outliers to be encountered in its results. While this means that it is more difficult to show the ASGA's improvements an advantage is that if an improvement is shown in the ASGA's results then it is unlikely to have been a statistical anomaly. The SGA was only run more often as there are more combinations of crossover and mutation rates to test in selecting these search parameters than there are of different implementations of the ASGA.

6.4. Results

At the outset it was unclear as to whether the ASGA selection operator would maintain performance or fail to adapt to new situations in determining which alleles were desirable. The tests conducted therefore included a set of tests where the ASGA's selection operator was replaced by the same selection operator used in the SGA.

The ASGA crossover operator featured a species tag but whether to use a multi-gene species tag or single gene species tag and how tags should be initialised was not apparent at the outset. A multi-gene species tag means that breeding between some species is more likely than for others although inter-species breeding is always rare. This would reflect that in nature it is possible for limited combinations of different species to successfully breed. A single gene species tag makes breeding between different species equally unlikely for every species. Random initialisation creates a different spread of species each time which can avoid any oversights in a deterministic method of initialisation. However, assigning each individual a unique species, allows additional time for selection to determine if a species is fit. Assigning two members to a species most resembles the well tested method of selecting two parents during crossover.

The ASGA mutation operator encodes the mutation rate onto an individual. How a mutation rate is represented and how initial values influence results was not immediately apparent. Gray-code encoding appeared likely to have advantages over binary encoding of genes for a mutation rate, but the place of single gene encodings was unclear. Random initialisation, again with the ability to avoid problems with determinism, could have too large a variation in

initial rates. Seeding mutation rates with fixed values, while deterministic, could allow commonly used values to be taken advantage of during initialisation.

Figures 40 to 52 show the results of the best performing configurations of both the ASGA and the SGA. To counteract the weaknesses of the SGA the test functions were individually scaled and shifted to give the SGA a good range of fitness function values for optimisation. Since these scalings are problem dependent they are reflected in the scales and ranges of the graphs to illustrate how the functions were changed. The value shown for the y axis (Performance) in each case represents the average scaled test function value obtained minus 2.326 times the scaled sample deviation. A higher value for performance represents better results. Due to the scaling giving the SGA an advantage, any results of ASGA that are better than the SGA indicate not only was ASGA better than the SGA but that ASGA was better than a well tuned SGA. Tuning the SGA to such a degree was computationally intensive and would not be practical for solving actual problems. The tuning process is only used here so that there is little possibility that an SGA with a different tuning would perform better. ASGA is adaptive and was left to adapt its own search parameters. The scaling function used changes between test functions dependent on what scaling gives better performance for the SGA, therefore the graphs show only relative performance between the SGA and the ASGA. In the larger Ones Problem, the Allele Alphabet Size test, and De Jong's F4 function the ASGA has better performance. In the Iterated Prisoner's Dilemma problem, De Jong's F1 function, the Branin RCOS function, and the Multi-objective Quadratic problem there are no significant differences between the algorithms. In the smaller Ones Problem, the Shubert function, and the Sine function the ASGA performed poorly. However as noted earlier, all problems were adapted for the SGA which requires prior knowledge of the problem and its solutions to perform well.

The Ones problem in Figure 40, without prior knowledge that the problem is highly decomposable, is considerably more difficult than the Ones problem of shorter length shown in Figure 41. A well scaled SGA has the advantage of being able to identify near optimal solutions to a short Ones problem more readily than the ASGA, due to the extra information supplied in the magnitude of the objective function. This means that the optimum value is identified as soon as it is found. This is a significant advantage that the ASGA does not have, as the ASGA does not assume objective functions have distributions suitable for direct use as a selection probability. ASGA continues its search building evidence to support the claim that it has found a good solution. The better performance of the ASGA in the larger problem

shows that it is able to take advantage of low epistasis when encountered more readily in larger problems than the SGA. In such large problems the optimum is unlikely to be found by chance thus reducing the SGA's advantage for large problems. Here the crossover and mutation operators have been able to adapt more effectively to the low epistasis in their rearrangements of alleles as opposed to the SGA's fixed strategy. This indicates that the crossover operator has selected its parent sets more effectively than the SGA. A better selection here means grouping together larger quantities of genes with an allele of '1'.

Many engineering systems in production consist of a large number of independent parts which from a genetic algorithms perspective means low epistasis. Good performance in the larger Ones problem can indicate an aptitude for such cases. The ability of using gray-code encoding for the mutation operator to give a rapid increase to the final value obtained and to ensure that the value is competitive with other alternatives is clearest in Figure 40. The best three plots all use gray-code encoding. An integer coding although initially exceeding the gray-code performance fails to maintain this and progress reduces eventually falling behind the gray-code encoding, taking a significant number of generations to reach the same performance. This transient characteristic is observed in the other results as well.

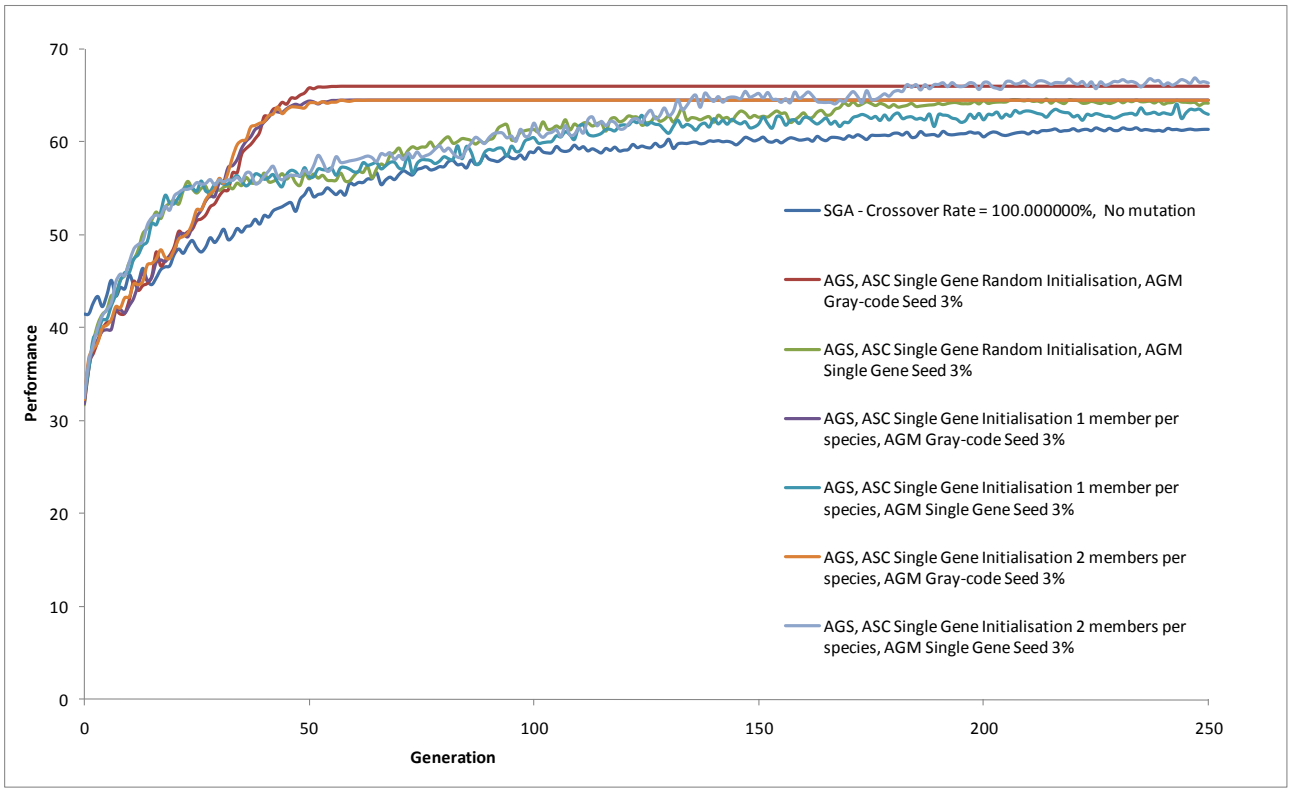

Figure 40: Large Ones Problem

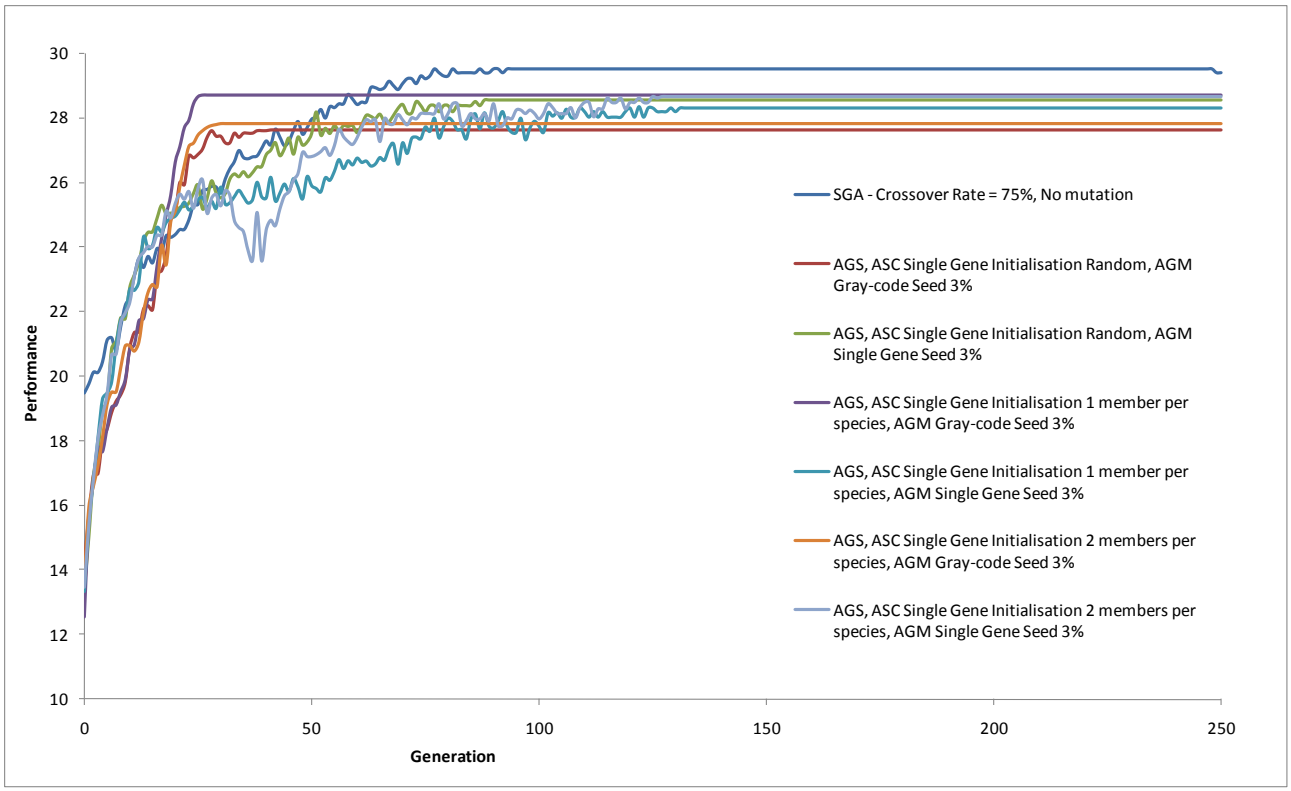

Figure 41: Small Ones Problem

The Allele Alphabet Size problem is used to test each algorithm's capability to deal with a highly separable problem but with multiple mutually exclusive alternatives. The analogy is where a system is being designed with separable parts, but in some sections of the design there are more than two suitable parts to perform that function. Compared with the SGA, the ASGA handled the situation of larger numbers of alleles better (as shown in Figure 42). This indicated that basing the design of operators on extensions of statistical tests and assumptions, combined with an unusual pattern filter and adaptive mutation has yielded an algorithm that processes genes with larger allele alphabets more effectively. When a sample is conducted on a set of values, then provided the sample is sufficiently large, inferences are more likely to hold true on a larger scale for producing better solutions. This relies on a low epistasis however. The SGA relies on implicit parallelism which is detrimentally affected by larger gene alphabets. The ASGA is able to circumvent this problem by more efficiently rearranging genes in order to process different combinations of alleles. This is because ASGA is able to adapt its groupings of species and mutation rates to the problem. The SGA however uses a fixed strategy and does not make such adaptations.

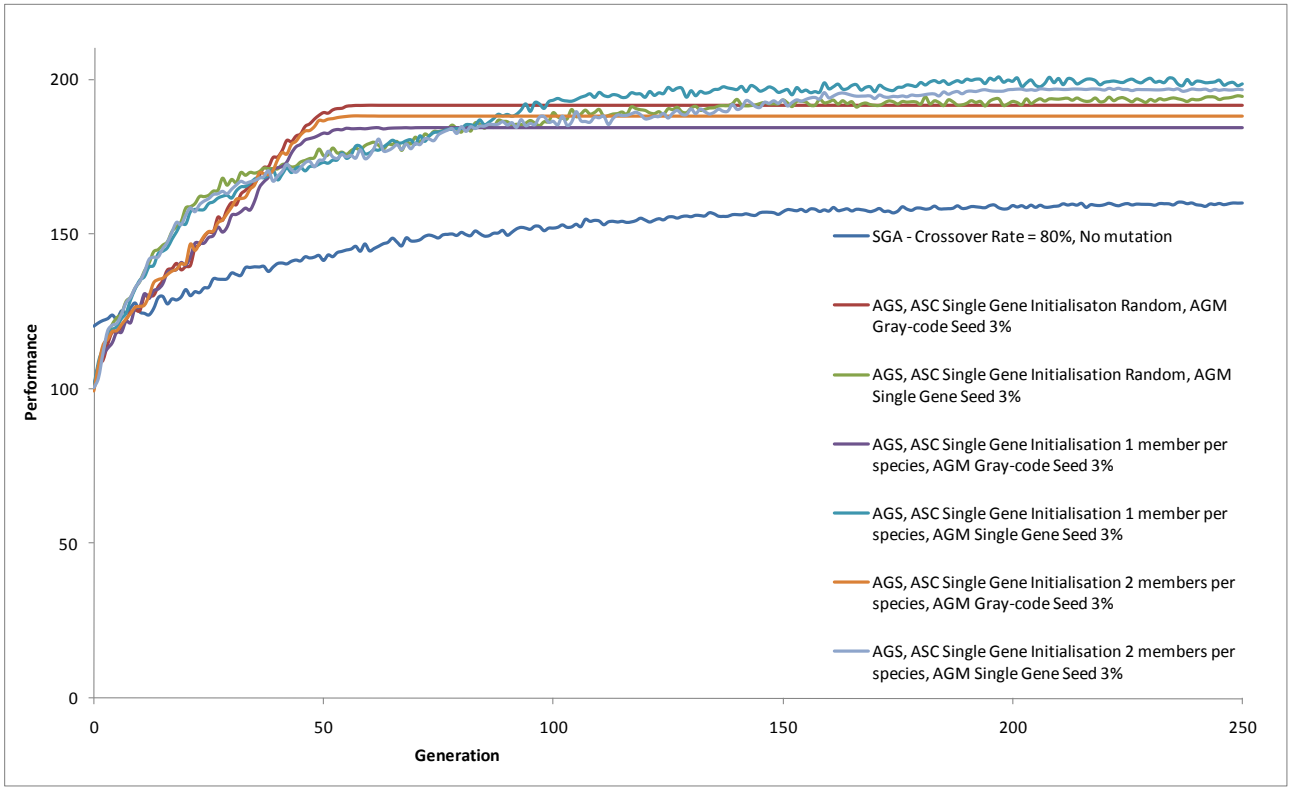

Figure 42: Allele Alphabet Size Problem

The Iterated Prisoner's Dilemma results given in Figure 43 indicate no significant differences in results once the ASGA has converged. This convergence occurs rapidly, within about 40 generations. In these tests due to the suitability of the objective function for use as a fitness function the SGA starts with the best possible solution. Due to this factor this test is more valuable for showing the ASGA's rapid convergence on the best solution rather than as a fair comparison against the SGA. The SGA in this case indicates that the crossover rate should be 100% and the mutation rate 50%. Since a mutation rate of 50% effectively randomly recreates a population each generation the SGA's result has only occurred because seven genes in the individual genome were "C". These genes were the initialisation genes for the Prisoner's strategies and the choice of what to do if the other prisoner kept cooperating. The encoding procedure of the Prisoner's Dilemma problem used may be found in Goldberg [**15**], while this outcome is a result of the fixed strategy used in comparisons.

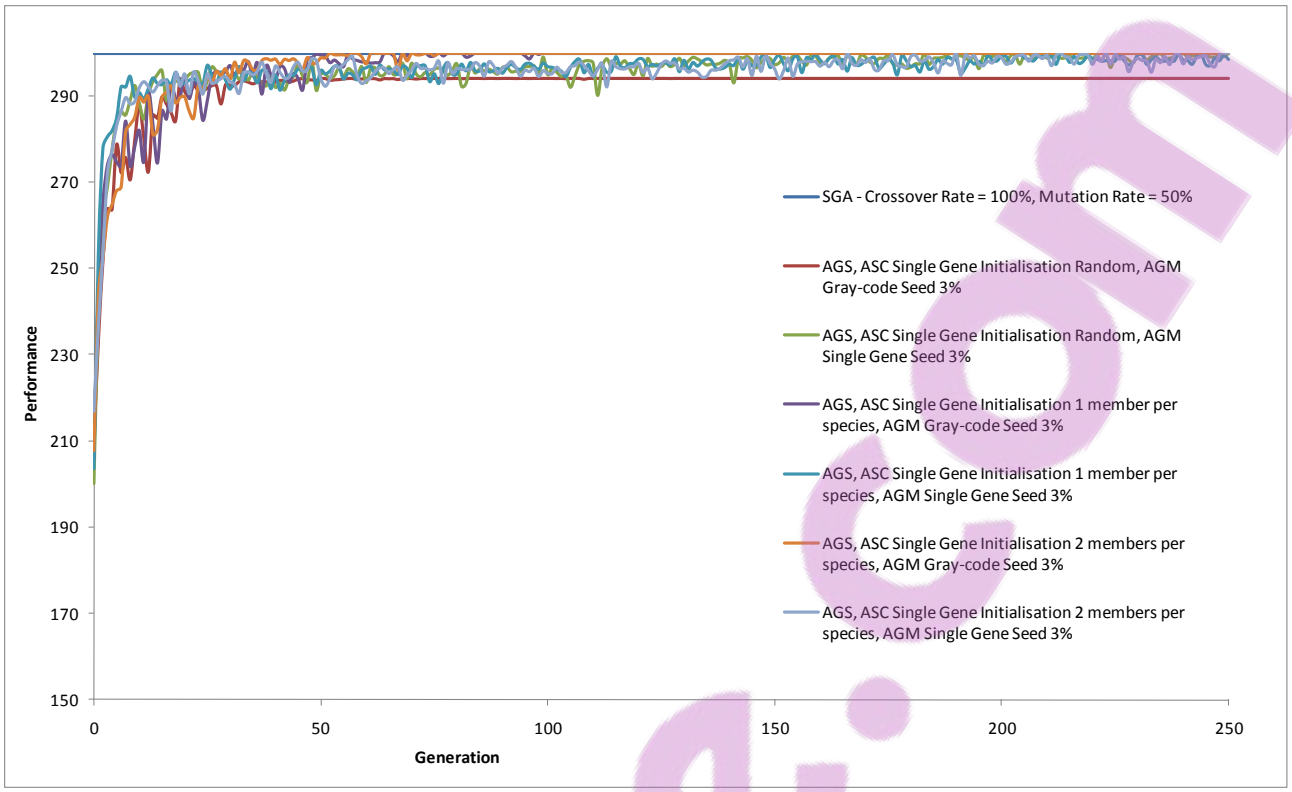

Figure 43: Iterated Prisoners' Dilemma Problem

The De Jong F1 function results in Figure 44 show no significant differences between the best results obtainable from the SGA and ASGA after allowing sufficient time for ASGA to converge. Again, due to function suitability for use as a fitness function the SGA has effectively started with the best solution. The performance plot is based on scaled versions of the De Jong F1 function used in the SGA. The ASGA solutions have been converted to the same scale for plotting purposes. In this case the relative values are significant rather than their actual magnitudes, as different scaling functions will produce different values but since linear scaling is used the plot shapes will remain the same. The same convergence characteristics may be observed for gray-code and integer encodings of the mutation rate.

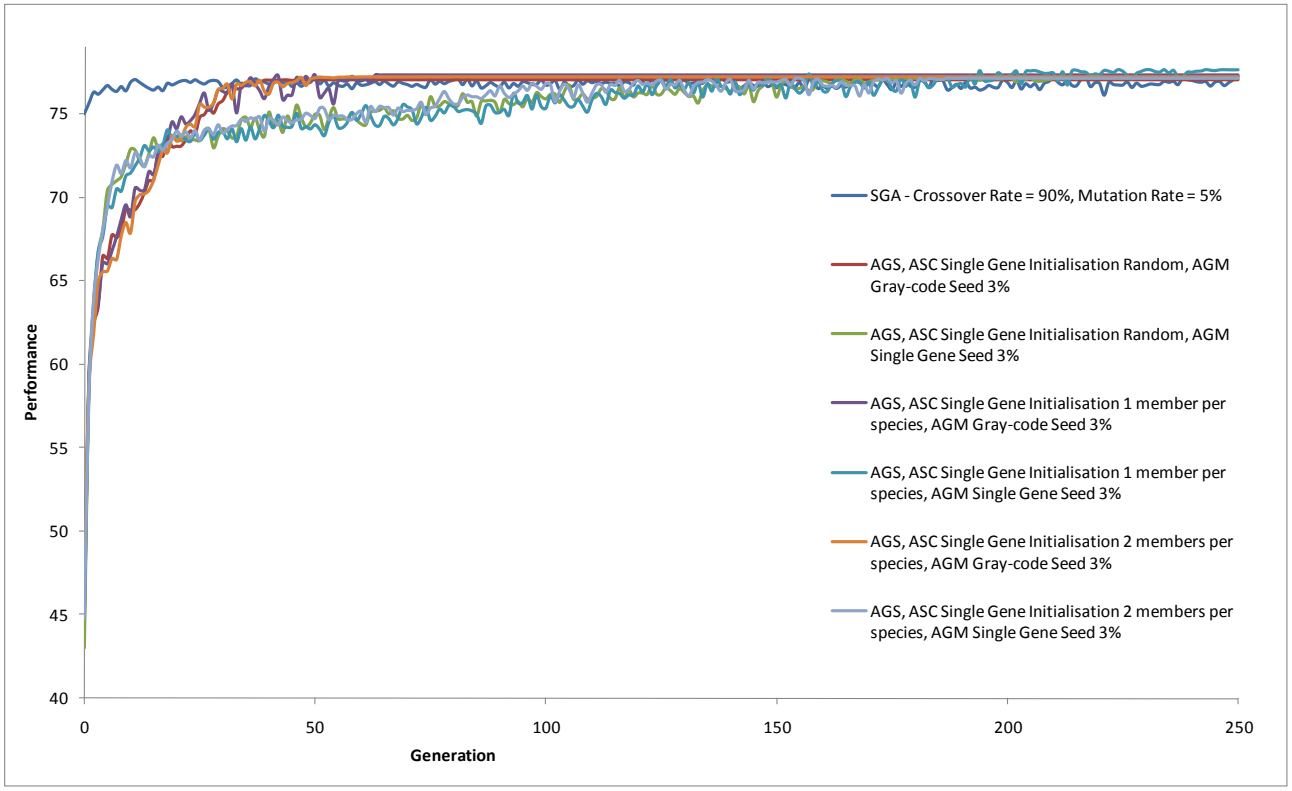

Figure 44: De Jong F1 Function

The De Jong F4 Function features disturbance by Gaussian noise. The ASGA algorithm has indicated an improvement over the SGA in this function as shown in Figure 45 which gives the scaled fitness values as a performance measure. This could be an indication of the affect of the ASGA's selection mechanism using the population averages and a warm-up period before making searches more restrictive. The population average is controlled by many individuals and the average of Gaussian noise is zero so its effect is minimised. In these results the best performance was achieved with a binary encoding for the mutation rate. In this case these results are likely to be the result of higher mutation rates being maintained due to the large hamming cliffs dispersed throughout binary encodings. This means a large number of mutations at each hamming cliff are necessary in the mutation rate encoding genes in order to decrease the mutation rate to the next level. This slows progression through the different levels of mutation. In this case the mutation rate has remained higher for longer and thus found better solutions before converging. The binary encoding does not appear in the best options for any of the other problems and so while giving good results for this problem it is not a suitable candidate in general.

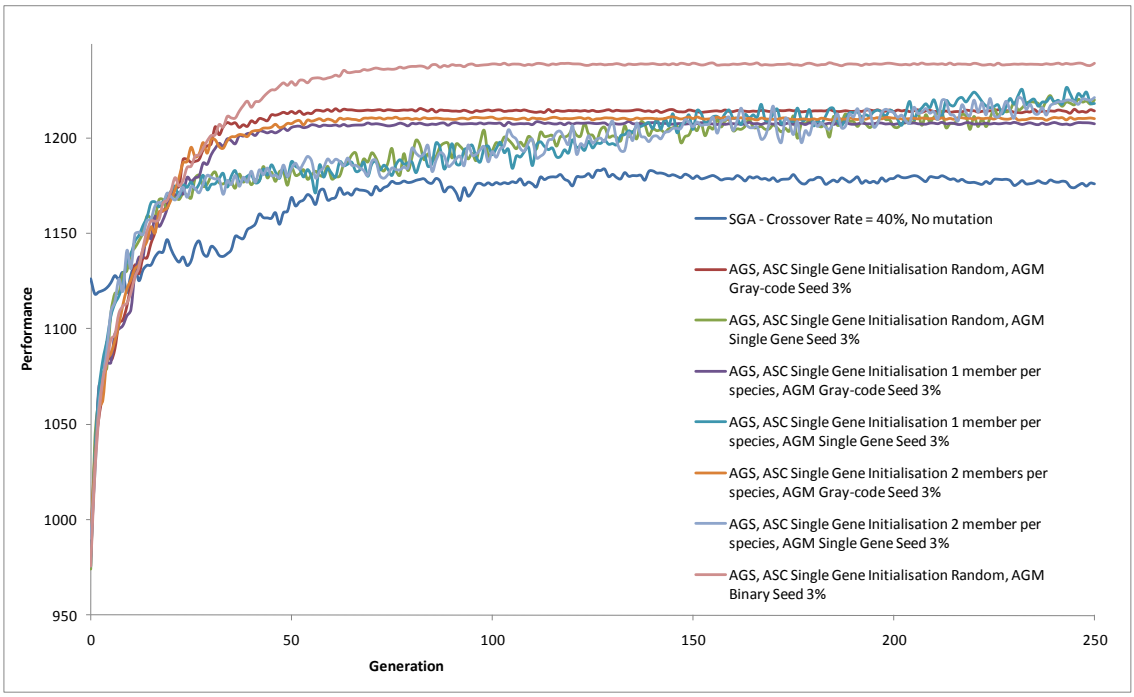

Figure 45: De Jong F4 Function

The results in Figure 46 for the Branin RCOS function show little difference in the converged states for ASGA and the SGA. The performance measure is base on the scaled fitness values. The same transient patterns may be observed for integer and gray-code encodings of the mutation rate. In this problem the SGA has essentially started with the best solution found.

Figure 46: Branin RCOS Function

The results in Figure 47 for the Shubert function show that the ASGA sometimes performs poorly with gray-code mutation operators however this is indicative that a larger population size should have been used as in these cases less than 50 generations were evaluated. The results of the better ASGA operators for this problem and population size setting were relatively close to the SGA's results given the lack of prior knowledge. The SGA results however are the best that could be expected from the SGA which was given substantially more information on the Shubert function than was supplied to the ASGA. ASGA's adaptive control of its search parameters requires the user to specify sufficiently the available resources for use via the population size. ASGA should not be typically used with population sizes similar in size to an SGA if large numbers of local optima are expected. ASGA requires a higher population size in these cases at which point it will evolve for longer rather than converging prior to completion. ASGA will then attempt a comparable number of solutions to the SGA.

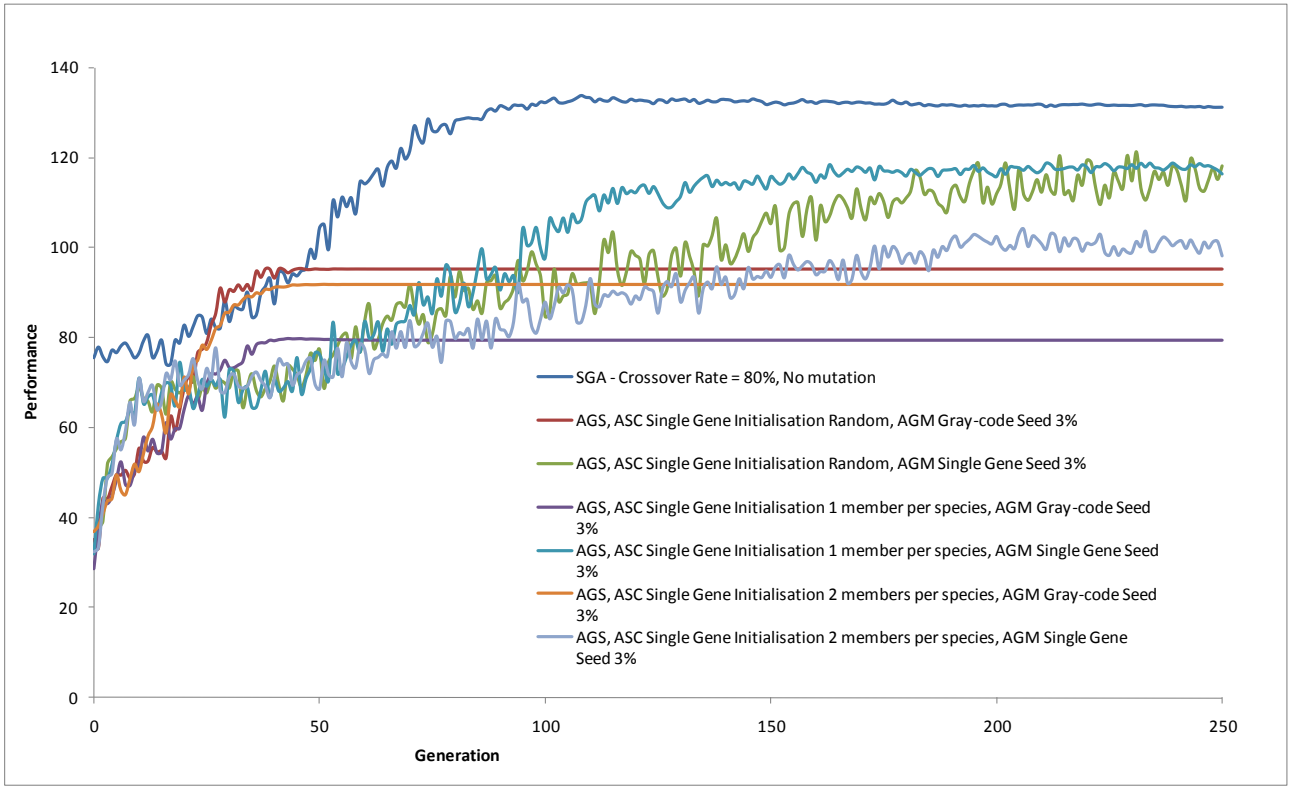

Figure 47: Shubert Function

The results in Figure 48 based on the scaled fitness values for Sine function show the ASGA gave poorer results but the margin of difference was not great and independent of the ASGA's settings. The configuration of a 0% crossover rate and a 5% mutation rate giving the best results for a SGA is indicative that the result for the SGA is possibly an outlier caused by the large set of 441 different configurations for the SGA. Since the best set is

taken from the 441 results the larger the number of results the more likely an outlier is to occur. In these results the best ASGA results were gained using a multi-gene species tag although the margin over other ASGA results is very small. Since a multi-gene species tag appears in the best results of only 2 of 10 problems as opposed to 8 of 10 problems featuring a single gene species tag the species tag was implemented using a single gene. In this example the SGA's use of the magnitude of the objective has given an advantage in finding the best value. The ASGA however is intended for problems where the magnitude of the objective has statistical properties that are not as easily optimised. Here this generality has resulted in the ASGA having a lower performance.

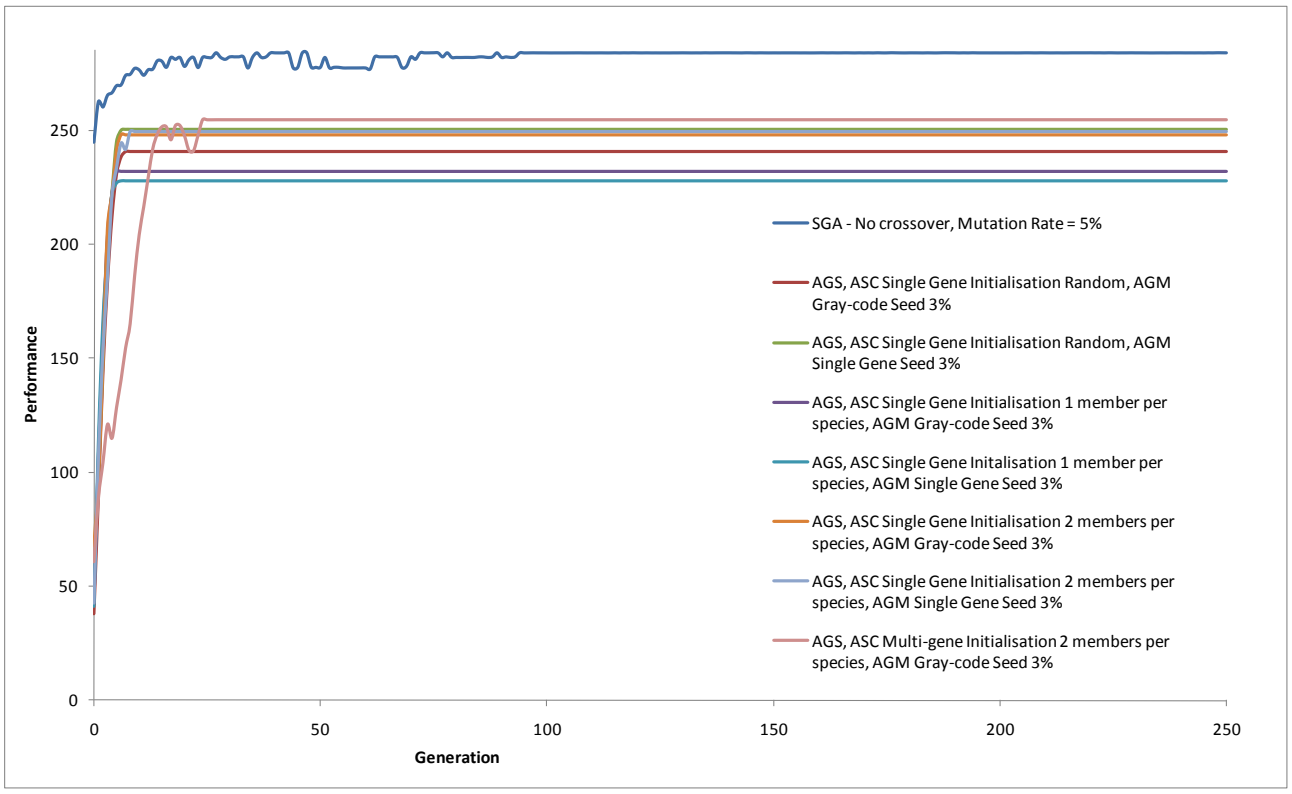

Figure 48: Michalewicz Sine Function

In the results in Figures 49 to 52 for the Multi-objective Quadratic problem there was negligible performance difference between ASGA and SGA. This was also the second case where a multi-gene encoding of the species tag appeared in the best results.

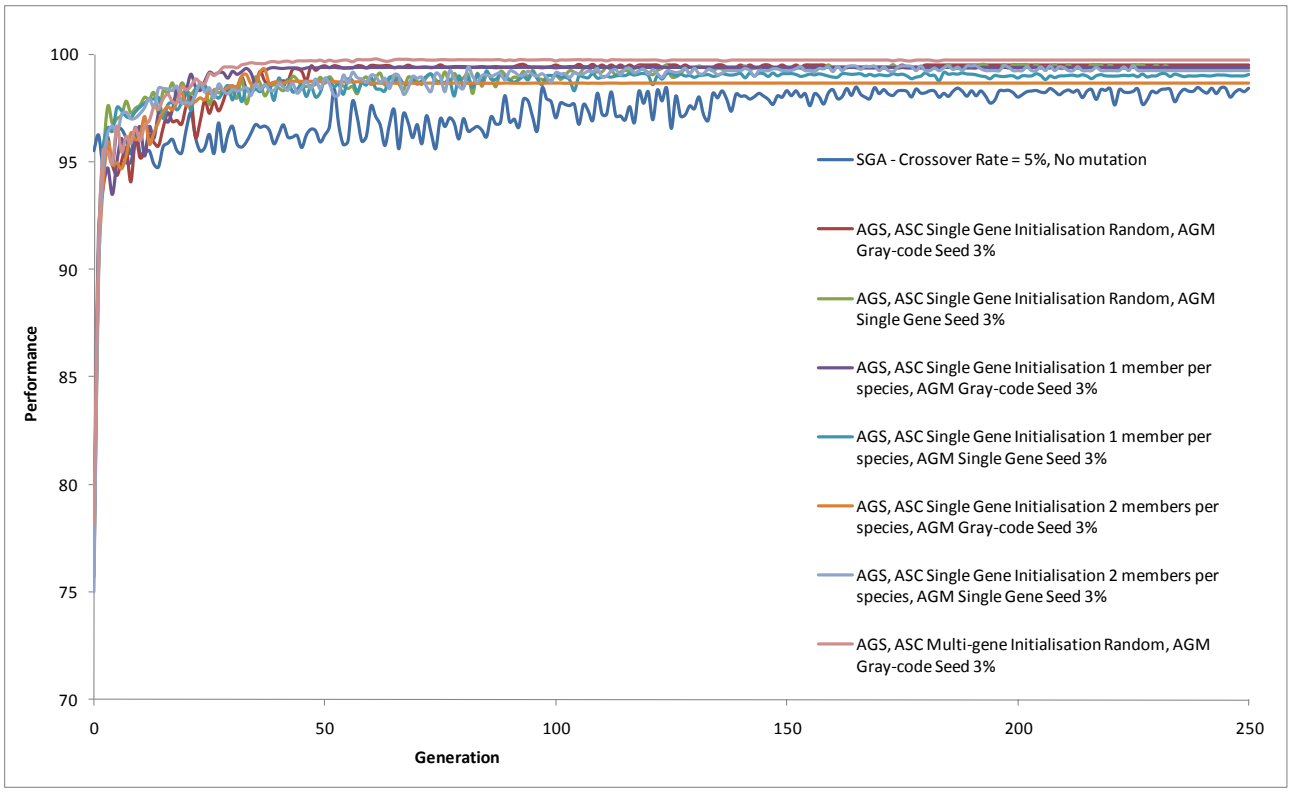

Figure 49: Multi-objective Problem Objective 1

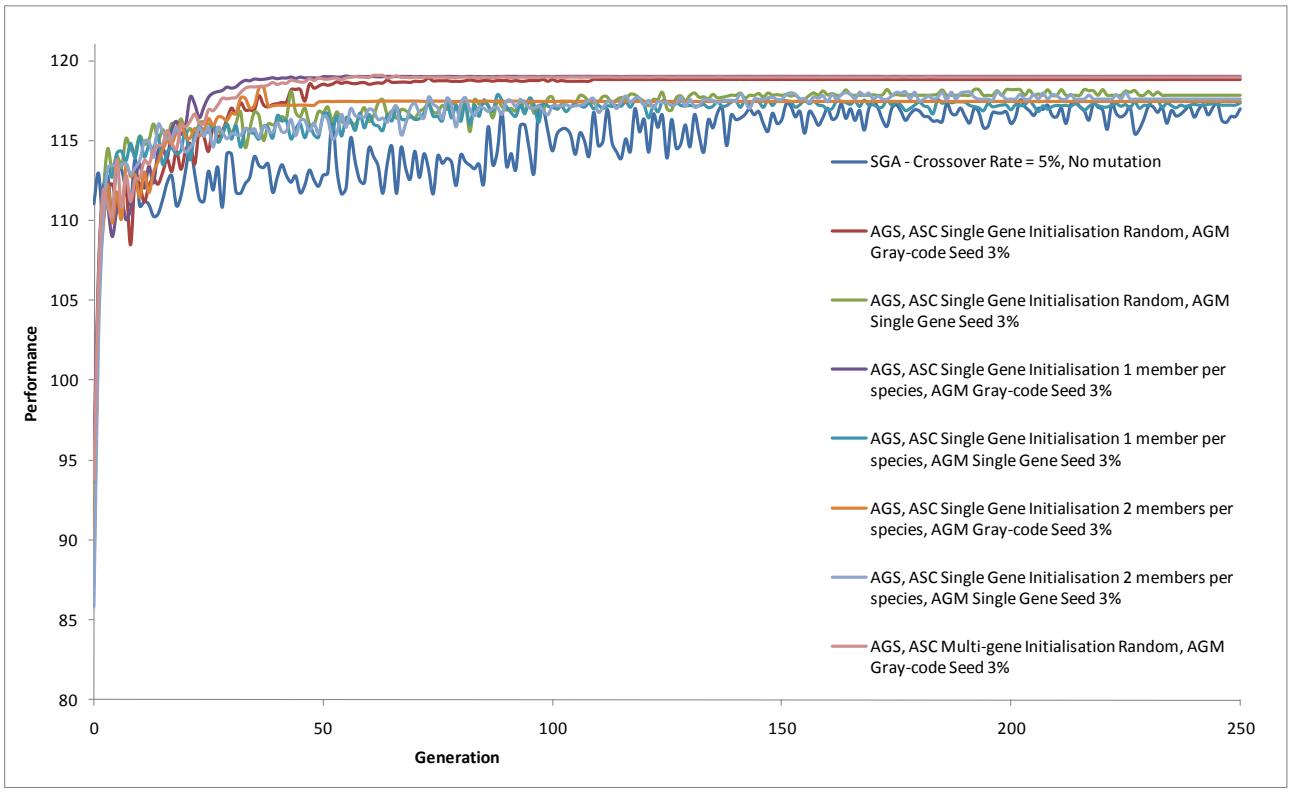

Figure 50: Multi-objective Problem Objective 2

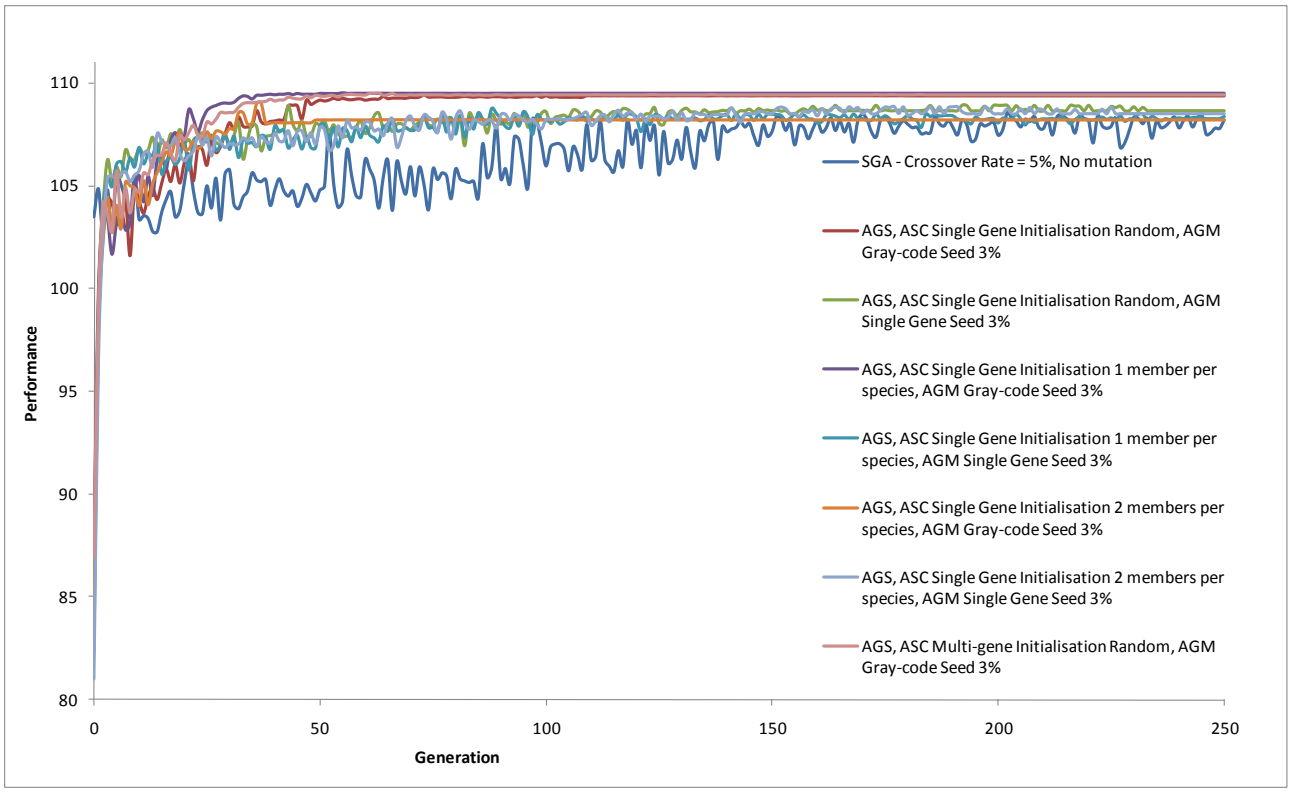

Figure 51: Multi-objective Problem Objective 3

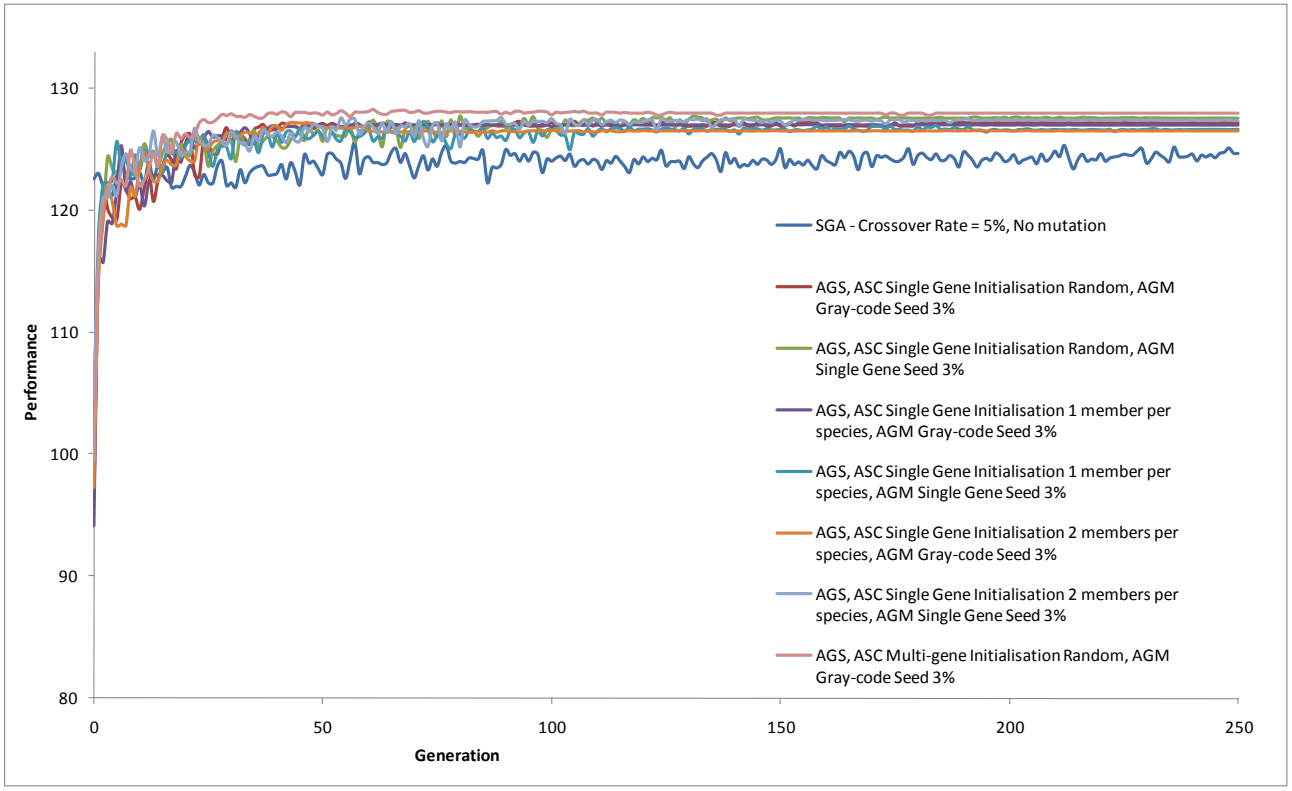

Figure 52: Multi-objective Problem Objective 4

The most common setups as shown in Table 10 for ASGA in the best sets of results were to use the ASGA's selection operator as opposed to the SGA's selection operator, a single gene

species tag rather than a multi-gene tag, and a mutation rate seed of 3% rather than a random seed or 50%. The species tag initialisation, choice of random, single member species, or double member species did not impact on whether the setup appeared in the set of the best results. Binary encoding of the mutation rate was found to be detrimental to performance. While integer and gray-code encodings of the mutation rate gave similar steady state performance, gray-code encoding gave a faster convergence on the final value. Integer encoding gives a faster initial convergence but convergence slows before reaching steady state performance.

Table 10 Best ASGA Combinations

AGS	ASC		AGM	
	Encoding	Initialisation	Encoding	Initialisation
Yes	Single Gene	Random	Gray-code	Seed 3%
Yes	Single Gene	Random	Single gene	Seed 3%
Yes	Single Gene	1 member per species	Gray-code	Seed 3%
Yes	Single Gene	1 member per species	Single gene	Seed 3%
Yes	Single Gene	2 members per species	Gray-code	Seed 3%
Yes	Single Gene	2 members per species	Single gene	Seed 3%

The ASGA crossover mechanism (ASC) was more successful when implementing the species tag as a single gene with a high number of alleles. While initialisation of ASC had little effect a random strategy will not suffer from deterministic problems incurred by poorly designed deterministic methods. For example, if the deterministic method has a single flaw that occurs during optimisation then every time the optimisation is repeated this flaw will occur. A random strategy might fail, on one iteration, but is unlikely to fail repetitively as each initialisation will be different. A random strategy was chosen given the lack of information regarding possible initialisation problems. Having multi-gene encodings for the species tag appeared to give little advantage compared to single gene encodings which appeared more prominently in results. This is indicative of the use of the species tag more as a pattern filter rather than a reflection of biological species, where some species are more likely to be able to successfully mate with other species.

The ASGA mutation operator (AGM) was found to work best when seeded initially with a fixed value. Furthermore, using gray-code numbers gave a faster convergence without loss of final solution quality. The fixed value for the seed used is consistent with commonly used values for implemented fixed rate mutation operators in the literature. The rate from the literature is typically between 3 to 5%. The ASGA uses a 3% mutation rate. This 3%

mutation rate changes rapidly during the evolutionary process. The advantages found in transient performance associated with gray-code numbers may be attributed to the small Hamming distance between successive numbers. This is beneficial in cases where changes should progress from one value to the next without large jumps and gray-code numbers can help genetic operators achieve this. For the AGM gray-code encoding with a fixed seed during initialisation was found to yield the best performance.

6.5. Chapter Summary

In this chapter, a set of 10 test functions were presented and used to test various ASGA operators against an SGA. Here, three operators are proposed, developed and then verified experimentally. The experiments were carefully chosen to verify the implementation and initialisation of each operator. As shown, the best results are achieved when a single gene is used for the species tag. The mutation operator is also found to work best when initialised with gray-code numbers having a seed value of 3%. The results show that the selection operator gives good performance when applied to single objective problems.

The chosen operators are tested further in Chapter 7 against leading algorithms from the literature. Chapter 7 focuses mainly on crossover and mutation, as the selection operator is found to perform poorly on multi-objective problems.

Chapter 7. Algorithm Comparison

This chapter evaluates and compares the genetic algorithm proposed in Chapters 4, 5, and 6 with other competitive genetic algorithms on multi-objective optimisation problems. Particular focus is placed on the multi-objective problems found in digital design optimisation as defined in this thesis. This chapter shows the proposed framework and ASGA to be superior to other classical methods for system design on FPGAs by illustrating that results obtained for large design problems such as the Extended Kalman Filter were better in a higher percentage of trials than for alternative algorithms.

7.1. Overview

This chapter sets out a methodology for comparing multi-objective optimization algorithms [**77**][**78**][**79**]. This methodology is an extension of the use of the Hypervolume Indicator for measuring multi-objective optimizer performance. An extension of the Hypervolume Indicator for use in template problems with unknown objective goal distributions is presented. Template problems refer to classes of problems, where given a set of parameters, an instance of a problem can be created. Such instances are related but have important differences. The presented analysis method makes no assumptions about the distribution of objective goals in a problem. Problem instances of a template problem are not assumed to have the same type of objective goal distribution. The problems encountered in this thesis do not have known distributions. Otherwise simpler optimization techniques could be used.

In this thesis four competitive genetic algorithms and one related genetic algorithm are identified. These algorithms are the Pareto Envelope-based Selection Algorithm (PESA) [**24**], 1-1 Pareto Archived Evolution Strategy (PAES) [**25**], Strength Pareto Evolutionary Algorithm 2 (SPEA2) [**26**], and the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [**16**]. Also tested is the Spears' Simple Subpopulation Scheme (SSS) [**70**] due to its similarity with the proposed crossover operator. PESA and PAES employ a crowding factor that reduces the likelihood of selection based on how many other solutions are nearby. PESA and PAES also select new individuals from an archive of non-dominated solutions. SPEA2 uses a fitness function called the strength value based on how individuals dominate each other to determine selection [**26**]. NSGA-II selects the best individuals in terms of dominance from each population in order to generate the next population. SSS employs species tags to control the crossover process.

The first comparison made in this chapter is on the well-known Knapsack problem. The type of knapsack problem used is a multi-objective d-dimensional 0/1 Knapsack problem as given in [**19**]. This represents a multi-objective problem with multiple constraints that is NP-Complete [**23**]. This is a difficult problem with no known simple solutions and is used for comparing genetic algorithms in [**17**]. This serves as the benchmark for the comparison of algorithms in this chapter.

The ASGA is then compared against the identified genetic algorithms on a number of selected digital design case studies to determine its performance capability for digital design. These are the design of a parameter estimation circuit in a Self-Tuning Regulator (STR), implementation of a Sum-of-Absolute-Difference (SAD) function for an 8 by 8 window, and the design of a 5-state Extended Kalman Filter (EKF). The parameter estimation circuit designed is for a second order implementation of the parameter estimation circuit for an STR for controlling plant processes. The SAD function is used in video motion detection but is used here as a large problem with a uniform substructure. The EKF is a desirable state observer to implement but costs are high and can impact on other elements of designs.

Based on performance comparisons a new combination of the ASGA with other identified genetic algorithms is proposed to achieve better results. This hybrid, entitled the "Hybrid-ASGA" (H-ASGA) is composed of the best elements of both the PESA and the ASGA. The PESA uses its own selection operator, but does not specify a crossover or mutation operator. The ASGA uses its own crossover and mutation operator. As such a hybrid combining PESA's selection operator with ASGA's operators was possible. The H-ASGA is then compared against the standard PESA and the ASGA to test the effect of combining the operators from each.

7.2. Methodology for Comparisons of Optimisation Algorithms

7.2.1 Problems in Comparison

The problems considered in this work are multi-objective problems and so there is seldom a clearly superior solution, but usually a set of equally or near equally good solutions. A multi-objective algorithm attempts to find all the non-dominated solutions in the solution space. This means that to compare two algorithms, a single decision must be made as to which is the better of two sets of solutions, each of which represent different objective trade-offs. Objectives are not interchangeable and must retain their meaning. This excludes a weighted

sum approach that assumes that objectives are interchangeable but at different rates. Also, when comparing, sets of solutions must be considered rather than a single best solution. To make a decision, these objectives and sets of solutions must be reduced to a single decision variable with a calculable threshold for determining whether results are significantly different and if so which is the better algorithm.

Many problems have parameters that define differences in problems with a common theme but these differences must be accounted for when making comparisons. Existing methods do not necessarily account for the influence of parameters in evaluating algorithm performance on parameterized problems. Such methods may, for instance, calculate the average of an objective value from data with different values of parameters for the problem. While the distribution of objective values to a problem may be similar enough and give sufficiently accurate results, this is not necessarily true in general. Different parameter values may give different distributions particularly where such procedures become skewed by a smaller subset of the data rather than being representative of the entire sample or the population from which the sample comes.

Design problems that incorporate optimal scheduling and partitioning can be NP-Complete, meaning that non-deterministic optimisation algorithms must be used [**23**]. Hardware Software partitioning is NP-Complete [**80**]. The capability of a single tool to support both hardware and software should enable more integrated design, and consequently lower costs. NP-Complete problems are solvable in polynomial time by a non-deterministic Turing machine [**23**]. However, NP-Complete problems have no polynomial time deterministic algorithm that is known for solving them [**23**]. The lack of determinism of the algorithms used to solve these problems means that proofs are 'practically impossible' to obtain [**51**], leaving only empirical studies. Empirical studies use samples of the population of possibilities to draw conclusions about the entire population. While such empirical studies are never 100% certain, they help establish a high degree of confidence in the results. All problems associated with calculation of statistics, sampling of a population, and experiment design apply here.

7.2.2 Background

The quality of solutions obtained from an algorithm is determined by how close to the pareto-optimal front the algorithm is able to reach and the coverage of possible solutions obtained [**16**]. Both are important factors as closeness to the pareto-optimal front not only directly

affects design quality, but also its spread measures how many of the design options were discovered and thereby the quality of the algorithm. In a multi-objective problem multiple solutions are common. It is desirable to find as many of the pareto-optimal solutions as possible. Usually the location of the pareto-optimal front is unknown. The number of solutions on the pareto-optimal front is also usually unknown. These unknown factors mean that the spread of solutions obtained must be used to estimate the coverage of a problem.

The Hypervolume Indicator [**81**], measures the amount of the solution space dominated by the proposed solutions. The Hypervolume is the hyper-area of the search space dominated by the proposed solutions relative to a common reference point. Formally, the Hypervolume is the hyper-area of set union of the hyper-cubes bounded by the proposed solutions and a common reference point. The reference point must be set to be poorer in every objective compared to all proposed solutions. The Hypervolume Indicator does not require knowledge of the pareto-optimal front in order to be calculated. In general the pareto-optimal front of a problem is not known.

Distribution-free statistical techniques [**50**], are methods of conducting statistical tests on distributions of problem data without knowledge of the distribution type. Distribution-free methods are not dependent on the distribution of the data the test is conducted on. A normal test, for instance, is only valid for a normal distribution or distributions that are approximately normal. A distribution-free test however can be applied to any type of distribution. Dedicated techniques offer better performance in analysis but may not be available. Distribution-free methods are important in many practical problems where the distribution of the problem is often unknown.

A sign test is a distribution-free statistical technique, which converts tests of unknown distributions to a test against a binomial distribution with probability $p = 0.5$ in order to determine if two populations of data are different [**50**]. Data from each population are organized into pairs for this test with the test statistic being the sign of the difference of each pair. The basis of this approach is that if each population is identical, the chance of a value from one population being higher than a value from the other population when selected at random is equal. This chance, provided replacement is observed, does not change with subsequent trials. This fulfils the requirements of a binomial distribution. The probability p is obtained from the equal probabilities of one result being higher than another and assuming the chance they are equal is minimal.

7.2.3 Experiment Design

The problem of multi-objective optimization performance analysis may be described as follows: "to decide from two non-deterministic optimization algorithms, which performs the best for a given class of instances of a multi-objective problem of unknown distribution". This problem has a number of elements which allow the application of the approaches in the background Section 7.2.2. These are applied before the experiment format is given.

The problem is qualitative in that it is only necessary to determine which algorithm performs best and not by how much. A quantitative comparison on the other hand shows how much of a difference exists between the algorithms. The test data for such an analysis is the objective values of solutions obtained from each algorithm. Such a quantitative comparison usually requires that the distribution type of the objective values from the solution sets of the algorithms be known, as such it cannot be used in this case because the distribution types of the design problems are not known.

An empirical experiment is used to answer this problem, since a theoretical proof is impractical given the number of states required to be taken account of [**51**]. A sample size of 30 or more is common with a degree of confidence of 99%. Given that differences in results are small and the threshold value decreases with increasing sample sizes, a sample size of at least 100 is used as this allows smaller differences to be detected.

In this experiment, the distribution types of objective values from solutions obtained from the algorithms are not known. The problem must be reformulated in a way such that the distribution type for the test is known. For this it is shown that the algorithms cannot be assumed to be identical. If both algorithms were identical, then since they are non-deterministic, they would have an equal probability of producing a better result and this probability would not be trial dependent. This describes a binomial distribution with probability $p = 0.5$. The goal is therefore to use the results to determine which outperforms the other. This is a sign test.

The Hypervolume Indicator is used to determine which set of solutions is the better of the two sets obtained from the algorithms tested. The Hypervolume Indicator converts a set of multi-objective solutions to a single value. The higher of the two Hypervolume Indicators is the better result. The conversion of the solution sets from the tested algorithms to a single value is essential as the test used must be able to pick the better result without the possibility

of another pareto-optimal case where each algorithm excels at different goals. Hypervolume is not the only method for performing this task, but is used in the literature that this work makes comparisons with, and hence is adopted here [**81**].

The approach is then extended to the case of parameterized problems. Parameterized problems are problems where a template defines a class of related problems. A set of parameters is applied to a template to instantiate any member of the problem class. For instance, a Knapsack problem is a class of related problems where the set of capacities, rewards, and weights is a set of parameters. With this set of parameters any instance of the knapsack problem can be instantiated from the template. The testing of the optimization of parameterized problems can be done by ordering each sample of solutions by their parameter values. Each algorithm generates a response to each of 100 different parameter sets. This gives 100 samples.

To make a comparison two algorithms are selected and then samples for each of the 100 parameter sets are compared. The algorithm that performs better in each parameter set is noted. For statistical comparison the test statistic is the number of times an algorithm performed better than the algorithm it is being compared against. Intuitively the higher the value the more likely the algorithm is to generate a result that is better than the alternative algorithm. In order to plot the result this is converted to a percentage of trials in which the ASGA is better. A threshold may be calculated which must be exceeded for the test to be statistically significant. The threshold formula of Equation (20) assumes a normal approximation to the binomial distribution for the test statistic.

$$threshold = \left(0.5 + Z * \sqrt{\frac{0.25}{n}}\right) \times 100\% \quad \textbf{(20)}$$

Here Z is the Z-value found from normal distribution tables and *n* is the sample size. For 100 samples and a 99% degree of significance the threshold is 61.63%. This means that if the percentage is 62 or higher then ASGA has performed statistically significantly better than the reference algorithm in the key.

7.2.4 Approach

Case studies for evaluating performance should be of comparable complexity to problems that the proposed ASGA will be applied to in practice. Ideally, a case study problem should not have trivial solutions as most algorithms will find these without difficulty. If the problem

is not difficult then there will be little spread in results between the algorithms being compared, making discerning performance differences difficult.

To assess how the ASGA performs, the Selection, Crossover, and Mutation operators described in Chapters 5 and 6 need to be evaluated separately as well as together to determine the importance of their interactions. Evaluation of separate operators and combinations of operators allows tests to discern if any performance increase is due to a single operator, only two of the operators, or requires all three operators to be used. The contribution of each operator can then be determined, which is important when considering operators for use by other researchers in algorithm design.

The running time of a genetic algorithm for complex problems may be measured by the number of unique individuals evaluated. The time taken to evaluate individuals is the most significant time requirement and previously encountered solutions do not need to be re-evaluated. Usually a database of previously encountered solutions is kept. This database allows the objective function values for a solution to be found in the database rather than re-evaluated. Re-evaluation is as expensive to perform as the initial evaluation of a solution and should be avoided where possible. The number of designs an algorithm will evaluate is never large due to the times taken to evaluate a solution. A database for storing previously evaluated individuals need not be complex or costly in terms of computing resources since the amount of data stored is always small.

7.2.5 Algorithms used in Comparison

To compare the effect of different operators a reference that represents the standard operator is required. In this case three standard operators are required as references for Selection, Crossover, and Mutation. When making a comparison either the ASGA operator is used or the appropriate reference operator is used. For example, a genetic run may use either ASGA selection or reference selection, ASGA crossover or reference crossover, and ASGA mutation or reference mutation. These references must be chosen appropriately.

The reference selection operator used when not using the ASGA selection operator is a binary tournament selection operator using the pareto-dominance rank to select individuals and using the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [**16**] crowding distance metric to select individuals in the event of solutions with equal rank. Pareto-dominance rank [**15**] is a common approach that forms the basis of many other multi-objective selection

operators. In this approach all non-dominated solutions are assigned a rank of 1 and removed from the population. All non-dominated solutions in the remaining population are assigned a rank of 2 and removed from the population. The process is repeated with successively higher ranks until every solution has been assigned a rank. All solutions of rank 1 are better than any solution of rank 2 or higher. All solutions of rank 2 are better than any solution of rank 3 or higher and so forth.

NSGA-II [**16**] uses a crowding distance metric to define how unique a solution is. Objective values in the population are first sorted. For any solution, the objective values on either side of the solution are used to define a hyper-cube. For example, given two objectives A and B where objective values present for A are [1 3 4 6 8] and for B are [3 5 9 10 13] then for an individual with A = 4 and B = 10 the hyper-cube is defined by A = 3, B = 9 to A = 6, B = 9 to A = 6, B = 13, to A = 3, B = 13 to A = 3, B = 9. The hyper-length of the edges of this hyper-cube once scaled in each dimension is the crowding distance of the solution. Figure 53 shows an example for two objectives F_1 and F_2 with the crowding distance shown as a dashed line for solution A. The larger the crowding distance the further away from similar solutions a proposed solution is.

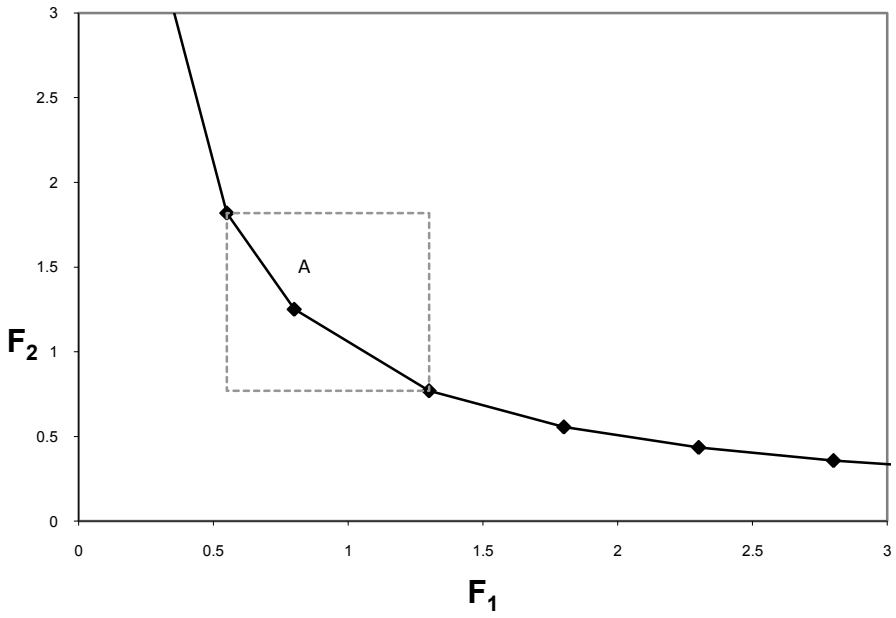

Figure 53 Crowding Distance

PESA's selection operator, which uses binary tournament selection to select solutions from an archive of non-dominated solutions, selects the individual with the lower Crowding Factor. The Crowding Factor is the number of neighbours a solution has in its neighbourhood. A neighbourhood is determined by dividing the solution space into equal

sizes regions with each region being a neighbourhood. Figure 54 shows the same set of solutions as Figure 53 but this time the space is divided into neighbourhoods. Three different neighbourhoods are labeled *a*, *b*, and *c*. Since *a* has no solutions in it, this neighbourhood could never be selected. *b* has two solutions and so has a crowding factor of 2. *c* has 1 solution and so has a crowding factor of one. During tournament selection the solution with the lowest crowding factor is selected. In this case *c* would always be given preference to *b*.

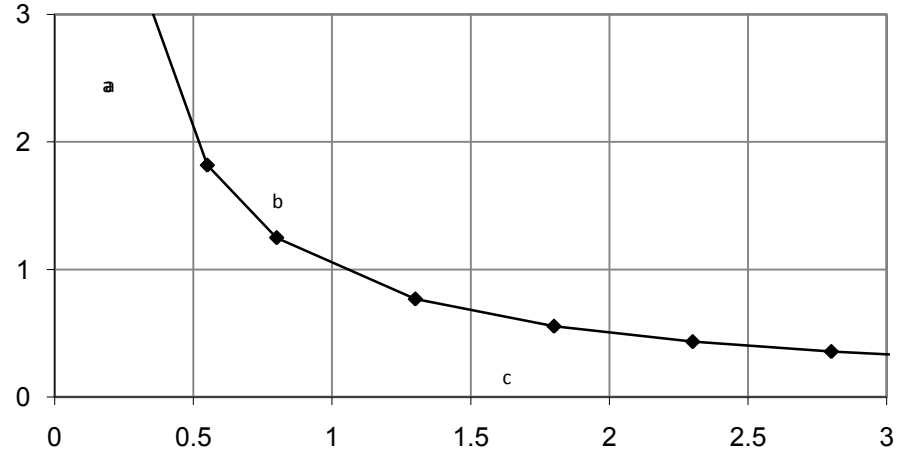

Figure 54 Crowding Factor

The crossover operator used as an alternative to the proposed ASGA crossover operator is uniform crossover with 2 parents and retaining both children. In uniform crossover equal proportions of each parent contribute to each child but the selection of genes in crossing does not depend on gene location in the chromosome. Table 11 shows an example set of parents. All genes in parent A are equal. All genes in parent B are equal. The genes are only equal to make the origin of genes in the children of these parents apparent. Table 12 shows one possibility for the children of the two parents in Table 11 after uniform crossover.

Table 11 Parents

A	1	1	1	1	1	1	1	1
B	2	2	2	2	2	2	2	2

Table 12 Children

A	1	2	1	2	2	1	2	1
B	2	1	2	1	1	2	1	2

The mutation operator used as an alternative to the ASGA mutation operator is a fixed rate mutation where mutated genes are assigned a new allele from the other alleles the gene may take. Under this scheme every gene has a fixed chance of mutating. Should a gene mutate then for this work it is assigned a new allele from the other possibilities. Table 13 shows a before and after case for an individual subject to fixed rate mutation. In the example two genes were mutated and changed from 1 to 0.

Table 13 Mutation

Before	1	1	1	1	1	1	1	1
After	1	0	1	1	1	0	1	1

7.3. Comparisons on a known problem

The goal of this comparison is to compare the performance of the proposed ASGA with other leading algorithms on an NP-Complete problem which is known to be difficult to optimise. In addition the problem selected is a multi-objective problem with multiple constraints and is also parameterised. Design problems which are a primary target for this optimisation algorithm are commonly multi-objective problems subject to multiple constraints on resources, time, and power with design tools typically being required to process different design problems. The Knapsack problem and design problems share these elements.

7.3.1 Multi-objective d-dimensional 0/1 Knapsack Problem

The Knapsack problem is simply how to fill a given number of knapsacks with items of various values and weights such that the value of items in the knapsacks is maximised without exceeding the weight capacity of any knapsack [**19**]. Formally this is described as

For a set of m items and n knapsacks where

$$r_{i,j} = reward\ for\ item\ j\ for\ knapsack\ i$$
$$w_{i,j} = weight\ of\ item\ j\ for\ knapsack\ i$$
$$c_i = capacity\ of\ knapsack\ i$$

find a vector

$$X = [x_1, x_2, \dots x_m] \in \{0,1\}^m$$

such that

$$\forall i \in \{1,2,\dots,n\}: \sum_{j=1}^{m} w_{i,j} \cdot x_j \leq c_i$$

and

$$f(X) = \{f_1(X), f_2(X), \dots, f_n(X)\}$$

is maximized where

$$f_i(X) = \sum_{j=1}^{m} r_{i,j} \cdot x_j$$

$f_i(X)$ are the objective functions and c_i are the constraints. The parameters are $r_{i,j}$, $w_{i,j}$, c_i, m, and n.

7.3.2 Results

A 2 objective knapsack was used with a choice of 100 items in order to limit the computational time of comparing the different combinations of the proposed ASGA's operators and other leading algorithms. Items and knapsack capacities were initialised as in Zitzler et al's work [**81**]. This size problem remains comparable to problem sizes encountered in typical design problems. 500 generations were calculated following the format in Zitzler et al's experiments [**81**]. These setup conditions were chosen so as to generate comparable results with Zitzler et al's work for evaluation of algorithm performance [**81**].

Table 14 gives the rankings of each algorithm for a population size of 100 after 50,000 function evaluations. NSGA-II gave the best performance while SPEA2 gave the worst performance from those tested. The algorithms NSGA-II, PESA, and 1-1 PAES all employ dominance and diversity control mechanisms in their selection operators. Since the test used in ranking employs the Hypervolume Indicator this may be why these algorithms performed well. Hypervolume rewards diverse solutions near the pareto-optimal front more than solutions near the pareto-optimal front in general. NSGA-II allows individuals that have pareto-dominance ranks higher than 1 to reproduce provided the population size has sufficient space. PESA and 1-1 PAES do not allow this. In this case this may have allowed NSGA-II to outperform these algorithms by maintaining a greater diversity.

Table 14 Ranking for Population Size 100

Rank	Algorithm
1	NSGA-II
2	PESA
3	1-1 PAES
4	ASGA
5	SPEA2

The population size was then increased to 2,000 while retaining the limit of 50,000 function evaluations to determine if population size influenced algorithm performance. The results are shown in Figure 55 where the performance of all other algorithms are benchmarked against the proposed ASGA. Whenever the values shown are greater than 50% the ASGA performed better in more trials but values must be greater than 61.63% to be statistically significant (see

Section 7.2.3). The ASC shows the result of only using the ASGA's crossover operator, while the AGM shows the result of only using the ASGA's mutation operator. At this level interest is only in which algorithm performed the best. In this case the combination of the ASGA's crossover and mutation operators gave equal best performance with the PESA (Figure 55). The difference between the ASGA and the PESA is only significant at the 90% level of significance. 1-1 PAES was not tested at this level due to long running times produced by an inability to escape regions that had already been evaluated. The Simple Genetic Algorithm (SGA) [**15**] and Spear's Simple Subpopulation Scheme (SSS) [**70**] were substituted in place of PAES. The ASGA's selection operator was found to bias search to the mid region of pareto-optimal fronts demonstrating poor suitability to multi-objective problems. It was found that the crossover and mutation operators as designed in Chapter 6 gave more effective optimization on the knapsack problem due to their ability to refine and identify genetic patterns. The ASGA selection operator performed poorly and was substituted with a pareto-dominance ranking method. The ASGA selection operator would need redesigning for multi-objective problems.

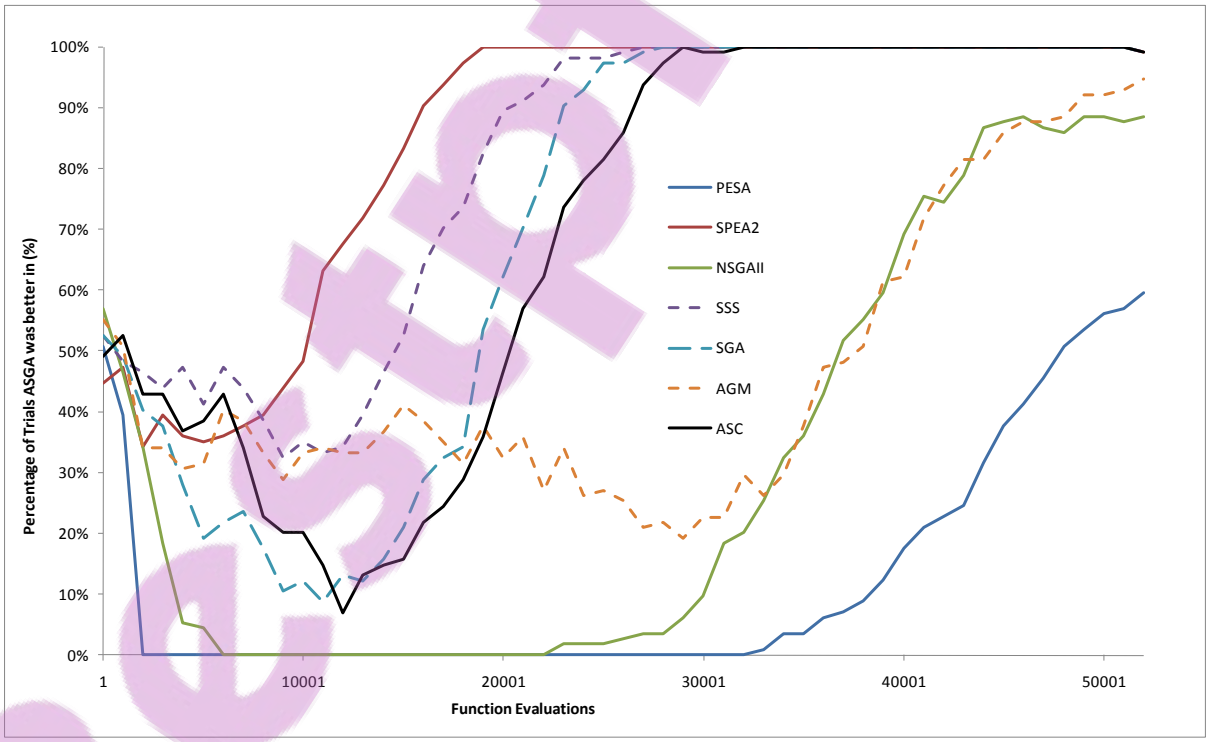

Figure 55 The ASGA Performance against other Genetic Algorithms

Population size was still found to influence performance when the total number of tried individuals was kept constant. This means that population size can determine algorithm performance on a problem even when algorithm processing times are kept constant. When keeping the processing times constant an increase in population size must be accommodated by a decrease in the number of algorithm iterations or generations. The trade-off between population size and number of generations is therefore important in determining the success of different genetic algorithms.

The ASGA has a longer initial delay before making progress at solving problems. This is shown in Figure 55 by an early decrease in performance compared to other algorithms. This can be attributed to the ASGA's crossover and mutation operators. Both these operators are adaptive and take time to adapt to the problem. However, once the ASGA's operators have adapted, the ASGA is able to take back the ground it has lost and exceed the performance of its competitors. Consequently, it is important to allow sufficient resources for the ASGA's operators to be able to adapt to a problem otherwise performance increases will not be possible.

7.4. Comparisons on Design Case Studies

This section presents a comparison of the ASGA on a range of design problems encoded in the design framework presented in Chapter 3. In addition to the comparisons, sample solution sets are presented to indicate the design solutions obtained from the ASGA. The encoding used for the genome for these case studies may be found in Appendix A.

7.4.1 Self Tuning Regulator

The ASGA is compared with other leading algorithms on the example design problem of the operation scheduling and resource allocation of a parameter estimation circuit for a Self-Tuning Regulator which is introduced in Chapter 3. The pre-processing stage given in Chapter 3 was performed manually to reduce the decision set to the allocation of resources to dividers and multipliers. The additions and subtractions were allocated separate components for each operation as generally multiplexer costs outweigh any savings by sharing these components. Table 15 shows the logic cells usage, embedded multiplier usage, and sampling period of a parallel and serial implementation in the proposed design framework. The costs of adders and subtractions are not included as these are constant following the pre-processing stage. Due to framework implementation the sampling period represents the maximum and

minimum possible values. The embedded multipliers also represent the maximum and minimum possible values. However, since logic cell usage depends on the components used and their costs for sharing, this column does not necessarily show the maximum and minimum possible values. The logic cell usage figures however can still serve as a guide of magnitude.

Table 15 STR Design Range

Design	Logic Cells	Embedded Multiplliers	Clock Cycles per sample
Parallel	23240	352	1
Serial	5232	8	46

Although the pareto-optimal front is a three dimensional surface, the most apparent trend was that the sampling period could be reduced by allocating more logic cells to the problem. The pareto-optimal front is concentrated in a narrow strip of the solution space and trends are most easily observed by plotting two dimensions at a time. Figure 56 shows the sampling period and logic cell usage of designs from the pareto-optimal front of the ASGA for a single evolutionary run. There is a general trend showing a decrease in sampling period as the number of logic cells increases. This means that as components are progressively assigned fewer operations they finish earlier driving the sampling period down. However, all operations must be performed, thus more components are allocated driving the logic cells usage up.

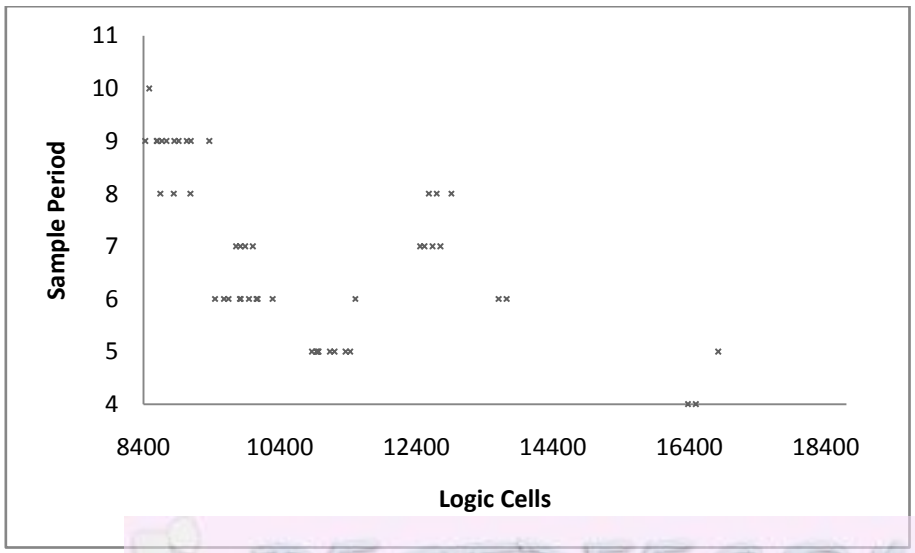

Figure 56 STR LUTs and Sampling Period

Figures 57 and 58, show different objectives for the obtained solutions although no apparent trends are observable for these objective combinations. In comparison to Table 15 there is a good coverage over the range of logic cells, while sampling periods represent a range of quicker designs with multiplier usage in the mid-range.

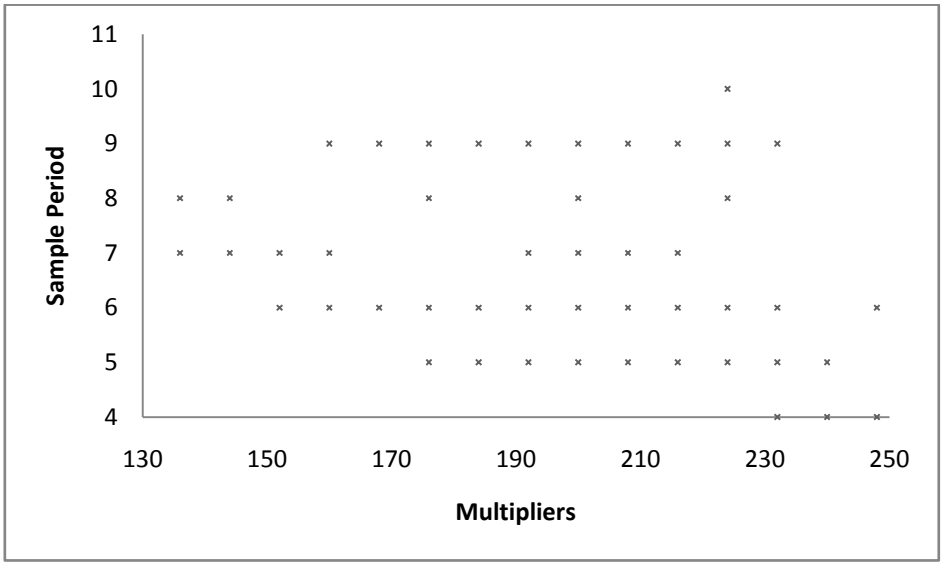

Figure 57 Multipliers and Sampling Period

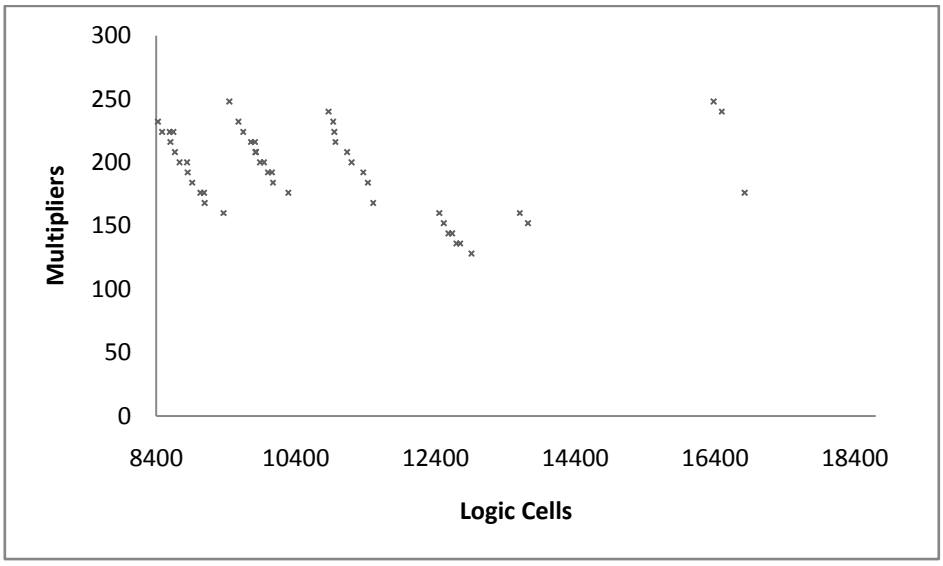

Figure 58 Logic Cells and Multipliers

Table 16 gives the solutions obtained from the ASGA for a single evolutionary run. This shows that 49 different solutions were obtained from one single run of the ASGA. These were plotted in Figures 56 to 58. Each solution in Table 16 is not dominated by any other solution in Table 16, and in addition every other solution that was encountered during the search process is dominated by at least one solution from Table 16.

Table 16 STR Solutions

STR					
Logic Cells	Multipliers	Clock cycles per sample	Logic Cells	Multipliers	Clock cycles per sample
16496	240	4	8482	224	10
16380	248	4	12452	160	7
16824	176	5	8602	216	9
18704	232	4	8422	232	9
10948	224	5	9824	208	6
10932	232	5	10054	192	6
11362	192	5	8850	192	9
8666	208	9	9576	232	6
13604	160	6	9084	176	8
11130	208	5	8914	184	9
11428	184	5	10070	184	6
11504	168	6	9754	216	7
11196	200	5	8732	200	9
9810	216	6	12582	144	8
9446	248	6	9818	208	7
10964	216	5	8644	224	8
10866	240	5	12696	136	8
12750	136	7	9886	200	7
12634	144	7	8840	200	8
10290	176	6	9092	168	9
9942	200	6	9030	176	9
13720	152	6	9644	224	6
12912	128	8	12516	152	7
9362	160	9	8590	224	9
10000	192	7			

Figure 59, shows that the combination of the ASGA's crossover and mutation operators outperformed all other algorithms except the PESA. The results for SSS, although close, are still statistically significantly different. Of note is that the ASGA crossover operator (ASC) appears to have contributed the most to the improved performance of the ASGA.

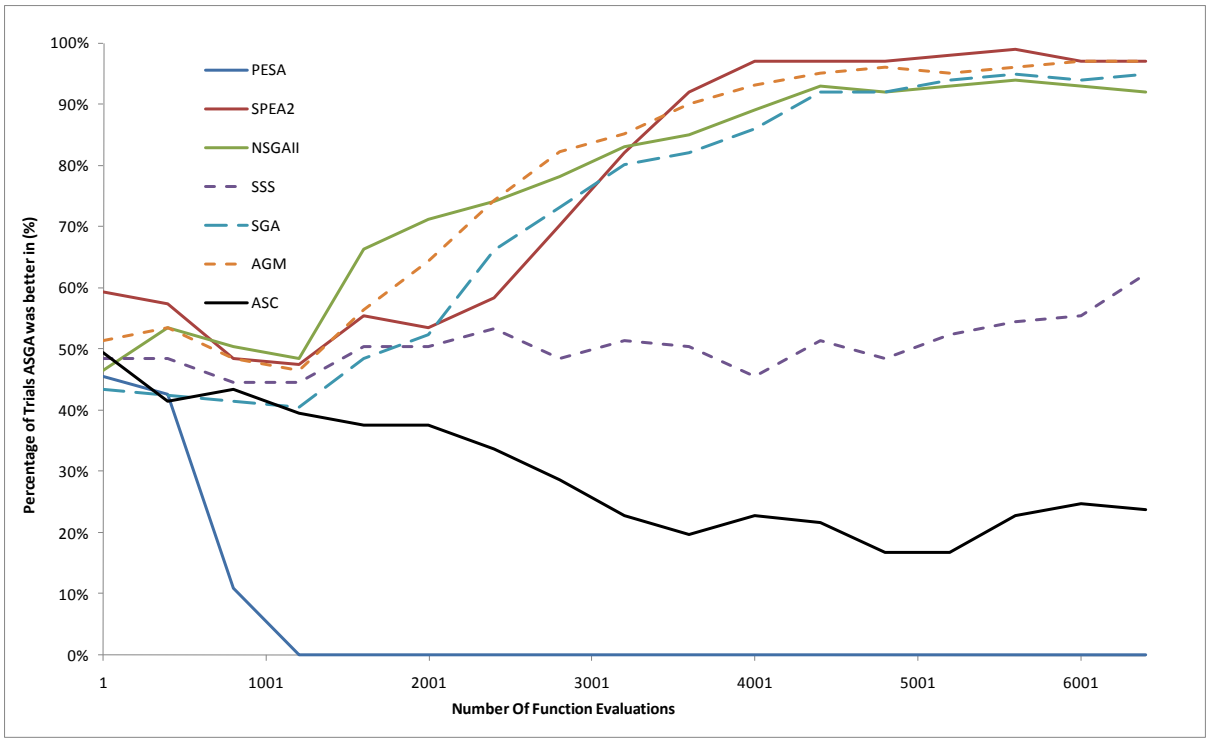

Figure 59 STR Problem

The solutions obtained are not dominated by a similar Self-Tuning Regulator from [**82**] in which a single multiplier function is used. The example from [**82**] is of the entire STR and requires at least 60 cycles per sample but only uses 4128 logic cells. The slowest design that can be represented by our framework, for the parameter estimation circuit only, takes 5232 logic cells and 46 cycles per sample. Since the parameter estimation circuit is the most costly part of the STR, these values are comparable.

In this case the advantages of the crossover operator and mutation operator in more effectively rearranging and identifying genetic sequences have not been sufficient to overcome PESA's selection strategy of choosing from an archive of non-dominated solutions based on crowding factor. In the cases of the other algorithms the advantages of ASGA's crossover and mutation operators are significant.

7.4.2 Sum of Absolute Differences

For this test the SAD function defined in Chapter 3 is optimised for an 8 by 8 window size. The component library has an absolute difference function and an adder function, which are used in this case study. No embedded multipliers are necessary and so this problem has only two objectives. A parallel and serial implementation profile are given in Table 17. Serialization of data through a component requires multiplexers to resolve input conflicts and

output registers to store intermediate results. In this case the cost of multiplexers and output registers for serialization has exceeded the savings from reusing components. The serial design both requires more logic cells and a higher number of clock cycles per sample.

Table 17 SAD 8 by 8

Design	Logic Cells	Clock Cycles per sample
Parallel	2224	1
Serial	3053	66

An example run of the ASGA produced two solutions to the SAD function problem. Table 18 gives these two solutions. The fastest design in table 17 is still better than either of these solutions but the solutions proposed by ASGA are close. It should be noted that the fastest design was produced with problem dependent knowledge about the SAD problem whereas the ASGA produced its solutions without this knowledge.

Table 18 ASGA solutions

Logic Cells	Clock Cycles per sample
2387	6
2474	5

The ASGA's combined crossover and mutation operators outperformed the SPEA2, NSGA-II, SSS, and the SGA but gave inferior performance against the PESA and the 1-1 PAES as shown in Figure 60. The use of archives of past solutions has yielded better performance than focusing on manipulating genetic patterns. In this case a smaller problem would make the influence on an archive more pronounced and here it is greater than the effect of ASGA's crossover and mutation operators.

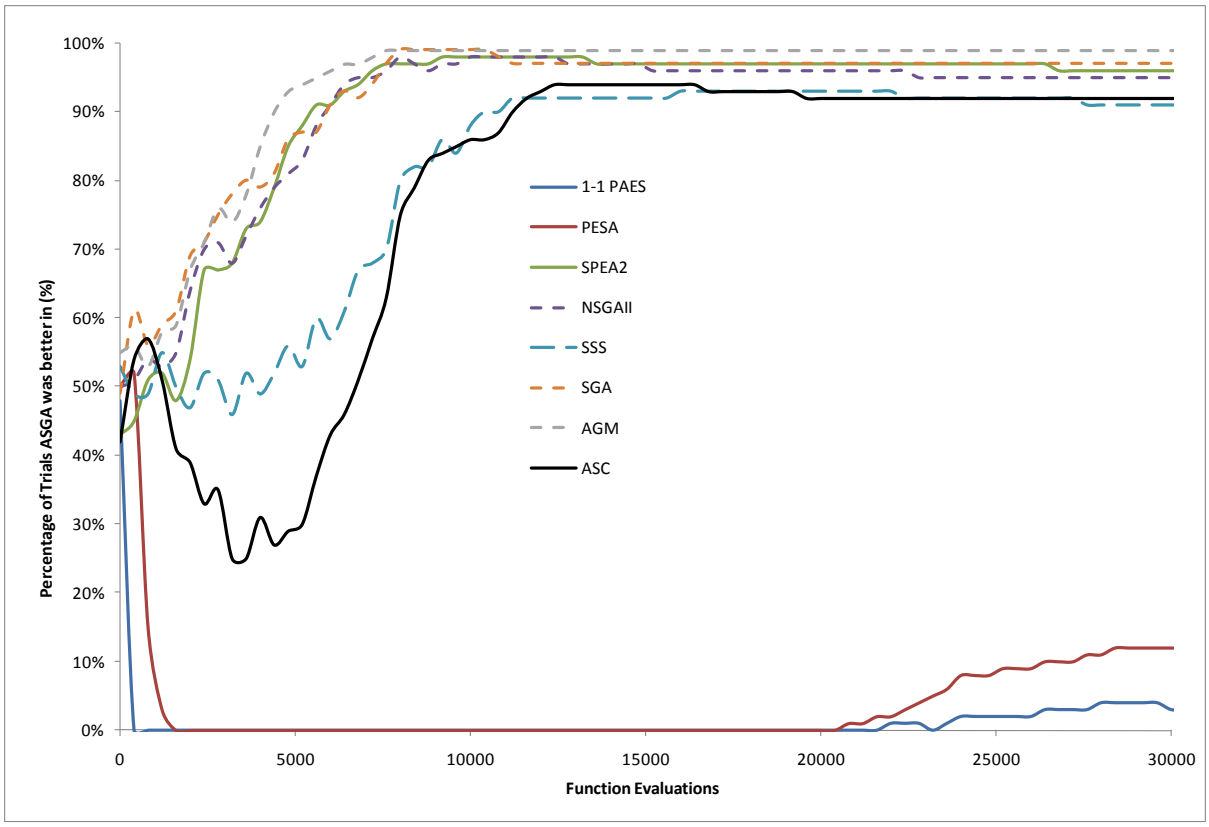

Figure 60 SAD Problem

7.4.3 The Extended Kalman Filter

The Extended Kalman Filter (EKF) implemented here is taken from [**22**] where it is part of a motor controller for sensorless control of an AC cage induction motor. The EKF implemented features 5 motor states that are estimated. Chapter 3 gives a general model for an Extended Kalman Filter. The EKF design is converted to the resource allocation and scheduling of the multipliers and dividers present in the design. Adders and subtractions are not included in the following costs. There are 405 multiplications and 4 divisions in the EKF considered. Table 19 gives a parallel and serial design implementation.

Table 19 EKF

Design	Logic Cells	Multipliers	Clock cycles per sample
Parallel	7457	810	1
Serial	20385	2	407

The algorithm shows a set of obtained solutions where the prevalent pattern is that the number of embedded multiplies decreases as the number of lookup tables used increases, as shown in Figure 61. This trend arises simply because in order to reduce the number of

multipliers it is necessary to share multiplier resources amongst higher numbers of tasks. The tasks must be able to supply their data to the multiplier resources. This requires larger multiplexers, implemented in logic cells. As the number of multipliers is reduced, the costs of these multiplexers increases, which in turn causes a rise in the logic cell cost of the system. Figures 62 and 63 show the other objectives for the obtained solutions. These graphs show clusters in the sampling period around 90 and 105 clock cycles. Due to the trend in shown Figure 61, Figures 62 and 63 are simply mirror images of each other. The two clusters denote two distinct solutions, which cause the spreading of results in Figure 61. The two solutions represent a faster expensive option and a slower cheaper option. Figure 61 shows a progression from one solution to the other.

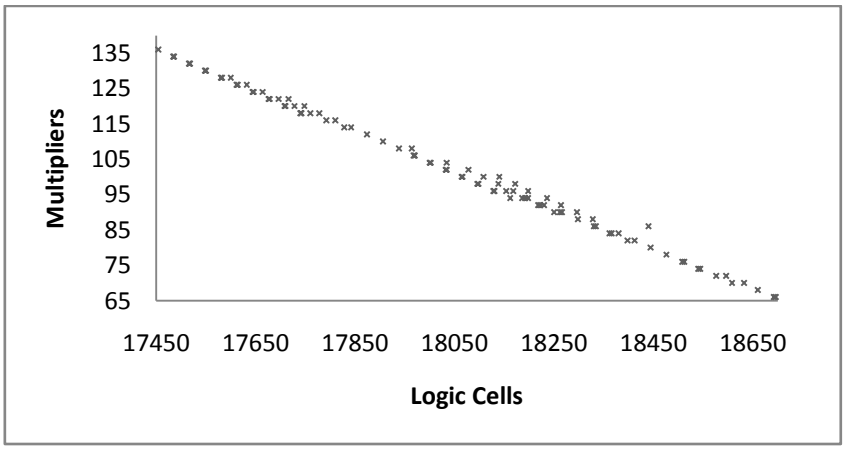

Figure 61 Logic Cells and Multipliers

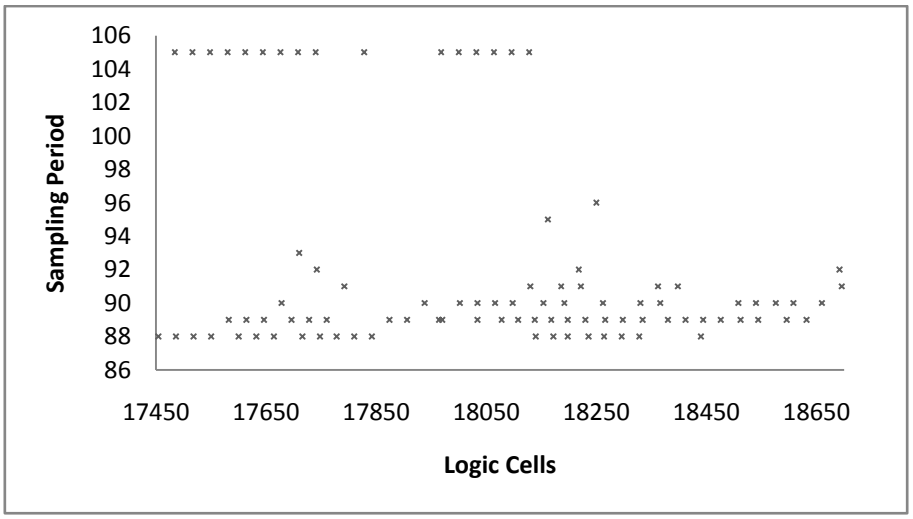

Figure 62 Logic Cells and Sampling Period

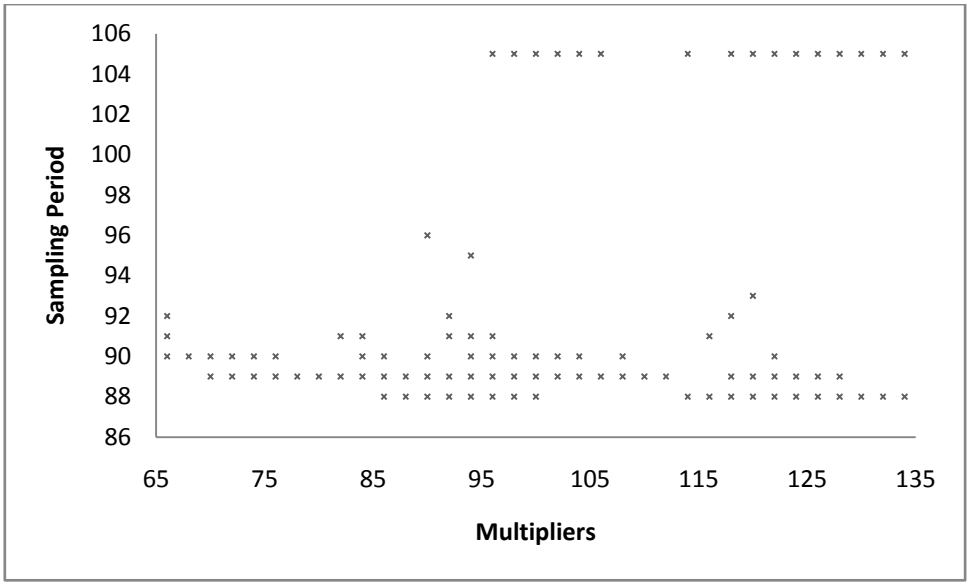

Figure 63 Multipliers and Sampling Period

Figure 64 shows that the ASGA outperformed all other algorithms. These results are statistically significant at the 99% level of significance. Although the 1-1 PAES and the PESA performs better than other algorithms they were still statistically significantly worse than the ASGA. Due to problem size an archive is less of an advantage as effective manipulation of DNA becomes more significant. The ASGA has greater performance in this larger problem.

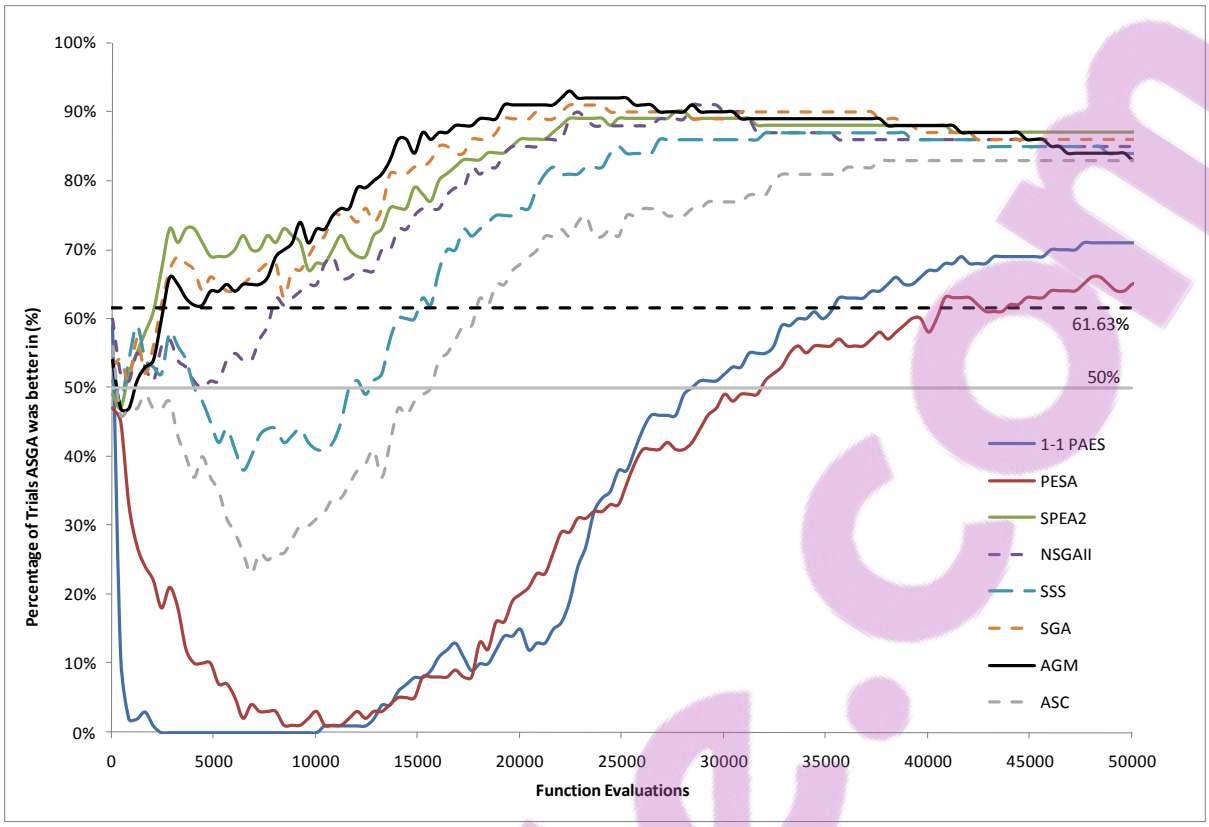

Figure 64 EKF Results

7.5. Hybrid-ASGA

In the previous work the PESA and the proposed ASGA have been found to be the best algorithms over a wide range of problems. The PESA uses reproduction to improve performance while the proposed ASGA uses recombination. Consequently it is possible to construct a hybrid algorithm, here labeled the H-ASGA, which combines the benefits of both approaches. The PESA stores an archive of every solution it is found that is not dominated by any other solution that it has found. The PESA then makes its selections of individuals for crossover and mutation from this archive. Some modification to the PESA's archiving procedure is necessary to allow ASGA's operators to work with archived solutions. In this case a more recent individual, where the functional genes are the same as those of a member of the archive, replaces that member in the archive. This updates the crossover and mutation genes of members of the archive. Functional genes are those genes that influence the phenotype of an individual. Crossover and mutation genes affect how the genetic process alters individuals but not the phenotype of an individual. The phenotype of an individual is its set of objective function values. The H-ASGA is composed of the front end of PESA, the PESA selection operator, and the ASGA crossover and mutation operators.

The H-ASGA yielded improved performance on the Knapsack problem for smaller numbers of function evaluations, but showed degraded performance for higher numbers of function evaluations. This meant that performance early in an evolutionary run was improved at the cost of quality of the final results. The hybrid used is therefore not as suitable for the Knapsack problem as the NSGA-II, the PESA, or the ASGA. There was no distinguishable difference in the final solution quality between the H-ASGA and the ASGA on the SAD problem. This means there are no advantages or disadvantages in using this hybrid compared to the ASGA.

Figure 65 shows the H-ASGA performance against the ASGA on the EKF problem. The H-ASGA significantly outperforms the ASGA on the EKF problem. Since the ASGA outperformed PESA, 1-1 PAES, NSGA-II, SPEA2, and the SSS then the H-ASGA must also have outperformed these algorithms. The EKF problem is significantly larger than the other design problems. This could be indicative that the H-ASGA performs better on larger problems. In this case the combination of the PESA's selection operator with the ASGA's crossover and mutation operators has yielded beneficial results.

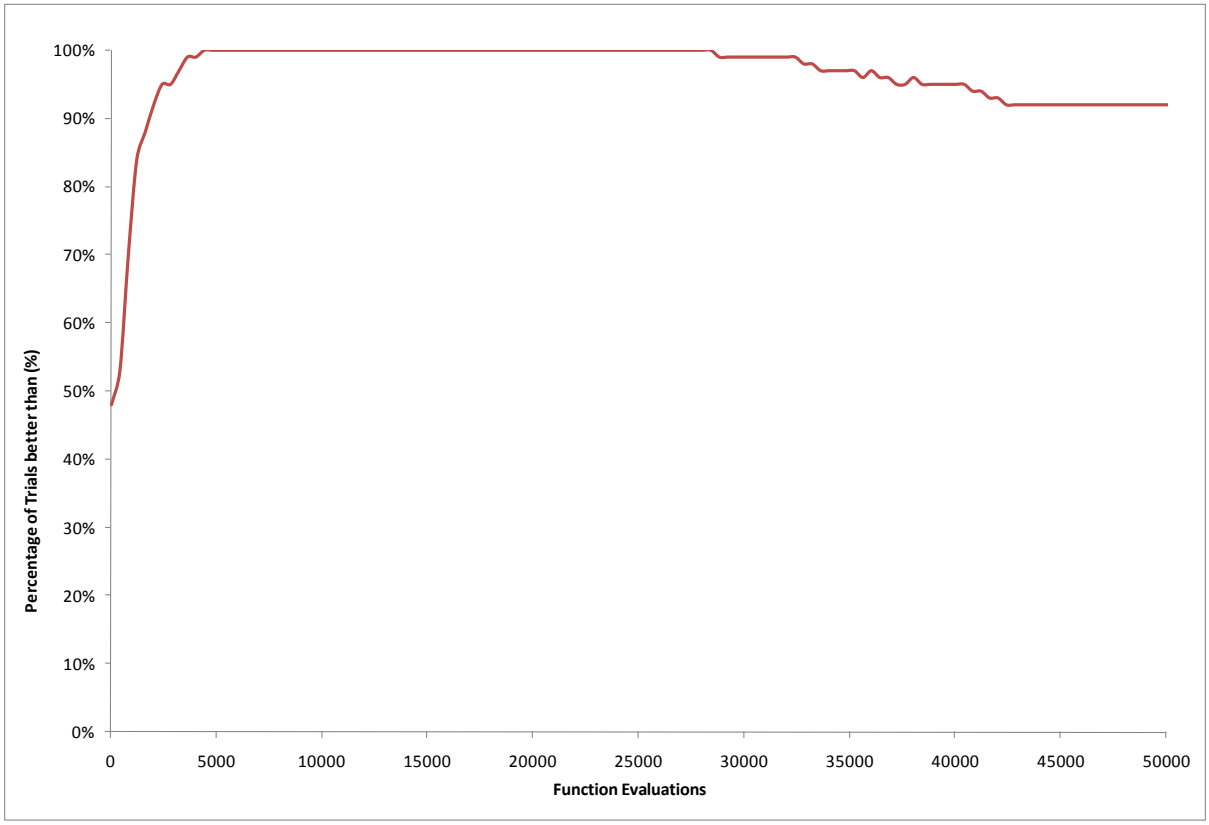

Figure 65 H-ASGA performance

Table 20 presents the solutions obtained for the EKF problem from one evolutionary run of the H-ASGA showing 96 different solutions.

Table 20 H-ASGA EKF solutions

EKF											
Logic Cells	Multipliers	Clock cycles per sample	Logic Cells	Multipliers	Clock cycles per sample	Logic Cells	Multipliers	Clock cycles per sample	Logic Cells	Multipliers	Clock cycles per sample
17582	128	89	17810	116	88	17550	130	88	18236	94	88
17454	136	88	18192	94	90	18130	96	91	18298	88	89
17484	134	105	17644	124	105	18096	98	105	18230	92	89
17842	114	88	17696	122	89	17716	122	88	17742	118	92
18034	104	89	18330	86	90	18186	94	91	18608	70	90
17970	106	89	17664	124	88	18398	82	91	17828	114	105
17516	132	105	18512	76	89	18064	100	105	17678	122	90
17778	118	88	18444	80	89	17676	122	105	17740	118	105
17548	130	105	17632	126	88	18596	72	89	18632	70	89
18002	104	90	18660	68	90	17614	126	89	17708	120	105
17612	126	105	18576	72	90	18362	84	91	17906	110	89
17748	120	88	18412	82	89	17792	116	91	18262	90	90
18066	100	90	18508	76	90	18692	66	92	18140	100	88
18034	102	90	18380	84	89	18032	102	105	18696	66	91
17486	134	88	18544	74	89	17710	120	93	17964	108	89
17874	112	89	18138	98	89	17760	118	89	18172	98	88
17938	108	90	17968	106	105	18540	74	90	18108	100	89
17580	128	105	17518	132	88	18128	96	105	18750	64	92
18328	88	88	17728	120	89	18198	94	89	18198	96	88
18296	90	88	18266	90	89	18000	104	105	18250	90	96
18366	84	90	18222	92	91	18168	96	89	18264	92	88
18440	86	88	17600	128	88	18078	102	89	18162	94	95
18098	98	90	17646	124	89	18718	66	90	18754	64	91
18154	96	90	18476	78	89	18334	86	89	18218	92	92

Comparison with a parallel and serial design implementation shows that results obtained were in the lower regions of sampling period times and embedded multipliers usage, but at the cost of logic cell usage.

7.6. Chapter Summary

This chapter has presented an extension of the Hypervolume Indicator for problems with unknown objective distributions instantiated from templates. This extension is necessary in order to enable design problems to be compared, which are instantiated from specifications and whose objective distributions are problem dependent. It uses the Hypervolume Indicator from Zitzler et al [**17**] which approximates the number of designs dominated by solutions obtained from optimisation algorithms. The extension incorporates a sign test into the design of experiments. This sign test is a distribution-free method, meaning the objective distributions need not be known. The sorting of pairs of results ensures that the properties of templates and instances of problems are allowed for.

Comparisons are made between the ASGA and the PESA [**24**], 1-1 PAES [**25**], SPEA2 [**26**], NSGA-II [**16**], and the SSS [**70**]. The first four are prominent genetic algorithms the have been developed in the literature. The last, SSS, is chosen because of the similarity of its operator with the ASGA's crossover operator. The PESA, PAES, SPEA2, and the NSGA-II make adaptations to the selection operator to enhance their performance. SSS alters the crossover operator to incorporate species. The SSS restricts crossover by species but uses a different encoding of the species tag to ASGA, has a different implementation of sharing genes amongst parents, and uses two parents. The ASGA adaptively changes the number of parents in crossover.

The ASGA is compared with the PESA, 1-1 PAES, SPEA2, NSGA-II, and the SSS on the multi-objective d-dimensional 0/1 knapsack problem [**19**]. The ASGA was found to perform better than the SPEA2 and the SSS but was found to have poorer performance than the PESA, 1-1 PAES, and the NSGA-II for lower population sizes. The ASGA required higher population sizes to be used to be effective on the knapsack problem.

When compared on the problems of the design of a parameter estimation circuit for a Self-Tuning Regulator, a SAD function, and an EKF problem results were varied. On the STR problem the PESA had superior performance but the ASGA outperformed the NSGA-II, SPEA2 and the SSS. On the SAD function the PESA and the 1-1 PAES had superior performance although the ASGA outperformed the NSGA-II, SPEA2, and the SSS. On the EKF problem, the ASGA outperformed all other algorithms tested. The results for design problems for the ASGA were more favourable with the ASGA performing better on the larger design problems.

A hybrid of the ASGA and the PESA, called the H-ASGA, was found to have poor performance on the knapsack problem, similar performance to the ASGA on the SAD function, and improved performance over the ASGA on the EKF problem. This highlights that the hybrid optimiser can lead to improvements, but results are dependent on the nature of the problem. H-ASGA may be more applicable to larger design problems requiring 400 or more design decisions while ASGA is more applicable to medium and small design problems requiring fewer than 400 design decisions.

Chapter 8. Conclusions and Future Work

This chapter presents the conclusions of this work and outlines where future work could be conducted.

8.1. Conclusions

The following summarises the design problems and how they are approached in the proposed design framework. This is followed by an overview of the design of the genetic algorithm used in the optimisation stage of the proposed design framework. Lastly, the results of comparing the design framework and genetic algorithm proposed with other leading genetic algorithms is given.

8.1.1 Design Problems

Design problems are presented as a multi-objective combinational optimisation problem. In this context they are considered as the selection of components from a library of different implementations to implement a system function specified by a designer. These components are ordered by their function, or the operation they perform, in order to enhance the reusability of different components for performing the same function. Each component is profiled so that a profile provides a number of characteristics on the physical implementation of the component. These include measures of performance and cost, but can also include power consumption.

A design framework was developed to systematically process a system specification and produce a set of pareto-optimal design solutions that implemented the system in VHDL. This was accomplished with a net list specification format involving operations and a similar component network format that could represent different implementations of a specification. The mapping from a specification to an implementation was defined, based on a specialised mapping string which contained all the necessary mapping details. By manipulating this string different implementations of the same specification could be produced.

The central process of this design framework was the optimisation of designs based on a system specification. This could be implemented by manipulating the mapping string of a specification. This mapping string was used so that the optimisation process could be considered independently of the rest of the design framework.

Optimisation is one of the key issues of this research and a limitation on the power of any automated design tool. This process was separated from the remainder of the design framework so that different implementations of optimisation algorithms could be tested to determine the best approach to use.

8.1.2 Genetic Algorithms

The genetic algorithm was selected as the basis for the design of an optimisation algorithm for the proposed design framework. This choice was made because of the robustness of the genetic algorithm and its similarities with how optimisation was being approached. The optimisation was approached as an approximation to a multi-objective optimisation problem exploiting design similarities in good solutions while testing design differences to find better solutions.

A genetic algorithm was produced, called the Adaptive Speciation Genetic Algorithm (ASGA) that has only one required search parameter, the population size. A typical genetic algorithm requires the provision of an additional crossover rate, mutation rate, number of generations to perform, and in some cases a set of weights for fitness evaluation. This reduction in the number of pre-selected search parameters makes the proposed ASGA far easier to apply than a typical genetic algorithm. More importantly, this reduction in search parameter requirements did not result in a loss of performance. The ASGA algorithm is, however, a more complex algorithm to functionally interpret, but easier to apply.

A selection operator was produced that allocates processing time in proportion to an estimate of the probability an allele is present in the pareto-optimal front. This estimate is produced based on the assumption that alleles on better individuals are more likely to produce better individuals than alleles on poor individuals. These alleles are also more likely to be on the pareto-optimal front based on the information available at the time. As this information becomes more complete, the estimate is adapted to reflect this aspect. If the allele is not on the pareto-optimal front then this estimate will reduce. If the allele is on the pareto-optimal front then this estimate will increase.

A crossover operator was developed that focuses on the refinement of individual key genetic sequences that are likely to generate better solutions. These key genetic sequences are identified by “species” tags. As tags are most commonly inherited from an individual’s parents these tags indicate an individual’s genetic history. Two individuals with the same tag

are likely to have had a common ancestor. These individuals will also be more likely to have other genetic factors in common. This commonality is considered part of an "ideal" genetic sequence. In this case each individual has a "good" genetic sequence that is a version of the "ideal" genetic sequence with some random perturbations. The crossover operator identifies sets of these individuals and uses each set to remove the random perturbations to recover the "ideal" genetic sequences. These sequences once found are the pareto-optimal solutions for the problem.

An adaptive mutation rate control system was developed using gray-codes to give good convergence characteristics without loss of solution quality. Goldberg [**15**] has proposed encoding the mutation rate into the genome of an individual before. Here, an actual implementation is used based on experimental evidence that supports the use of gray-codes as an encoding mechanism and indicates that the mutation rate should be able to vary between 0 and 100%. Experiments also found that initialising the genes that define the mutation rate, so that initially they produce a rate of 3 to 5%, gave the best performance for the tests conducted.

8.1.3 Comparison with other Algorithms

An extension of the Hypervolume Indicator for use in problems instantiated from templates was proposed. The Hypervolume Indicator is used in analysing the performance of multi-objective optimisers [**17**]. This extension accounts for the effects of template parameters in problem instances and also means that the distribution of objective values for a problem need not be known. The cost however, is that results of this extension are qualitative although they do have an associated degree of confidence. This means that it is possible to determine which algorithm is best, but not by how much.

The ASGA is compared against the Pareto Envelope Selection Algorithm [**24**], 1-1 Pareto Archived Evolution Strategy [**25**], Strength Pareto Evolutionary Algorithm 2 [**26**], Non-dominated Sorting Genetic Algorithm II [**16**], and the Simple Subpopulation Scheme [**70**] for a knapsack problem and three design case studies. The design case studies were the parameter estimation circuit of a Self-Tuning Regulator (STR), a Sum-of-Absolute-Difference (SAD) function for an 8 by 8 window size, and a 5 state Extended Kalman Filter (EKF). The ASGA performed well in all cases although the PESA was found to have similar performance.

A hybrid of the best two algorithms, the proposed ASGA and the PESA, was developed and shown to give further improvements over either algorithm on the EKF problem. Performance of the hybrid on the STR problem was worse, and performance on the SAD problem was similar to the ASGA and the PESA. The suitability of applying the hybrid should thus depend on the problem at hand.

8.2. Future Work

Future work that could be expected from this research would be in the areas of specification format, automation and extension of the pre-compiler for function acceleration and problem reduction, and extension of the user library.

The specification format used consists of a net list. Net lists are not easy to produce for large systems and verifying that a net list is an accurate representation of the intended function can be problematic. Additional work in this area to enhance the ease of use would help with the adoption of the proposed design framework in designing systems. While suitable as an internal representation for optimisation and processing, a net list is not suitable for user specifications of large systems.

An automated pre-compiler is essential for large and complex systems in order to accelerate the calculation of objective functions and perform routine specification optimisations. Pre-compiling can be used to produce an intermediate set of lookup tables that can be used in accelerating the calculation of component profiles and thereby system profiles. At this stage, any routine optimisations to specifications can also be conducted. In some cases this could include removing redundancy, resolving constant expressions, or basic reordering of operations to improve layout or performance. Such optimisations should only be those that do not have pareto-optimal effects. For instance, resolving a constant expression reduces cost without degrading performance.

At present, the user library is adequate only for testing purposes. The user library needs to be extended to incorporate more operations, components, and data types to extend the range of systems that can be optimised. Provided an operation can be represented with a Mealy Finite State Machine, the operation can be included in the user library.

Further investigation into the application of H-ASGA would determine for which problems H-ASGA is likely to outperform its competitors. This in turn could highlight where further modifications could be made.

References

[1] G. De Micheli and R. K. Gupta, "Hardware/Software Co-Design," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 349-365, Mar. 1997.

[2] R. K. Gupta, "Hardware-Software Co-design: Tools for Architecting Systems-On-A-Chip," in *Proceedings of the ASP-DAC'97*, Makuhari Messe Nippon, 1997, pp. 285-289.

[3] J. Vanne, E. Aho, T. D. Hamalainen, and K. Kuusilinna, "A High-Performance Sum of Absolute Difference Implementation for Motion Estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 7, pp. 876-883, Jul. 2006.

[4] N. Thepayasuwan and A. Doboli, "Hardware-Software Co-Design of Resource Constrainted Systems on a Chip," in *Proceedings 24th International Conference on Distributed Computing Systems Workshops*, 2004, pp. 818-823.

[5] L. Song and K. K. Parhi, "Low-Energy Software Reed-Solomon Codecs using Specialized Finite Field Datapath and Division-Free Berlekamp-Massey Algorithm," in *Proceedings of the 1999 IEEE Internation Symposium on Circuits and Systems*, vol. 1, 1999, pp. 84-89.

[6] R. Goswami, V. Srinivasan, and M. Balakrishnan, "MPEG-2 Video Data Simulator: A Case Study in Constrained HW-SW Codesign," in *Twelfth Internation Conference On VLSI Design*, 1999, pp. 128-131.

[7] F. Cloute, et al., "Hardware/Software Co-Design of an Avionics Communication Protocol Interface System: an Industrial Case Study," in *Proceedings of the Seventh Internation Workshop on Hardware/Software Codesign*, Rome, 1999, pp. 48-52.

[8] N.-E. Zergainoh, G. F. Marchioro, and A. A. Jerraya, "Hw/Sw Codesign of an ATM Network Interface card starting from a System Level Specification," in *1998 URSI International Symposium on Signals, Systems, and Electronics*, 1998, pp. 315-320.

[9] S. K. Lodha, S. Gupta, M. Balakrishnan, and S. Banerjee, "Real Time Collision Detection and Avoidance: A Case Study For Design Space Exploration in HW/SW Codesign," in *Eleventh Internation Conference on VLSI Design*, 1998, pp. 97-102.

[10] V. K. Sagar, C. Greening, W. Y. Tan, and C. S. A. Leung, "Hardware/Software Co-Design of a Fingerprint Recognition System," in *IEE Colloquium on Partitioning in Hardware-Software Codesigns*, 1995, pp. 10/1-10/5.

[11] J. Olivares, J. Hormigo, J. Villalba, and I. Benavides, "Minimum Sum of Absolute Differences Implementation in a Single FPGA Device," in *Field Programmable Logic and Application*. Springer Berlin / Heidelberg, 2004, vol. 3203, pp. 986-990.

[12] Altera Corporation. Altera Devices. [Online]. http://www.altera.com

[13] G. Beltrame, D. Sciuto, and C. Silvano, "A Power-Efficient Methdodology for Mapping Applications on Multi-Processor System-on-Chip Architectures," in *Fourteenth International Conference on Very Large Scale Integration of System on Chip (VLSI-SoC2006)*, Nice, France, 2006, pp. 177-196.

[14] IEEE SOCC Conference Homepage. [Online]. http://www.ieee-socc.org/

[15] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. USA: Addison-Welsey Publishing Company, Inc., 1989.

[16] J. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182-197, Apr. 2002.

[17] E. Zitzler and L. Thiele, "Multiobjective Optimization Using Evolutionary Algorithms -

A Comparative Case Study," in *Parallel Problem Solving from Nature - PPSN V*, vol. 1498, 1998, pp. 292-301.

[18] C. M. Fonseca, L. Paquete, and M. Lopez-Ibanez, "An Improved Dimension-Sweep Algorithm for the Hypervolume Indicator," in *IEEE Congress on Evolutionary Computation*, Vancouver, 2006, pp. 1157-1163.

[19] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*. Berlin: Springer, 2004.

[20] J. Cao, Z. Salcic, and S. K. Nguang, "A Floating-point All Hardware Self-Tuning Regulator for Second Order Systems," in *Proceedings of IEEE Region 10 Techinical Conference On Computers, Communications, Control and Power Engineering*, vol. 3, 2002, pp. 1733-1736.

[21] N. Yu, *A New Motion Estimation Algorithm for Low Bit-rate Real-time Video and its FPGA Implementation*. Auckland, New Zealand: The University of Auckland, 2003.

[22] A. Hasan, *Sensorless Vector Control of Induction Motor Drives*. Auckland, New Zealand: University of Auckland, Feb. 1999.

[23] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: Bell Telephone Laboratories, Incorporated, 1979.

[24] D. W. Corne, J. D. Knowles, and M. J. Oates, "The Pareto Envelope-based Selection Algorithm for Multiobjective Optimization," in *Proceedings of the Parallel Problem Sovling from Nature VI Conference*, Berlin, 2000, pp. 839-848.

[25] J. Knowles and D. Corne, "The Pareto Archived Evolution Strategy: A New Baseline Alogirhtm for Pareto Multiobjective Optimisation," in *Proceedings of the 1999 Congress on Evolutionary Computation*, vol. 1, 1999, pp. 98-105.

[26] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm," Swiss Federal Institute of Technology (ETH), May 2001. [Online]. http://e-collection.ethbib.ethz.ch/ecol-pool/incoll/incoll_324.pdf

[27] A. Khare, N. Savoiu, A. Halambi, and P. Grun, "V-SAT: A Visual Specification and Analysis Tool for System-On-Chip Exploration," in *Proceedings 25th EUROMICRO Conference*, vol. 1, 1999, pp. 196-203.

[28] A. Bhattacharya, A. Konar, S. Das, C. Grosan, and A. Abraham, "Hardware Software Partitioning Problem in Embedded System Design Using Particle Swarm Optimization Algorithm," in *International Conference on Complex, Intelligent and Software Intensive Systems*, 2008, pp. 171-176.

[29] W. Wolf, "A Decade of Hardware/Software Codesign," *Computer*, vol. 36, no. 4, pp. 38-43, Apr. 2003.

[30] S. Parkash and A. C. Parker, "A Design Method for Optimal Synthesis of Application-Specific Heterogeneous Multiprocessor Systems," in *Proceedings of the Workshop on Heterogeneous Processing*, 1992, pp. 75-80.

[31] A. A. Khan, C. L. Mccreary, and M. S. Jones, "A Comparison of Multiprocessor Scheduling Heuristics," in *International Conference on Parallel Processing*, vol. 2, 1994, pp. 243-250.

[32] J. F. Whidborne, D.-W. Gu, and I. Postlethwaite, "Simulated annealing for multiobjective control system design," *IEE Proceedings Control Theory and Applications*, vol. 144, no. 6, pp. 582-588, Nov. 1997.

[33] R. P. Dick and N. K. Jha, "MOGAC: A Multiobjective Genetic Algorithm for the Co-Synthesis of Hardware-Software Embedded Systems," in *IEEE/ACM International Conference on Computer-Aided Design*, 1997, pp. 522-529.

[34] H. Dhand, N. Goel, M. C. Aggarwal, and K. Paul, "Partial and Dynamic Reconfiguration in Xilinx FPGAs - A Quantitative Study," in *Proceedings of VLSI Design and Test Symposium (VDAT 2005)*, Banglore, India, 2005.

[35] G. Berry and G. Gonthier. (1988) INRIA. [Online]. http://hal.inria.fr

[36] Open SystemC Initiative. [Online]. http://www.systemc.org

[37] VHDL Analysis and Standardization Group. [Online]. http://www.eda.org/vhdl-200x/

[38] G. T. Leavens. (2008, Jan.) School of Electrical Engineering and Computer Science. [Online]. http://www.eecs.ucf.edu/~leavens/JML/

[39] Synopsys. Synopsys. [Online]. http://www.synopsys.com/products/sls/system_studio/system_studio.html

[40] C. Cote and Z. Zilic, "Automated SystemC to VHDL translation in hardware/software codesign," in *9th International Conference on Electronics, Circuits and Systems*, vol. 2, 2002, pp. 717-720.

[41] J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*, D. Gries, Ed. New York, USA: Springer-Verlag New York Inc, 1993.

[42] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541-580, Apr. 1989.

[43] F. Balarin, et al., "Metropolis: An Integrated Electronic System Design Environment," *Computer*, vol. 36, no. 4, pp. 45-52, Apr. 2003.

[44] M. Chiodo, P. Giusto, A. H. H. C. Jurecska, Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-Software Codesign of Embedded Systems," *IEEE Micro*, vol. 14, no. 4, pp. 26-36, Aug. 1994.

[45] S. Singh, "A Demonstration of Co-Design and Co-Verification in a Synchronous Language," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibtion (DATE'04)*, vol. 2, 2004, pp. 1394-1395.

[46] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-Order Reduction in Symbolic State-Space Exploration," *Formal Methods in System Design*, vol. 18, no. 2, pp. 97-116, Mar. 2001.

[47] E. S. H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 133-120, Feb. 1994.

[48] R. Lysecky and F. Vahid, "A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning," in *Design, Automation and Test in Europe*, vol. 1, 2005, pp. 18-23.

[49] F. Sun, X. Li, Q. Wang, and C. Tang, "FPGA-based Embedded System Design," in *Asia Pacific Conference on Circuits and Systems*, 2008, pp. 733-736.

[50] E. Kreyszig, *Advanced Engineering Mathematics*, Seventh Edition ed.. Singapore: John Wiley & Sons, Inc., 1993.

[51] J. Sprave, "A Unified Model of Non-Panmictic Population Structures in Evolutionary Algorithms," in *Proceedings of the 1999 Congress on Evolutionary Computation*, vol. 2, 1999, pp. 1384-1391.

[52] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME - Journal of Basic Engineering*, vol. 82, no. D, pp. 35-45, 1960.

[53] W. Fornaciari, P. Gubian, D. Scuito, and C. Silvano, "Power Estimation of Embedded Systems: A Hardware/Software Codesign Approach," *IEEE Transactions on Very Large*

Scale integration (VLSI) Systems, vol. 6, no. 2, pp. 266-275, Jun. 1998.

[54] D. Ragan, P. Sandborn, and P. Stoaks, "A Detailed Cost Model for Concurrent Use With Hardware/Software Co-Design," in *39th Design Automation Conference (DAC'02)*, New Orleans, 2002, pp. 269-274.

[55] Wikipedia. (2008, Jul.) Wikipedia. [Online]. http://en.wikipedia.org/wiki/Extended_Kalman_filter

[56] M. I. Ribeiro. (2004, Feb.) [Online]. http://users.isr.ist.utl.pt/~mir/pub/kalman.pdf

[57] C. M. Fonseca and P. J. Fleming, "An Overview of Evolutionary Algorithms in Multiobjective Optimization," *Evolutionary Computation*, vol. 3, no. 1, pp. 1-16, 1995.

[58] G. liu, J. Zhang, R. Gao, and Y. Sun, "A Coarse-Grained Genetic Algorithm for the Optimal Design of the Flexible Multi-body Model Vehicle Suspensions Based on Skeletons Implementing," in *First International Workshop on Intelligent Networks and Intelligent Systems*, 2008, pp. 139-142.

[59] Z. Man, T. Wei, L. Xiang, and K. Lishan, "Research on Multi-project Scheduling Problem Based on Hybrid Genetic Algorithm," in *International Conference on Computer Science and Software Engineering*, vol. 1, 2008, pp. 390-394.

[60] B. Mahdad, T. Bouktir, and K. Srairi, "Fuzzy Controlled Genetic Algorithm For Enviromental/Economic Dispatch with Shunt FACTS Devices," in *T&D, IEEE/PES Tranmission and Distribution Conference and Exposition*, 2008, pp. 1-8.

[61] S. Garg, P. Konugurthi, and R. Buyya, "A Linear Programming Driven Genetic Algorithm for Meta-Scheduling on Utility Grids," in *16th International Conference on Advanced Computing and Communications*, 2008, pp. 19-26.

[62] X.-Q. Zhu, Y. Gui, and X.-H. Gao, "A Novel Multi-subpopulation Quantum Genetic Algorithm," in *International Conference on Machine Learning and Cybernetics*, vol. 6, 2008, pp. 3530-3534.

[63] Y. Jewajinda and P. Chongstitvatana, "FPGA Implementation of a Cellular Compact Genetic Algorithm," in *NASA/ESA Conference on Adaptive Hardware and Systems*, 2008, pp. 385-390.

[64] C. Zhu, X.-m. Liang, and D.-h. Yan, "The Mechanism Research of a Novel Genetic Algorithm Based Species Selection," in *International Conference on Computer Science and Software Engineering*, vol. 6, 2008, pp. 462-466.

[65] B. Li, T.-S. Lin, L. Liao, and C. Fan, "Genetic Algorithm Based on Multipopulation Competitive Coevolution," in *IEEE Congress on Evolutionary Computation*, 2008, pp. 225-228.

[66] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed.. New York, USA: Springer-Verlag, 1996.

[67] E. Zitzler, K. Deb, and L. Thiele. (2000) [Online]. http://www.tik.ee.ethz.ch/sop/publicationListFiles/zdt2000a.pdf

[68] C.-F. Huang. [Online]. http://citeseer.ist.psu.edu/612029.html

[69] J. H. Holland. [Online]. http://lia.deis.unibo.it/Courses/AI/fundamentalsAI2005-06/lucidi/seminari/roli/holland.GAIntro.pdf

[70] W. M. Spears, "Simple Subpopulation Schemes," in *Proceedings of Conference on Evolutionary Programming*, 1994, pp. 296-307.

[71] E. Mezura-Montes and C. A. C. Coello, "Multiobjective-Based Concepts to Handle Constraints in Evolutionary Algorithms," in *Proceedings of the Fourth Mexican International Conference on Computer Science*, 2003, pp. 192-199.

[72] A. H. Aguirre, S. B. Rionda, and G. Lizarraga, "ISPAES: Evolutionary Multi-Objective Optimization with Constraint Handling," in *Proceedings of the Fourth Mexican International Conference on Computer Science*, 2003, pp. 338-345.

[73] H.-L. Liu and Y.-P. Wang, "Solving Constrained Optimization Problem by a specific-design Multiobjective Genetic Algorithm," in *Proceedings Fifth International Conference on Computational Intelligence and Multimdeia Applications*, 2003, pp. 200-205.

[74] V. Bhuvanshwaran and R. Langari, "Design Optimization using Genetic Algorithms and Fuzzy Constraints and Fitness Functions," in *The Twelfth IEEE International Conference on Fuzzy Systems*, vol. 1, 2003, pp. 354-359.

[75] D. A. G. Vieira, R. L. S. Adriano, J. A. Vasconcelos, and L. Krahenbuhl, "Treating Constraints as Objectives In Multiobjective Optimization Problems Using Niched Pareto Genetic Algorithm," *IEEE Transactions on Magnetics*, vol. 40, no. 2, pp. 1188-1191, Mar. 2004.

[76] R. P. Wiegand, *An Analysis of Cooperative Coevolutionary Algorithms*. Fairfax, Virginia: George Mason Univeristy, 2003.

[77] E. Zitzler, M. Laumanns, L. Thiele, C. M. Fonseca, and V. G. d. Fonseca, "Why Quality Assessment of Multiobjective Optimizers Is Difficult," in *Proceedings of the Genetic and Evolutionary Computation Conference*, San Francisco, 2002, pp. 666-674.

[78] T. Okabe, Y. Jin, and B. Sendhoff, "A Critical Survey of Performance Indices for Multi-Obective Optimisation," in *The 2003 Congress on Evolutionary Computation*, vol. 2, 2003, pp. 878-885.

[79] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonxeca, and V. G. d. Fonseca, "Performance Assessment of Multiobjective Optimizers: An Analysis and Review," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 2, pp. 117-132, Apr. 2003.

[80] J. P. Castellano, D. Sachez, O. Cazorla, and A. Suarez, "Pipelining-Based Tradeoffs for Hardware/Sofware Codesign of Multimedia Systems," in *Proceedings 8th Euromicro Workshop on Parallel and Distributed Processing*, 2000, pp. 383-390.

[81] E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257-271, Nov. 1999.

[82] J. Cao, Z. Salcic, and S. K. Nguang, "A Generic Single-Chip Second-Order System Digital Self Tuning Regulator," in *The IEEE sponsored Ninth Mediterranean Conference on Control and Automation*, Dubrovnik, 2001.

Appendices

Appendix A: Genome Encoding for Design Problems

The design framework in chapter 3 allows a specification for a design problem to be created by connecting operations from a user library. Each operation has a function that it performs. The optimization process maps a specification to a design solution that implements the system described in the specification. This process determines how many components and what type of components to allocate. Operations are then scheduled onto the available components. This completes the optimization process and yields a description from which a set of VHDL files can be generated to implement the design solution. Typically more than one solution is provided.

The design framework used requires a genome to specify how many FPGA resources to allocate for operations, which implementations of those resources are to be selected, and how many operations to assign to each resource. This is addressed by using part of the genome to form subgroups of operations that perform the same function. Each subgroup defines a resource to allocate. Each operation in a subgroup shares that resource. The number of subgroups determines how many resources to allocate. The remainder of the genome selects which implementation to use for each subgroup.

Genome Design Requirements

Each operation can have multiple components or implementations that perform that operation. These components are referred to by their architecture name. An example name could be “fast” for a high performance component or “cheap” for a low cost component, although more descriptive names are recommended. An operation can have many different components that perform its operation or relatively few. There is no restriction that each operation has the same number of components that perform the operation. Each component will have a different performance and cost profile. During optimization the combined effects of these performance and cost profiles determines the system profile. The system profile defines the characteristics of the design solution. This might be a fast and costly design solution, a cheap but slower design solution, or designs that fall between these two extremes.

Each operation must be scheduled onto a component that implements that operation. While every operation must be assigned a component, each component may have more than one

operation assigned to it. When multiple operations are assigned to the same component they are said to share that component. Sharing is a mechanism to reduce the cost of a design solution but at the expense of performance. Not all components may be shared. The ability of a component to be shared is determined by how that component is implemented on an FPGA. A component with internal memory or a state register must contain a separate copy for each operation that shares the component. If this is not the case, the component cannot be shared. The user library has a record of which components may be shared and the conditions under which they can be shared.

Operations that perform different functions cannot share the same component in the proposed framework. This is because each component implements only one function. Operations that perform the same function can be shared on components that perform that function without restriction. The selected components for performing an operation do not need to be the same. This means that for instance a “fast” and “cheap” version of a multiplier may appear in the same design. In this case the “fast” version would most likely be shared by more than one operation.

Genetic Concepts

A genome is the complete set of chromosomes that describe the genotype of an individual. The genotype is the genetic makeup of an individual. A chromosome is a group of related genes. A gene is a location within a chromosome. An allele is the value or content of a gene. Figure 66 shows a diagram with these annotated.

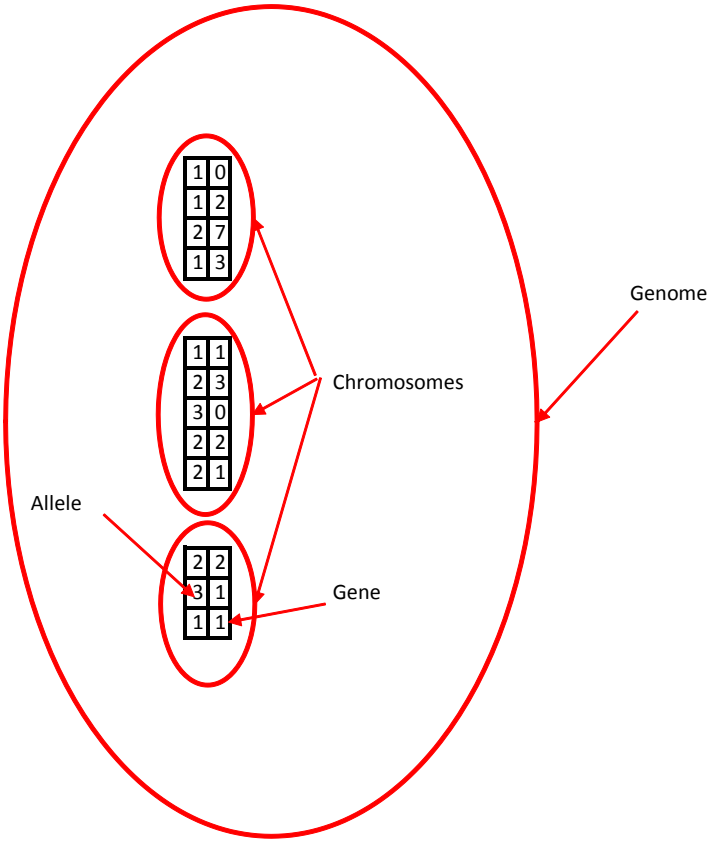

Figure 66 Genome Parts

Implementation

The ASGA crossover operator uses uniform crossover. Uniform crossover exchanges equal portions of parents to create children but does not consider gene position during this process. This means that the genome should be encoded so that gene position is not important.

Since operations that perform different functions cannot be shared in the design framework from Chapter 3 each operation is assigned a separate chromosome or group. Each chromosome thus describes all the features of a design solution related to one type of operation found in the specification. Figure 67 shows the same genome from Figure 66 but this time each chromosome has been given a name or type. In Figure 67, these are adder, multiplier, and divider which would correspond to adder, multiplier, and divider operations in the specification. All the subgroups for adders can be found on the adder chromosome for example. All the genes in the adder chromosome would concern only the selection of components and sharing of operations for adders. Similarly the other two chromosomes would concern multipliers and dividers.

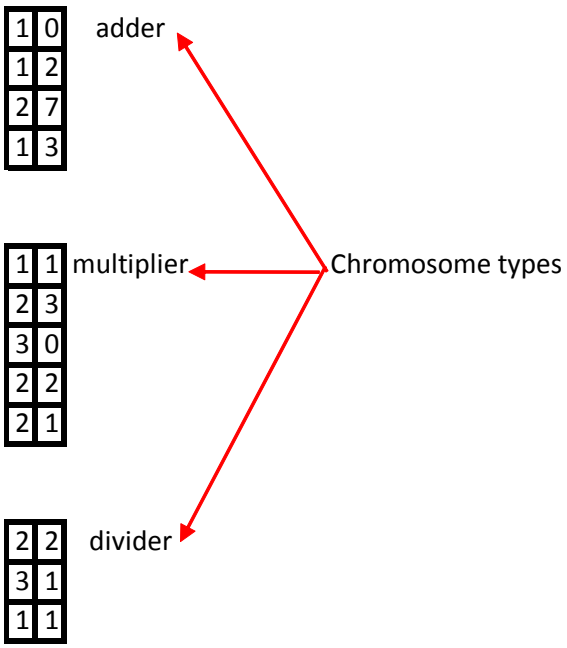

Figure 67 Chromosome Types

This reduces the problem now to the structure of a chromosome and requires only operations of the same function to be considered at this level. Potentially every operation on the same chromosome can be shared. In addition, the components that implement operations must be selected. A complication to be aware of is that due to sharing, the number of components allocated in a design solution can be less than the number of operations in the specification. Considering this the simplest process is to separate sharing and component selection to separate parts of the chromosome. The first section of a chromosome was chosen to describe how operations are shared or the division of subgroups. The second section was chosen to describe how components or implementations are selected.

Figure 68 shows the adder chromosome from the genome in Figure 67. The first section of the chromosome is highlighted in red. This section describes the sharing of operations. There is one gene in the first section for every operation belonging to the chromosome with the exception of the case when only one operation is present[3]. In Figure 68, there are four adders and each has a single gene.

[3] If only one operation is present no genes are assigned and the operation is automatically assigned its own component.

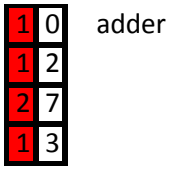

Figure 68 Chromosome: First Section

When sharing operations on a component the shared operations form a set or subgroup. The maximum number of sets is achieved when each set has one operation in it so the maximum number of sets is the same as the number of genes in the first section of the chromosome. For example, four adders can be placed in a maximum of four sets when each set contains one adder. The minimum number of sets possible is one.

A gene is a location in the chromosome. The alleles for the first section of the chromosome are set numbers. These set numbers range from one to the maximum number of sets possible for the operation the chromosome belongs to, in this example these range from 1 to 4. The set an operation belongs to is the allele held in its gene. In this example, adders 1, 2, and 4 (first, second, and last rows) belong to set 1, while adder 3 (third row) belongs to set 2. This means that adders 1, 2, and 4 will share a component as they belong to the same subgroup while adder 3 will have its own component.

The number of sets or different alleles present in a sharing arrangement is the number of components or resources that must be allocated. In Figure 68, there are two sets of adders as the alleles 1 and 2 appear. While the first section states how the operations share components it does not state which implementation of those components is used. This is the job of the second section of the chromosome. This is complicated by the fact that the number of components allocated depends on the number of sets present for sharing purposes.

The number of components allocated can vary from one to the number of genes in the first section of the chromosome. Since ASGA was designed based on genotypes of fixed lengths, and the second section of the chromosome must have sufficient genes to describe component selection in all cases, the second section must be able to accommodate the maximum number of component allocations. The second section is therefore the same length as the first section with the exception that when there is only one component choice or there is only one operation present[4]. Unlike the first section, not every gene in the second section is used as

[4] If there is only one component choice then no genes are assigned in the second section and sets are automatically assigned the only component choice available. If there is only one operation present and more than one component choice available then one gene is assigned to make the component selection.

this depends on the number of sets in the first section. Figure 69 shows the same chromosome from Figure 68 but this the second section highlighted in blue.

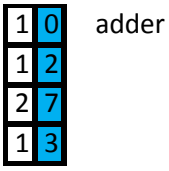

Figure 69 Chromosome: Second Section

There were two alternatives to the component selection problem. In the first option, the first set of shared components could be assigned the first gene in the second section of the chromosome, while the second, third and subsequent sets were assigned the second, third and subsequent genes in the second section of the chromosome. This system however means that the first few genes (first rows) in the second section are commonly used and the last genes (last rows) are rarely used as they only appear when a large number of sets of operations occurs. This means that the first few genes in the second section would be more important. Also, every time a gene in the first section of the chromosome changed several operations would shift the component selection gene they were based on. This is due to the changes in set memberships and relative placement. Both these factors are undesirable. The genes in the second section should not be able to change their meaning as rapidly as this may introduce instability to the genetic process; nor should gene positions be of differing importance in a genetic algorithm that uses uniform crossover.

The other option, which was the one adopted, solves this problem at the expense of introducing multiple copies of each design into the solution space. This states that the component selections in each gene of the second section belong to the set whose label matches the gene index. For example set 1's component selection is given by the first row. Set 2's component selection is given by the second row. Set 3's component selection is given by the third row and so forth. This means that the set belonging to a gene in the second section of the chromosome is fixed. Also, since each allele in the first section of the chromosome is equally likely to occur, then every gene in the second section is equally likely to be used. In Figure 69, set 1's adder component uses implementation 0, set 2's adder component uses implementation 2, set 3's adder component would have used implementation 7 if any operations belonged to set 3, and set 4's adder component would have used implementation 3 if any operations belonged to set 4.

A problem is, however, introduced in that there are many encodings that lead to the same design. Figure 70 gives one example of two chromosomes that lead to the same design. They give the same design because set 1 and 3 contain the same members with the same component selection of 0, and both set 2's use component 2. This is produced by how sets are defined and how their components are selected. To remove these identical designs would require a more complex genome or to re-enumerate alleles in a repair process. Both these procedures would make gene position important which is contrary assumptions made in the design of ASGA's crossover operator.

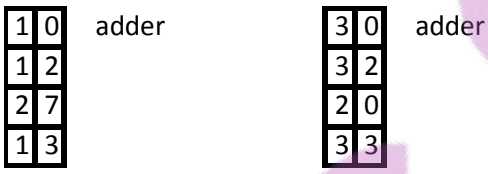

Figure 70 Identical Designs

SAD Function Example

Figure 71 gives the specification for a SAD function with a 2 by 2 window size. SAD functions are used in video motion detection application but a 2 by 2 window size is not a practical size and is only chosen to illustrate how a genome is composed by the design framework in Chapter 3 and how a genome is used to map a specification such as that in Figure 71 to an implementation in VHDL.

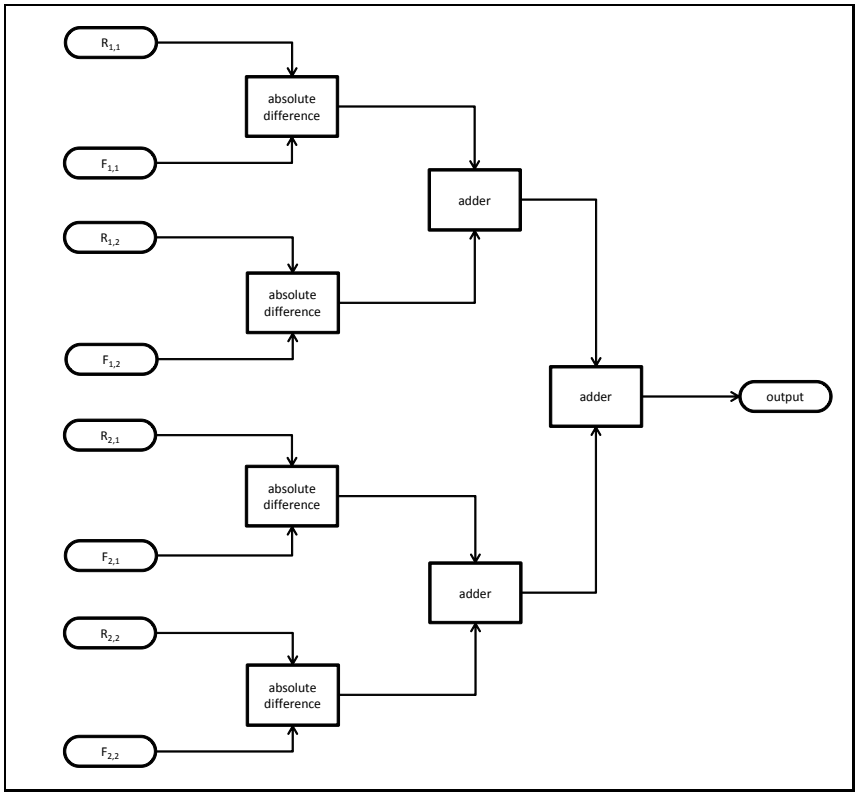

Figure 71 SAD function for a 2 by 2 window size

There are only two types of operation in the SAD specification in Figure 71. These are an absolute difference operation and an adder operation. This means the genome will have two chromosomes called absolute difference and adder. Since there are four absolute difference operations the absolute difference chromosome will have four rows. Similarly, since there are three adder operations, the adder chromosome will have three rows. Assuming each operation has more than one suitable component to implement it in the user library then the genome so far will look like that in Figure 72[5].

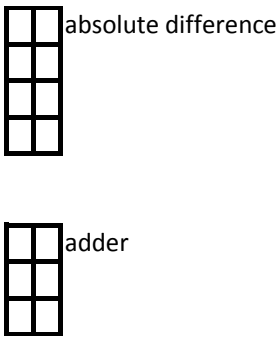

Figure 72 Genome Outline

[5] Practical design problems would usually have more than two chromosomes and more than 14 genes in their genomes. The EKF problem for example has 409 genes.

Next a mapping from specification to genome must be established. Figure 73a shows a colour coded version of the SAD specification with the genes that belong to each operation in the same colour in the genome in Figure 73b.

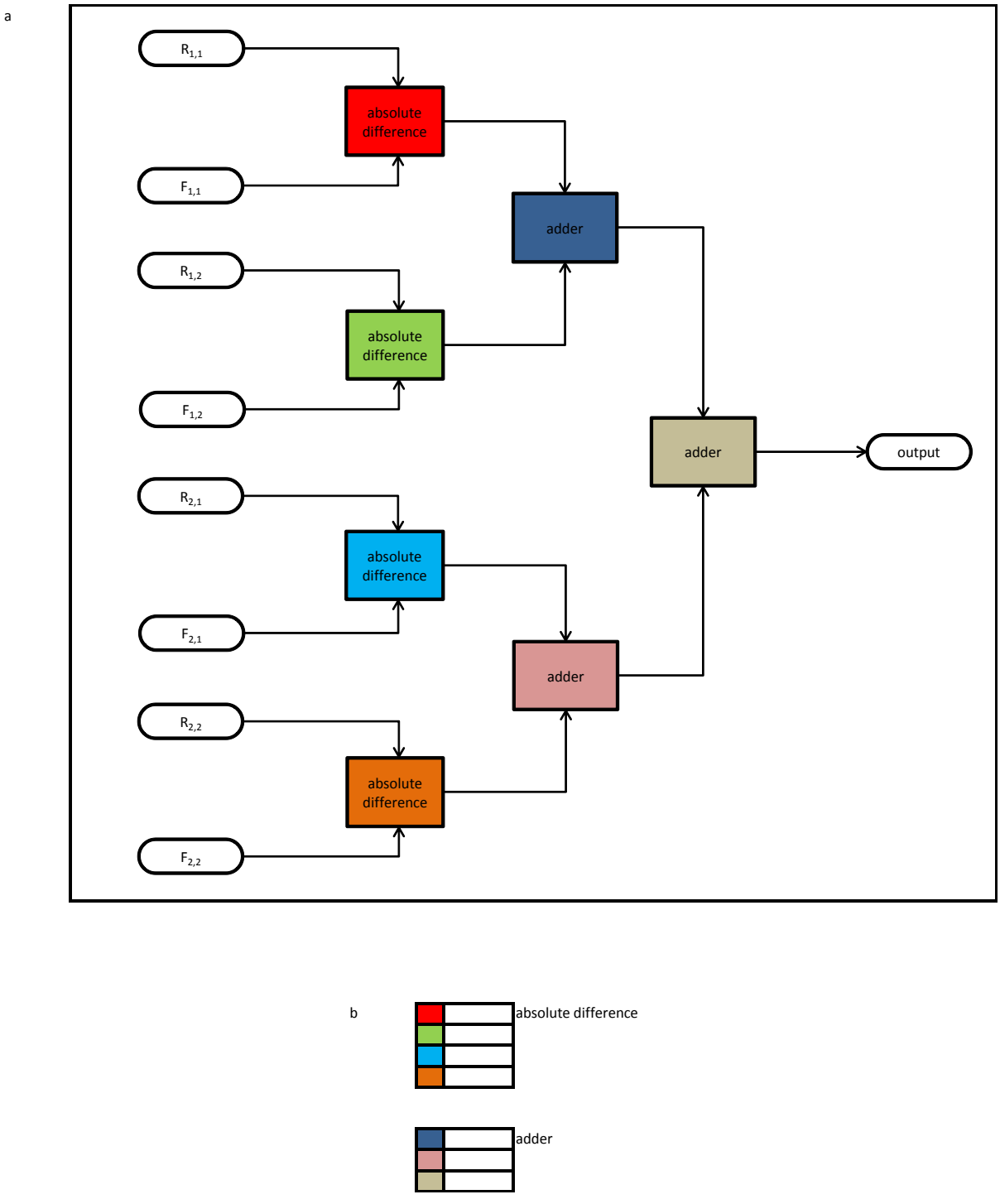

Figure 73 Specification - Genome Map

The alleles in these genes will determine which sets the operations belong to for sharing purposes. Lastly the components to implement each set must be selected. For this example it

is assumed the absolute difference function has a “fast” and “cheap” option while the adder has a “parallel” and “serial” option for component choices. These choices appear in the second section of the chromosome. Figure 74 gives a complete example of one genome for an individual that might appear during the evolutionary process.

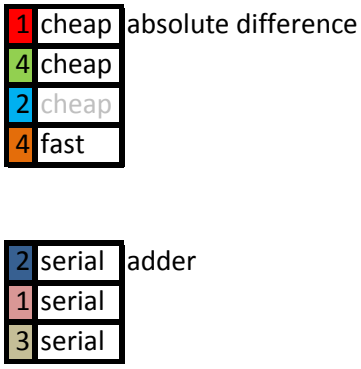

Figure 74 SAD Genome

The genome in Figure 74 would be interpreted as stating that the orange and green absolute difference operations will share a common component, and this component will be the “fast” absolute difference component as Set 4’s implementation is “fast”. The red and light blue absolute difference operations will each have their own component and these components will be the “cheap” absolute difference component. All the adders will have their own component and each adder component will be the “serial” adder component. Set 3 for the absolute difference operation has no members, so no component is allocated for Set 3. This is shown by the allele for Set 3 being “greyed out”. This indicates that while this gene will still be present in the genome its allele is not actively used for this individual. During mating, however, this allele can still be transferred like any other allele.

When implemented this individual would result in a design solution with three absolute difference components and three adder components. Two of the absolute difference components would use the “cheap” implementation while the third would use the “fast” implementation. The orange and green absolute difference operations would be performed by the “fast” absolute difference component which the design framework would add multiplexers and storage registers to. The “fast” absolute difference component would perform two absolute difference operations per sample. The red and light blue absolute difference operations would each have their own “cheap” absolute difference component and perform one absolute difference operation per sample. The three adder components would all be “serial” adder components and each would only perform one adder operation per sample.

The design framework would also add any other channel components necessary to ensure that the resulting implementation matched the function of the specification.