Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Exploring Evolutionary Hardware:

Evolved Binary Arithmetic Circuits and Dynamic Problems

Alexander Dalton

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 4th May, 2018

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Alexander Dalton, Friday 4th May, 2018

Contents

List of Figures

List of Algorithms

Executive Summary

The intersection between machine learning and hardware design is an oft-unexplored area. Evolutionary hardware is one such field which strives to automate hardware design through the application of genetic algorithms. This thesis explores how genetic algorithms can be improved in the context of binary arithmetic. The resulting evolutionary hardware systems are subjected to a series of dynamic problems, including fault injection and contextual hardware optimisation.

Evolutionary hardware systems are built on Field Programmable Gate Arrays (FPGAs), a configurable flexible hardware platform. To improve development time, and remove execution bottlenecks, a simulated FPGA will be constructed. This will act as a self-contained unit and provide the evaluation back end to the genetic algorithm.

The research hypothesis is that genetic algorithms can be used to construct efficient hardware systems suited for tackling the suite of dynamic problems hardware encounters on a regular basis.

The main achievements include:

- Building a highly specialised FPGA simulator (in C) to allow quick genetic algorithm development. FPGA size is arbitrary and subject to user specification.
- Implementing and improving on known genetic algorithms to create FPGA configurations for the binary arithmatic problem. The system is highly configurable with multi-objective training weightings, selection schemes, population size, mutation rate, elitism, coevolution (parasite size, and virulence), and crossover probability all subject to the whims of the user.
- Exploring fault recovery and dynamic optimisation with evolutionary hardware on the FPGA simulator.
- Creation of a clean user interface to allow for easy mid-execution user evaluation.
- Explore scaling bottlenecks and potential for mitigation with coevolutionary techniques.

Supporting Technologies

- I used the NCURSES C library to develop a GUI.

Notation and Acronyms

FPGA	:	Field Programable Gate Array
CLB	:	Configurable Logic Block
ASIC	:	Application Specific Integrated Circuit

Acknowledgements

Firstly, I would like to thank Prof. Seth Bullock for his invaluable advice and guidance throughout this project. Thanks is also owed to the various members of the computer science department who dissuaded me from working with a physical FPGA, and convinced me to play to my strengths and operate firmly within software. I would be amis if I didn't also extend my appreciation to my family, who's unwavering food supplies were surpassed in value only by their blind appreciation for the work I have been producing.

Chapter 1

Contextual Background

Unlike conventional hardware designs which are often meticulously handcrafted by experts, evolvable hardware applies genetic algorithms to flexible hardware platforms to automatically explore the design space of potential circuit solutions to specified hardware problems [13]. The space of all possible circuit designs is huge. Currently no deterministic search procedures perform well navigating through all possible solutions. Despite some serious shortcomings the best performance so far in automated hardware design has been through the application of genetic algorithms. Currently this approach has only been successfully used on relatively trivial problems, due to huge scaling issues. But where it has been used the application of the robust nondeterminism inherent in genetic algorithms to hardware design has correctly autonomously developed application specific hardware.

The remainder of this chapter is dedicated to outlining the motivation of the project and setting project objectives. Chapter 2 deals with the existing literature and provides the technical understanding required for the rest of this thesis. In Chapter 3, the high level design choices made during the project will be outlined, followed by a thorough description of the implementation details. Chapter 4 focusses on evaluating the project and includes all tests conducted throughout the project. Finally, Chapter 5 contains the conclusion to this dissertation.

1.1 Evolvable Hardware

Decades of evolvable hardware research has explored the application of genetic algorithms to the domain of hardware design. Genetic algorithms come from the field of natural computing and use Darwinian-inspired probabilistic procedure to improve a population of candidate solutions' performance given a fitness criteria via evolutionary pressure [7]. In the case of evolvable hardware, each individual is a bitstring which can be mapped onto a circuit design and the fitness criteria is rooted in the circuit's ability to perform some predefined function.

The most common evolvable hardware arrangements are built on Field Programmable Gate Arrays (FPGAs), these are immensely flexible integrated circuits and constitute the substrate that the genetic algorithm creates a solution within. Each member of the population represents an FPGA configuration, and the individual's fitness is based on the physical performance of the chip when configured to the individual's specification.

Conventional circuit design requires a huge amount of domain specific knowledge. Applying machine learning to hardware design constitutes a potential offloading of this information, allowing a user to define the success criteria for a circuit and letting the machine autonomously construct a novel solution. Thus far, there have been great successes deploying evolvable hardware in a few narrow domains; creating user-specific prosthetic hand controllers [19], image filters [1], arbitrary logic circuits [46], and even industrial robot controllers capable of fault recovery [14].

Current solutions are severely limited by scaling issues. As a problem gets larger, a larger FPGA is required to tackle it, this results in the search space exploding in size and crippling search times. Another factor contributing to the scaling issue is that of test-case scaling. When evolving a 1-bit binary adder circuit there are 4 test cases; $0 + 0$, $0 + 1$, $1 + 0$, and $1 + 1$. However, when you attempt 2-bit addition this becomes 16 test cases, for 4-bit addition 256 individual sums have to be trailed to evaluate a candidate solution. The vast search space and rigorous evaluation step are the main problems with scaling evolvable hardware beyond trivial problems.

One of the advantages of genetic algorithms is also one of their largest weaknesses; they excel in

generating strange and esoteric solutions. These novel solutions sometimes outperform their more conventional hand-designed counterparts, but due to the seemingly arbitrary way key design decisions are made, by a genetic algorithm, direct comparison to known solutions is difficult. This makes learning from a genetic algorithm design hard, often we know that solutions work well but know *how* it works well. Analysis of the resulting hardware produced by an evolvable hardware project rarely extend beyond an acknowledgement of functional correctness.

An example of one such strange design (although outside the domain of circuit synthesis), created with genetic algorithms, comes from NASA. Where an evolutionary algorithm designed an, ironically alien looking, antenna for use in space (Figure 2.2)[18], this antenna performed better than any hand designed counterpart.

Unfortunately many applications of genetic algorithms are not as successful as NASA's antenna. It is uncommon that, by many design metrics, genetic algorithms produce results better than the hand designed alternatives. However, conventional hand designed circuitry requires a huge amount of intuition from experienced designers as they balance knowledge about the manufacture process, fault probabilities, power consumption, among any number of other requirements. Any effort to automate this must be seriously explored.

Despite being a relatively old field, and serious developments in genetic algorithms (and machine learning in a wider sense), evolvable hardware has for stagnated in recent years. In this project, exploration into how modern genetic algorithm techniques can improve on an algorithm capable of designing solutions to conventional hardware problems, such as simple binary arithmetic, will be conducted. It is also hoped that an exploration into the scaling issue will highlight the largest contributor to this problem and potential mitigation strategies.

1.1.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays [23] are the backbone of the vast majority of evolutionary hardware setups. They are a type of integrated circuit which can be configured after manufacture to perform a variety of operations. Conventionally an FPGA design is specified in a hardware design language and then compiled into a chip-specific bitfile. This bitfile is uploaded to the device and configures the internal components to perform the defined task.

The core functionality is built from a homogeneous mesh of thousands of Configurable Logic Blocks (CLBs). Each of these blocks takes input from their neighbouring cells, has an internal binary function, and sends distinct outputs to each of the neighbouring cells. The values sent to each output can be the result of the internal function or a direct mapping from any of the inputs. The function, the input(s) to the function and the values sent to each output are completely configurable. Figure 1.1 demonstrates the internal structure of a CLB and how they can be arranged to form a simple FPGA. Modern FPGAs also have a host of more complex features, such as expansive input/output, wide data throughput, and look up tables. Configurations are stored on-chip in ROM as a bitfile. This bitfile is usually generated extrinsically [21] and loaded onto the device [23]. If properly configured (and containing an appropriate number of CLBs) an FPGA can functional identically to any printed digital circuit. They can be thought of as the polar opposite to an ASIC (Application Specific Integrated Circuit), in that rather than only performing one manufacturer-defined function, as ASICs do, their functionality can be modified at will. FPGAs share the power efficiency and execution proficiency of ASIC hardware, but are considerably more expensive at scale.

When a great number of chips are required ASIC development makes sense; however, when a user needs ASIC-like performance for only a handful of devices, it can often make more fiscal sense to develop an FPGA configuration which matches the design needs. This is because ASIC development can be a lengthy and expensive process. The limited number chips required and associated small-scale manufacturing costs often makes FPGA development the clear option. This pressure for highly specific cost effective hardware on a small scale has driven mainstream FPGA development. One FPGA chip design can service many specific hardware needs.

These immensely flexible circuits see wide use across industry. The most immediately obvious example is within firms developing more conventional integrated circuits who also ship accompanying software. In these situations a team of engineers can develop software while the hardware is developed in parallel, by using an FPGA loaded with a beta version of chip. This avoids the many month wait times for a finished chip to be fabricated. Recently FPGAs have seen a great deal of use as deep learning accelerators [49]; this is because FPGAs configurations are cheaper to develop than ASIC hardware, and see huge performance and power consumption benefits over more general purpose hardware (GPGPUs for example).

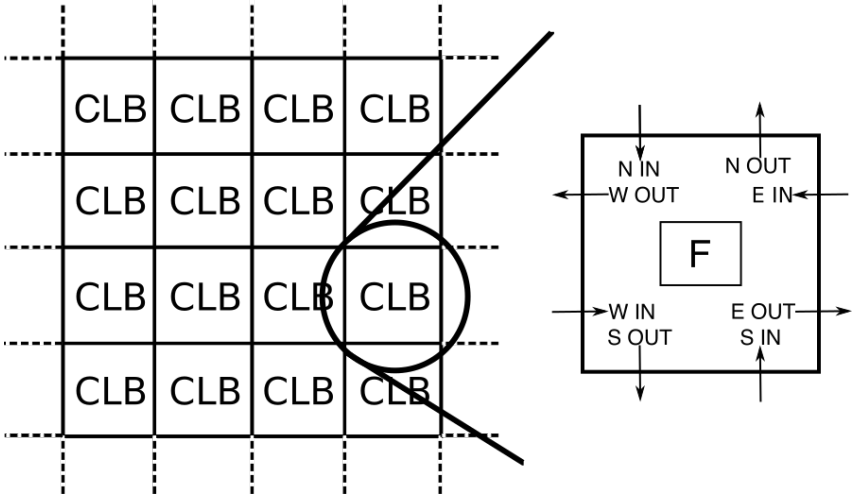


Figure 1.1: FPGA architecture

The flexibility also means iterative improvements can be made to coincide with emerging research without the need to purchase new hardware. Similarly, many industrial institutions require bespoke hardware solutions to small-scale specific problems; suppose there is a demand for a real time, high performance pump controller which will only be used in a handful of locations. In these situations, more often than not, ASIC development is too expensive and not worth the narrow application setting. This demand has resulted in the widespread use of FPGAs in roles requiring an ASIC but without the market pressure to drive ASIC development [29]. An application specific design can be built on an FPGA, while approaching peak performance and power efficiency without massive development costs. The military [30] and aerospace [12] sectors use FPGAs extensively for these reasons along with the built-in cryptographic and anti-tamper hardware obfuscation capabilities on military-grade FPGAs.

1.1.2 Genetic Algorithms

The core of any genetic algorithm takes a randomly seeded population of binary strings and applies evolutionary pressure by performing a cycle of selection, crossover (an optional step), and mutation to move the population towards potential solutions to a given problem [7]. Beyond this there are many variations, some of which will be explored in this thesis. The basic genetic algorithm is outlined in Algorithm 1.1.

Randomly initialise a population P
while *evolving* **do**
  $Evaluate(P)$
  while *New population* P' *not full* **do**
    if *Crossover occurs* **then**
      $p_1 \leftarrow Select(P)$
      $p_2 \leftarrow Select(P)$
      $i_1, i_2 \leftarrow Crossover(p_1, p_2)$
      Add i_1 and i_2 to P'
    else
      $i \leftarrow Select(P)$
      Add i to P'
    end
  end
  $P \leftarrow Mutate(P')$
end

Algorithm 1.1: Basic genetic algorithm

An initial population is randomly generated and then evaluated ($Evaluate(P)$). Evaluation provides each individual in the population with a score. Members of a new population are generated by selecting members of the old population at random, based on some selection mechanism (the most common schemes

will be outlines in Chapter 2) and the scores each individual received. The individuals selected in this manner have a single parent, some genetic algorithms also have a mechanism by which an individual can have two parents; this is called crossover. Crossover has a random chance of occurring, if it does, two individuals are selected from the old population (via $Select(P)$), then from a random point in each parent's bitstring, the parents swap genetic material. This generates two new individuals, both are added to the new population. Finally, if an individual is made up of a bitstring of length l and we have a mutation rate of m, the $Mutate(P')$ function iterates over each bit in each individual, flipping the bit with probability $\frac{m}{l}$. The new population P' then replaces the old P, and the cycle continues. These simple processes coalesce into a high performance robust search procedure. Genetic algorithms are covered in more detail in Section 2.1.

1.1.3 Genetic Algorithms with an FPGA Configuration

In a biology a genotype is an organism's DNA, and a phenotype is the organism itself, the expression of the DNA. In the context of genetic algorithms, the genotype is a binary string which is mapped onto a phenotype (in evolvable hardware, often an FPGA configuration). When designing an evolvable hardware system one must decide how to map from binary string to FPGA configuration.

Given a mapping from binary string to FPGA configuration and an FPGA test bed, one can evaluate a population of bitstrings as the FPGA configuration to solve some digital problem. This framework is at the core of evolutionary hardware. The bottleneck for physical evolvable hardware is often the evaluation step, this involves converting each individual to an FPGA configuration, uploading each in turn to the FPGA, and extensively testing it. When each upload takes in the order of multiple seconds the evaluation process can be laborious. To resolve this problem many platforms simulate an FPGA until a design is chosen to be deployed, this reduces training time aggressively, and is the direction this project has taken.

1.2 Dynamic Problems

A problem with evaluation critera which shifts over time is termed a dynamic problem. Designing evolutionary hardware robust to this set of problems is of considerable benefit as dynamic problems span a class of practical but notoriously problematic challenges including real-time optimisation, and fault tolerance.

1.2.1 Hardware Faults

Hardware faults are catastrophic for electronic devices. A relatively short life span is mostly accepted, and is relatively benign in many areas; but when the cost of replacement is extraordinarily high or the scale of the operation is large enough, there are huge benefits to improving the fault tolerance of devices. An extreme example of the high cost of replacement can be found with satellites, surveys of in-orbit satellites reveal that once deployed the reliability of satalites drops aggressively, and despite the highest manufacturing standards, after 15 years reliability drops to below 90% [5]. Be it due micrometeor impacts, or the ionising effects of radiation, satellites are known to fail and a great deal of work is expended improving their reliability. With the cost of putting a satellite into orbit set in the millions of pounds, extending the lifespan of such devices would have significant economic impact.

A little closer to home, data centres are vast structures contain thousands of servers. Each of these has an 8% probability of experiencing a failure each year [47]. Individually this is of no great concern, but when compounded across an entire data centre server recovery and replacement becomes a primary concern for the management of such an establishment.

One popular use of dynamic problems in evolutionary hardware is designed to breed fault tolerance into a design. This involves repeatedly turning on and off simulated faults in the FPGA during evaluation in order to create a design both robust to faults and not dependent on faults, as could happen if only evaluated in a faulty system [41]. Another fault resilience technique requires evaluating the configuration against a fault-free FPGA and then combining the result with evaluation runs on FPGAs simulating known frequent faults [41][22]. These faults could include component wear out, and manifest as blocked communications between CLBs or render the function performed by a CLB inert. This extension of the fitness function (rather than mid-execution fitness function modification) removes this approach from the domain of dynamic problems, but it is worth noting as an alternate method to improve circuit reliability.

All of these methods require faults to be set before the genetic process begins, and can only be used to develop evolvable hardware tolerant to specific faults. This requires a huge amount of knowledge about

the underlying hardware implementation and the frequency and severity of faults to generate accurate fault models. This is an important avenue of exploration but with evolvable hardware there is a missed opportunity, with a system capable of quick iterative improvements, to work around a problem as it occurs. This idea has been briefly explored in [14].

1.2.2 Dynamic optimisation

Another successful area of dynamic problems with evolutionary hardware involves extending the evaluation function when a perfect solution has been found [20]. For example, one could successfully evolve an audio filter and then incorporate a measure of "smallness" into the fitness. This would add evolutionary pressure to not only be correct, but also use as little of the FPGA resources as possible.

Little work has been done exploring the reaction of evolvable hardware to tackling related-but-not-identical problems (addition and subtraction, for example), and observing the effect of varying the relative benefits for correct answers for either, on the performance of the circuit for each problem. Information in this domain could drive development of systems capable of dynamically optimising in real-time under shifting conditions.

1.3 Project Aims

The broad aim of this project is to develop an improved evolvable hardware platform capable of effectively addressing dynamic problems. More specifically:

- Apply the genetic algorithm from [42] to the binary arithmetic problem.
- Combine state of the art genetic algorithms to improve evolvable hardware performance for binary arithmetic.
- Explore the application of evolvable hardware to dynamic problems, including FPGA faults and weighted binary arithmetic.
- Develop a specialised FPGA simulator to act as the evaluation back end for the genetic algorithm.
- Study and improve the scaling performance of evolvable hardware.

1.4 Project Challenges

The project is not without challenges:

- There are few ways to evaluate individuals and population health beyond how correct they are, so understanding why a system works or does not work may be difficult.
- The issue of scaling will slow development of anything other than trivial problems (2-bit addition and subtraction).
- FPGA configurations for a variety of problems investigated here are very fragile, in the context of evolution. Frequent mutations could be disastrous.
- More so than in many evolutionary contexts the prospect of evolutionary dead ends and dominating local optima will need to be addressed.

Chapter 2

Technical Background

This chapter provides the technical background to the project and discusses the relevant existing work.

2.1 Genetic Algorithms

Genetic algorithms grew from a branch of engineering which takes influence from nature. By applying Darwinian evolutionary theory to a population of binary strings, researchers developed a robust directed nondeterministic procedure to navigate search spaces with noncontinuous fitness. These search spaces are not well suited to conventional search methods, such as enumerated bruteforce or heuristic driven methods like A* search, and previously were only navigable by simple hill-climbers, random walks, or bruteforce search. Genetic algorithms provide a good general method to approach problems where little information is known about a space of solutions. Goldberg's book *Genetic Algorithms: In Search Optimisation & Machine Learning* [7] contains a good summary of the knowledge of genetic algorithms as it stood in the early 1990s. The genetic algorithm as presented by Goldberg consists of a repeated cycle of reproduction, crossover, and mutation; and operates on a population of randomly seeded binary strings.

Reproduction is the mechanism by which a new population is generated. Given a fitness function f, providing a measure of the quality of an individual, each member of the population is evaluated and assigned a score. To improve the fitness across the entire population, some form of Darwinian selection is used such that the probability of an individual reproducing is higher for individuals with better fitness. One such commonly chosen selection mechanism is roulette wheel (stochastic) selection. For each individual x, in a population of size n, with fitness function f, the probability of x producing offspring with roulette wheel selection is given by equation 2.1. After the reproduction stage a new population has been created, sampled from the old.

$$P(x) = \frac{f(x)}{\sum_{i=0}^{n} f(x_i)} \tag{2.1}$$

Crossover is an optional part of the core genetic algorithm definition provided by Goldberg [7]. It allows two strings to act as parents and share genetic material for an individual in the new generation. Given a crossover probability, each slot in the new population has a chance to be filled by crossover. If crossover is selected as the means to fill a position in the new population, two parents are selected by the genetic algorithm selection mechanism, a number k, is selected such that $1 \leq k \leq l$, where l is the population string length. Both individuals then swap every bit from position k onwards. This generates two new individuals, both are added to the new population.

Mutation occurs after reproduction and crossover. Given a population of strings, each of length l, and the number of mutations expected per individual m, each bit of each string is flipped with probability $\frac{m}{l}$.

These relatively simple mechanisms provide the non-deterministic but focussed, and surprisingly robust search procedure capable of addressing a problem when little is known about the search space and systematically forming an answer is unfeasible.

An interesting tangential application of FPGA technology to evolutionary algorithms comes as a physical evolution hardware accelerator [34], which uses the flexibility from the FPGA to allow incremental

runtime hardware changes, which allow the hardware realisation of functions which would otherwise be impossible to construct directly, due to the need for fluid function reconfigurability. This highlights the opportunity for aggressively efficient evolvable hardware systems.

2.1.1 Selection Mechanisms

Goldberg et al. offer a comparison of common selection schemes for use in genetic algorithm reproduction [8]. They compared proportionate selection, rank based selection, and tournament selection, amongst others.

Proportionate selection is any selection scheme in which the probability of an individual being selected as a parent is proportional to their fitness score. This includes roulette wheel selection.

Rank selection orders each member of a population by their fitness, then the probability of each individual being selected as parent is proportional to their position in the ranking. This serves as a way to "flatten out" the selection curve, and reduces the influence of a huge spread across individual fitness. Commonly a linear ranking is chosen, in which the most fit individual is twice as likely to be selected as a parent as the individual with median fitness.

Tournament selection is a type of proportionate selection, which operates by randomly selecting a subset of the population and the best individual from this set is chosen for further genetic processing. Tournaments can be as small as consisting of 2 individuals, or much larger.

There are key differences between these selection mechanisms which influence the decision to incorporate them into an evolvable hardware platform. Proportionate selection discriminates heavily by the fitness of individuals. Rank based discriminates less and cares more about who performs better (rather than by how much they perform better). Tournament selection discriminates intensely within a given tournament, but the selection process to assemble the tournament is uniformly random, so depending on the tournament size this can be highly discriminatory (large tournament size) or minimally discriminatory (small tournament size). At certain tournament sizes, tournament selection performs similarly to roulette wheel selection. For evolvable hardware, proportionate selection sees little use due to the range of fitness values often associated with members of the population, rank and tournament selection are frequently used.

2.1.2 The Fitness Function

$$f(x) = \sum_{i=1}^{z} w_i f_i(x) \tag{2.2}$$

By extending the fitness function to a weighted linear combination of a function measuring accuracy (the original fitness function), and any other measurements, we can use evolutionary pressure to optimise for additional parameters [6]. For an individual x, a set of functions measuring distinct desirable aspects of individual $\{f_1, f_2, \ldots, f_z\}$, and a set of weightings associated with each function $\{w_1, w_2, \ldots, w_z\}$ the new fitness function $f(x)$, is given by Equation 2.2.

This technique has been used to expand the success criteria for a evolvable hardware configuration to create smaller, or more fault resistant hardware designs [43][41][22].

2.1.3 Coevolution

Coevolution refers to the practice of evolving two populations in tandem, this can act as a way to divide labour (two populations solving different parts of a problem) [31], or as adversaries (one population of problem proposers and another of problem solvers). The former is used to improve the scalability of evolvable hardware to naturally decompose the problem into subproblems. The latter is often likened to a host/parasite or prey/predator relationship. The framework for a coevolutionary system is only a slight extension to the generic genetic algorithm outlined previously. In a competitive system, P_s is a randomly seeded population of problem solvers, and P_p is a randomly seeded population of problem proposers; where each individual is represented by a bitstring of appropriate length. The fitness function of one population is inexorably tied to the fitness function of the other. When evaluating, each problem solver

$p_s \in P_s$ is randomly associated a problem proposer $p_p \in P_p$, and is scored based on how many problems proposed by p_p that are correctly solved. p_p is scored based on how many problems are incorrectly solved. Once each individual is scored the genetic process continues as expected, with independent reproduction, crossover, and mutation for each population.

The evolutionary pressure on the problem proposers pushes them to be as difficult to solve as possible, therefore emphasising problems the population of solvers find difficult. This proves a highly effective distinguishing tool, and produces problem proposers which create hard problems specifically tailored to the population of problem solvers [16].

Extending the parallel with nature, one can modify the virulence of the parasitic coevolutionary population. Virulence is a measure of the hostility of the aggressor. Malaria is a highly virulent virus, often killing it's host; the evolutionary process which drove it's development encourages causing maximum harm to the target. An example a of virus with a low virulence is the common cold, it gains nothing from killing the host, it wants the host to survive and spread the virus; this means there is evolutionary pressure discouraging maximal virulence.

In some settings a population of highly virulent problem proposers develop to be much more aggressive than the solvers can handle, meaning no solver ever manages to successfully solve any problem, even if it could have solved the simpler problems. When individuals can no longer differentiate themselves and push the population up the evolutionary ladder, the coevolved populations are said to have disengaged. By modifying the fitness function Cartlidge et al. [4] discouraged maximally virulent parasites. This encourages populations to stay engaged, and improves the quality of discovered solutions.

2.2 Evolvable Hardware

2.2.1 Theory

The foundational work in evolvable hardware is summarised by Higuchi et al. in their paper *Evolvable Hardware with Genetic Learning* [13]. They describe the flexibility of FPGA devices and how this can be exploited by genetic algorithms by treating the bitstring used by the hardware for calibration as the genetic material to be evolved. They also describe the fitness function as the total number of correct output bits read off the hardware for all possible inputs. They succeed in evolving multiplexors and counters, but highlight a limitation of the direct evolution of the FPGA calibration bitstring; only a subset of the bitstring include bits relevant to the function of a specific region of the hardware, some bits were redundant, act as checksums, or perform routine non-optional calibration. This increase in the amount of genetic material expands the search space and extends the time required for the genetic algorithm to find a solution. Using the proposed framework, they evolve an image pattern recognition system, and a welding robot controller which takes sensor information and traces a ditch.

In their work they suggest operating on abstract logic gates, rather than the FPGA configuration bitfile directly. Figure 2.1 outlines this principal. A mapping is defined which translates a binary string into a collection of logic gates. This logic structure can then be synthesised into an FPGA configuration bitstring and is uploaded to the device. They, also highlight the need to abstract from gate level evolution to improve the execution times of the genetic algorithm and present function level evolution as a solution. This involves replacing gates (AND, OR, NOT, etc.) with functions (adder, subtracter, etc.) as the atomic evolutionary components.

The capacity for evolutionary hardware to come up with bespoke and novel solutions was explored by Thompson [42]. A genetic algorithm operated on a 10x10 grid of cells in the top corner of an FPGA, the aim was to evolve a circuit capable of differentiating a high frequency input signal (10kHz) and a low frequency input signal (1kHz). The output signal should read "high" (+5V) for one frequency and "low" (0v) for the other. This was a truly bespoke application of evolutionary hardware as the region exposed to the genetic algorithm had no access to any hardware capable of timing; the differentiation task was one which the hardware conventionally would not be able to do.

The algorithm driving evolution was standard in many respects; with a population size of 50, the most fit individual was copied verbatim into the next generation (a mechanism called *elitism*), and linear rank-based selection (where the fittest individual had a probability of selection double that of the median individual) was used for the remaining members of the population. The probability of (single-point) crossover occurring was 0.7 and the expected number of mutations per individual (except the individual copied over via elitism) was 2.7. These values are in accordance with existing research and were arrived at after domain specific experimentation.

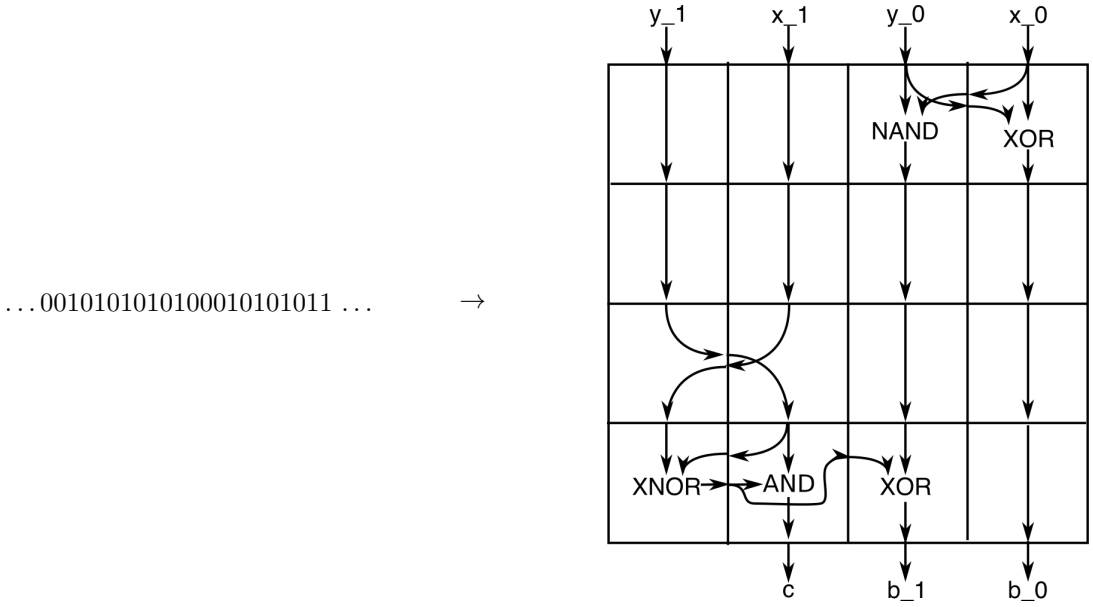


Figure 2.1: For gate-level evolution a mapping has to be devised from bitstring to logic structure.

The 100 cell area was encoded as a string of 1800 bits. Each cell was defined left-to-right row-by-row. The fitness evaluation was conducted on physical hardware (many evolvable hardware schemes use simulations to improve training time), a series of test frequencies were fed into the device and a single cell's output was read. The fitness of a configuration was defined as the difference between the average output for each of the two input frequencies (multiplied by a constant to avoid otherwise inescapable local optimum), therefore the larger the distinction between inputs, the more fit the configuration was deemed. Without the constants in the fitness function the genetic algorithm directly connected the input and output of the circuit through the FPGA, creating a useless configuration which performed well enough to discourage any further exploration.

After 3500 generations of evolution, the genetic algorithm produced a specification capable of distinguishing the two input signals cleanly. This is behaviour beyond what the hardware was designed for and demonstrates the power of evolvable hardware. One of the interesting things to come from Thompson's work was the demonstration that genetic algorithms happily exploit undefined and strange behaviour on-chip, such as feedback loops and more strangely: unconnected circuitry. The successful individual's schematic was pruned of any circuitry which should not influence the output (a direct path could not be traced from the input to the output via that route), and the fitness of what remained *dropped*. This reduction in quality was attributed to strange undefined chip behaviour, unwittingly exploited by the genetic algorithm. FPGA cells exerted subtle influence over neighbouring cells despite the absence of any explicit connection. The search procedure took 2-3 weeks due to each evaluation taking up to 5 seconds.

By extending the fitness function we can apply evolutionary pressure to configurations in order to nurture more than just accuracy. One of the first uses of multi-objective fitness function in evolvable hardware was to reduce the size of the circuit [20]. Once a correct solution has been evolved the fitness function shifts to a linear combination of correct output bits and the number of cells inactive. This paper also introduced a mechanism for evolving the size and shape of the underlying fabric in parallel to conventional evolvable hardware evolution, this allows for a system to define how large it should be. Both multi-objective fitness functions and evolved circuit structure apply evolutionary pressure to improve the quality of evolutionary hardware beyond simple performance accuracy.

An alternative substrate for evolvable hardware to FPGAs comes in the form of field programmable transistor arrays (FPTAs) [39]. These offer similar functionality to FPGAs, but where FPGAs allow gate-level flexibility, FPTAs allow a user to specify the placement and configuration of transistors. Evolutionary hardware operating on this level is subject to worse scaling issues than gate-level evolvable hardware, but has a great deal more flexibility.

One of the largest problems with evolvable hardware is how the system scales as the circuit function increases. The poor scaling of the genetic algorithm is due to two features; the larger circuit size required (and the consequently larger genetic material), and the larger number of test-cases in the evaluation function. One way to combat this involves decomposing the problem into subsystems [21]. This reduced

granularity shrinks the search space of solutions and improves search performance at the cost of the flexibility to construct novel circuitry.

A similar method proposed by Torrensen suggests allowing the evolutionary process itself to subdivide the process, increasing evolution complexity [44]. The area of an FPGA is divided into subsets of cells, and the function of each of these subsets is decided manually or by the evolutionary process. The first system, dubbed "partitioned training vectors" involves feeding the complete input information individually to each subset, but only reading off a subset of the output bits from each region of cells. In this way, each subset can distinguish a few answers perfectly and the other input combinations are identified by another group of cells. A parallel is drawn with artificial neural nets, where evolution is applied to the connections between these subsets, and each subset evolves it's functionality locally. To demonstrate the efficiency of such a system using the vector approach, a character detection system was evolved, capable of classifying images of letters of size 5 by 6 pixels. The number of generations required to train the classifier dropped substantially as the number of allowed subsystems increased. More success was found applying this technique to a prosthetic hand controller which achieved better performance than the artificial neural net equivalent. A similar decomposition strategy has been proposed in [40], with the aim of reducing the number of generations needed to sufficiently explore the search space by automating the division of problems when the genetic algorithm appears to be making no progress and is stalling.

A novel approach to shrinking the search spaces involves using compression techniques to reduce the size of the genetic material. [11] uses Lempel-Ziv compression to limit the genome complexity and suggest directing research into evolvable hardware scaling to considering only "active" areas of the genetic material and limit the search space to only include meaningful distinctions between circuits.

These methods all deal with reducing the search space or dividing the problem into more manageable workloads. The problem remains with scaling inefficiency in regards to the number of test cases to be applied to a circuit during evaluation. This other, less explored facet of the scaling issues appears in the form of the number of evaluations required to calculate the fitness function. For evolved circuitry with n input bits, the fitness function requires $O(2^n)$ time to evaluate; for example, a 2-bit adder requires evaluation against all 16 ($2^{2\cdot2}$) possible combinations of 2 2-bit numbers, whereas for a 3-bit adder this value grows to 64 ($2^{2\cdot3}$). For practical evolution of larger circuits this requires some careful thought.

2.2.2 Application

Part of the charm of evolutionary hardware is it's vast capacity for strange solutions to problems. The nature of a simple fitness function driving the evolutionary process means that any point in the search space which performs perfectly is equally weighted as any other point in the search space.

Nowhere is this more apparent than with the antennas featured on NASA's ST5 mission, which were designed by genetic algorithms [18]. Although this is a slight departure from the circuit design applications mentioned thus far it is a fantastic example of the strengths of evolution derived design. Traditional (human-driven) antenna design is a laborious process requiring seasoned professionals with a great deal of experience in the area. By creating a fitness function which outlined the design requirements of the device, and tuning a suitable genetic algorithm, huge portions of the design process can be automated.

Figure 2.2 features the evolved antenna design which saw use in space. Clearly it is a novel design, looking like an art piece more at home in the Tate Modern than a piece of space-grade equipment, and such designs are typical of evolved systems. The antenna performed successfully throughout the operational lifetime of the spacecraft. This avenue of design provided completely fresh ideas, unburdened by the prejudice of previous success. The antenna achieved a larger range of detection angles than the hand designed counterpart and required only 3 person-months of work to set up and monitor the evolutionary process, rather than the 5 person-months of work which would otherwise be required.

The list of evolutionary hardware successes in digital ciruits is long [36]; from logic circuit synthesis [46] to high bandwidth image filter evolution [35][1]. Using tournament selection and elitism Higuchi et al. developed a custom evolvable hardware chip for use in telecomunications and robots [15], claiming successful evolution of robot controllers and ciruits performing lossless image compression, outperforming existing standards. All this was performed on off-the-shelf components, designed to accelerate the evolvable hardware process for gate-level evolution [15]. A similar success story with hardware designed for evolution takes the form of a prosthetic hand controller with a training time of only 10 minutes to adapt to a new user [19].

In the name of robustness, adaptability and complexity, POEtic tissue [45] is an effort to build an extended evolvable hardware platform inspired by earlier work on phylogenesis (analysis of the evolution individuals and species), ontogenesis (development of single-molecular organisms), and epigenesis (the

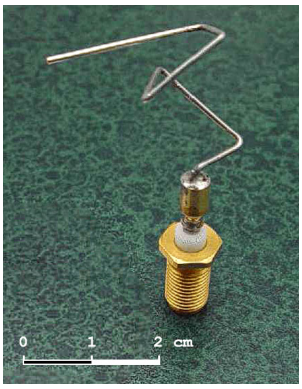


Figure 2.2: NASA evolved antenna [18]

learning over an individuals lifespan) [33]. These three traits inspire the genotype, mapping, and phenotype layer of the hardware system proposed. This system would act as an evolutionary hardware specific replacement for current FPGA systems.

Adding to the literature regarding the evolutionary scaling issue, specifically in the domain of evolvable hardware, Modular Evolvable Hardware (MEH) [17] performs functional decomposition of the overall hardware requirements and then individual module evolution.

2.2.3 Arithmetic Circuits

Binary arithmetic has been used as a benchmark problem for evolvable systems. The most common applications of genetic algorithms within the hardware space are as learning robot/prosthetics controllers or fast image filters; however in [28] Kalganova et al. demonstrate the capacity for evolvable hardware to generate 1-bit addition circuits. These circuits are capable of adding up to 3 different variables and provide the carry-out bit of a full-adder.

Successful evolution of robust 2-bit addition was achieved by Miller et al. through the deployment of tournament selection [24]. This was evolved in the presence of an imperfect fitness function and used the adjustable selection pressure of tournament selection to develop a fine tuned genetic algorithm tailor made for various noise levels.

2.3 Dynamic Problems

Dynamic problems are challenges which change over time. These can take many forms, from scheduling [26], to fault mitigation, and optimisation problems [3]. The vast majority of evolutionary computing research focuses on static problems, but the dynamic variant are more indicative of real problems. In [26], Lin et al. develop a genetic algorithm which schedules continuously arriving jobs. A series of dynamic qualities, including timing information and job importance, fed into a multi-objective fitness function. This approach improved over current deterministic job scheduling schemes. [3] details the use of distinct subpopulations in a multi-population structure to improve diversity and better tackle the breadth of a dynamic problem.

2.3.1 Fault Tolerance

A dynamic problem specifically in the domain of evolvable hardware is fault tolerance. In hardware design, fault tolerance is defined as a systems capacity to remain functional despite component failure. This is usually in reference to post-fabrication faults which can effect chip yields, but evolutionary hardware also looks extensively into faults which occur some way into the operational lifespan of a product; this can be component failures due to environmental conditions or an aging device.

Genetic algorithms by their nature produce designs which are resilient in the face of change [25][41]. If a solution is particularly fragile, a simple single mutation could cripple its performance, the probability that an individual has successful offspring is therefore directly tied to the probability an individual is able to perform well despite being subjected to random mutations. More fragile designs are often mutated into obscurity whereas ones with natural fault tolerance endure. Over time this tendency to maintain a population of individuals capable of surviving minor changes leads to a naturally fault tolerant solution.

Amplifying the effects of this subtle pressure towards fault tolerant design is a mainstay of applied evolutionary hardware research.

In [14] a robot welding controller designed to trace ditches comes equipped with an evolvable hardware backup system which is activated should a hardware fault be detected. Controllers such as these conventionally use FPGAs anyway [29]; the economy of bespoke high-performance industrial requirements often make FPGAs the best choice (as detailed in Subsection 1.1.1). Because of this there is a prime opportunity to deploy evolvable hardware to recover from a fault which would otherwise require a complete hardware overhaul. [14] successfully demonstrate such capacity.

Along with Thompson's genetic algorithm exploiting undefined chip behaviour [42], in another paper Thompson also notes that evolvable hardware has designed systems which rely on undefined temperature sensitive component behaviours [43]. In order to breed solutions capable of functioning on different chips and in a broad array of conditions, an extended fitness function was developed to apply selection pressure to temperature robustness qualities. This fitness function repeated the evaluation step on a series of different FPGAs at different temperatures and then combined the results.

By defining a list of the faults a device is expected to encounter during it's lifespan, one can design a fitness function which tests a given circuit under normal conditions and faulty conditions [41][22]. This technique applies explicit evolutionary pressure to find designs which can tolerate a specific set of known and understood faults. Alongside the proposed fitness driven fault mitigation approach, [22] introduces an alternative mechanism which takes the highest performing individual in a fault-free system and explores the performance of mutants generated from it in a faulty environment; this emphasises the natural fault tolerant properties of genetic algorithms. In some situations the mutants could not recover from the injected fault, in these cases the genetic algorithm was to restart and continue searching through the design space under the conditions of the fault.

Using a conventional genetic algorithm, along with tournament selection, and elitism Lohn et al. demonstrate the capacity of evolvable hardware to reduce the impact of "stuck at zero" faults [27]. They conducted their experiments on a simulated FPGA, attempting to evolve a quadrature decoder, a device which indicates the direction of rotation of a wheel. The system was evolved in the presence of the fault and was capable of operating perfectly despite this.

In the face of major advances in chip fabrication technology, due to the fact transistors are approaching the atomic scale, post-fabrication chip faults are a major issue. Walker et al. demonstrate the capacity for genetic algorithms to design hardware which would improve fabrication yields [48]. The structures they successfully evolve prove highly tolerant to variations in the fabrication process and they demonstrate the possibility to improve the success rate of chip manufacture through deployment of evolutionary hardware design.

The space industry naturally wants to emphasise longevity in their design goals, this was a clear desire with the NASA's CT5 mission, and as previously mentioned, the genetic algorithm devised antenna performed to the highest standards until the end of the missions lifespan [18].

Circuitry hosted in space is subject to a breadth of extreme conditions; temperature drifts, radiation, and poor serviceability to name a few. Radiation is a particular issue, bombarding electronics with electrons, protons, and gamma rays causes huge damage. The space industry has traditionally used radiation-hard materials to achieve radiation resistance. These materials resist the ionising effects of radiation and reduce the probability of radiation-induced silicon faults. These materials are expensive and definitely not the only way to create long-life deep space electronics. Sotica et al. propose using genetic algorithms to drive in-situ hardware reconfiguration to bypass faulty areas [38]. Experiments were performed on an FPTA with evolution controlled by an external Stand-Alone Board-Level Evolvable System (SABLES), capable of performing an evaluation in 2ms. In experiments bombarding the chip with up to 350krad, the system was usually able to recover functionality, and there were significant performance improvements over the unevolved comparison chip.

In [50] a brushless DC motor control circuit with built in hardware capable of executing a genetic algorithm, was shown to be capable of self-repairing damage to up to 27.8% of the hardware resource. Their system requires the complete shut down of the motor while evolution finds an alternative control circuit configuration.

Fault tolerance in conventional hardware design requires redundancy in some form. Predictive pre-emptive instruction rescheduling on a different processor core [37] uses multi-core redundancy to mitigate the cost of faulty instruction execution. Cross-core redundancy is used by core salvaging [32], which shares execution pipeline resources between cores, in order to reroute execution pipelines around faulty components. This is similar in principal to StageWeb [9], which also features pipeline rerouting but with an emphasis in scaling flexibility.

This is a very different approach to evolvable hardware centric fault tolerance which exploits flexibility. Many of the aforementioned modern fault tolerance techniques do little to improve the performance of faulty architectural processor components, and rely on the ability to reroute around them.

The bulk of evolvable hardware research addresses "toy problems", and is a long way from being able to compete against conventional fault tolerant design techniques [10]; but there have been promising results and unprecedented ability to design circuit structures with fault survivability properties.

Chapter 3

Project Execution

The core of this project involves building a flexible evolvable hardware platform, upon which novel genetic algorithms can design solutions which can be evaluated in a narrow context. Due to the high demands of an iterative design cycle, reducing the evaluation time within the genetic algorithm is at the forefront of the design process; to this end a simulated FPGA is used to simplify development and improve the speed of evolutionary training.

A single problem has to be chosen to which we will tailor the genetic process.

3.1 Project Management

The source code is hosted online at https://github.com/AlexDalt/MalleableHardwareAdder, written in C to maximise training performance and Git was used for version control. Regular supervisor meetings ensured the project stayed focussed. The aims of the project are limited to evaluating various performance metrics of different and new evolvable hardware processes.

3.2 Design

The high level design choices were made to improve development time and discern insights into evolutionary hardware with dynamic problems and the scaling issues involved. The system should be flexible so parameters can be altered between searches, and information about the population performance should be offered as the genetic algorithm runs.

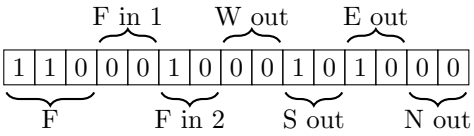
3.2.1 Phenotype to Genotype Mapping

The mapping from binary string to FPGA configuration is one of the first design choices to be made. The simulated FPGA will be of configurable width and height, and each cell will operate as described in [42] and Figure 1.1. Each cell takes input from each of it's neighbours, performs a binary function F on chosen input(s) and sends distinct configurable outputs to each of it's neighbours. The binary function F or any of the inputs can be sent to any of the outputs.

To fully describe the operation of a cell a 15-bit binary string was chosen. This is arranged in memory as 2 bytes per cell, of which one bit is unused. The least significant byte defines all the outputs, 2 bits per neighbour (from least significant bits to most significant bits: north, east, south, and west), where a 0 is defined as a direct mapping from the northern input to that output, a 1 means the eastern input is used, 2 the southern, and 3 western.

The most significant byte describes the function F. The low 2 bits describe the first input (00 $\rightarrow$ north, 01 $\rightarrow$ south, 10 $\rightarrow$ east, 10 $\rightarrow$ west), the next 2 bits describe the second input in the same manner. The next 3 bits describe the operation performed (000 $\rightarrow$ OFF, 001 $\rightarrow$ NOT, 010 $\rightarrow$ OR, 011 $\rightarrow$ AND, 100 $\rightarrow$ NAND, 101 $\rightarrow$ NOR, 110 $\rightarrow$ XOR, 111 $\rightarrow$ XNOR).

A description of the complete string can be found in Figure 3.1. In this example the cell performs an XOR operation on the inputs coming from the cells to the north and south. The southern input is mapped to the eastern output, and the northern input is mapped to the western output. Both the northern and southern outputs get the result of the XOR operation, as under this scheme an output cannot be mapped to it's input. In the situation that the number associated with an output refers to the input coming from the same direction, the result of function F is used as input instead. As the northern



Direction	Encoding
NORTH	00
EAST	01
SOUTH	10
WEST	11

Function	Encoding
OFF	000
NOT	001
OR	010
AND	011
NAND	100
NOR	101
XOR	110
XNOR	111

Figure 3.1: Genetic representation for a single FPGA cell

output is specified as 00 (NORTH) and the southern input is specified as 10 (SOUTH) they both receive the output of the function.

Cells are defined from left to right, top to bottom on the FPGA. The binary string required to describe an FPGA of width w and height h is $2wh$ bytes long.

This mapping was chosen because it is loosely in line with the number of bits used by [42] and it allows concise and complete expression of the FPGA functionality. The order in which cells are defined also follows [42], and means any crossover occurring tends to keep the horizontal sections consistent with each other.

3.2.2 Problem Choice

The choice of problem is a key component in designing and configuring a genetic algorithm to tackle it.

Within evolutionary hardware there are a number of benchmark problems, these usually take the form of robot controllers, prosthetics controllers, or signal processing circuits. Considering generic computer hardware, binary addition appeared to be an interesting commonplace problem which would require a great deal of genetic algorithm tuning.

With a desire to explore dynamic problems, addition also opens up the possibility of introducing subtraction as a related but distinct problem. To this end, evaluation will consist of checking both addition and subtraction, both of these will have an associated weighting which can vary over time.

Despite appearing to be a trivial problem, 2-bit binary arithmetic has a number of nuances which might hinder the progress of a genetic algorithm. A greedy genetic algorithm could quickly achieve a relatively good accuracy, but through uneconomic resource use, is completely unable to progress. Genetic algorithms struggle with backtracking, and would prefer to blindly hold on to the current best solution, mindlessly exploring related FPGA configurations. The arithmetic problem also requires information to be chained together, the result from one single binary calculation impacts the next binary calculation, so a genetic algorithm needs to demonstrate the capacity to structure components and chain together functionality.

3.2.3 Evaluation in the Genetic Algorithm

Developing a genetic hardware on a physical FPGA requires a large amount of hardware design language knowledge; without it, the work required could have constituted the entire project. Also, with physical hardware, the time required to evaluate each population member can be measured in seconds, whereas in simulation each evaluation requires a fraction of a second. These factors made the creation of a streamlined software FPGA simulator as the evaluation back end for the genetic algorithm an enticing prospect.

The simulation works as a self contained module which exposes a series of evaluation functions to the evolutionary system. It includes intermediate variables which are updated throughout execution, within these variables 0 and 1 represent the binary values 0 and 1, and 2 represents an “undefined” variable. When evaluating binary addition or subtraction, the FPGA is initialised to contain entirely “undefined” values, then the individual bits constituting the binary representation of the two numbers which are to be added are fed to each cell across the top of the FPGA in turn, and a control bit (designating addition/subtraction) is fed to the top left cell. The FPGA execution consists of two steps, a tick, and a tock; the tick involves pulling values from the surrounding cells and storing them internally, then the tock involves performing any computation and storing the necessary values in the output buffer associated

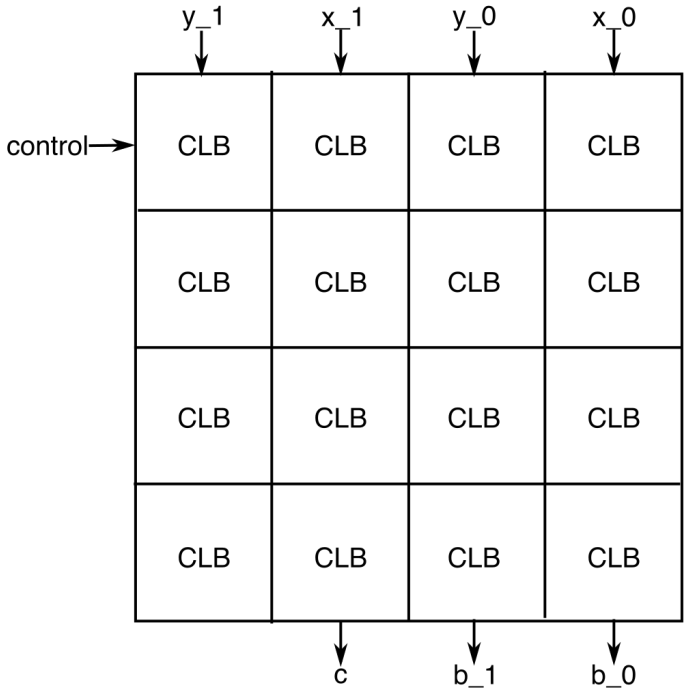


Figure 3.2: Diagram detailing how evaluation parameters are fed into the FPGA and answers read off for a 2-bit arithmetic problem

with each neighbour. The tick tock cycle continues until the southern output of the bottom cells is no longer undefined and stabilises on a constant output; if this doesn't happen within 64 iterations the simulation halts. The system is deamed stable if the output doesn't change after a tick tock cycle. The halting compromise is necessary as some FPGA configurations have an oscillating output which never settles. This process is repeated for all possible additions and subtractions and then the individual is scored based on the number of correct bits read off the bottom most cells.

Figure 3.2 demonstrates the framework evaluating a 4x4 FPGA attempting binary arithmetic. Two 2-bit numbers, x and y are deconstructed into their constituent binary values $\{x\_0, x\_1\}$ and $\{y\_0, y\_1\}$, where $x = x\_0 + 2^{x\_1}$ and $y = y\_0 + 2^{y\_1}$, these values are made available to the top row of FPGA cells, and a control bit (indicating addition or subtraction) is presented to the top left cell. After FPGA evaluation has concluded, 3 bits are read from the bottom, the two bits constituting the 2-bit answer (b_0,b_1) and the carry-out bit.

The FPGA initialisation step is essential to ensure a deterministic evaluation process. Without the initialisation, depending on the order in which evaluations are performed, the FPGA can settle on an correct (or incorrect) output due to the time taken for new values to propagate through the circuit. This leads to identical configurations which receive wildly different scores in two separate evaluations.

3.2.4 Fitness Landscape

When tackling a problem with a genetic algorithm, it is important to have an underlying appreciation for the space of potential answers and how these are related to one another. In the case of evolutionary hardware, mapping the entire search space is unfeasible, even for a relatively small FPGA configuration. The genetic material describing a 4x4 FPGA is 32 bytes long, therefore there are $2^{32 \cdot 8}$ potential solutions, any enumerated bruteforce search processes can only uncover a miniscule, uninteresting and unrepresentative corner of the fitness landscape within any reasonable time frame.

Furthermore each CLB has 7 axes of freedom, multiplying this across the entire 4x4 FPGA there are 112 dimensions. Visualising a fitness landscape which exists in 112 dimensions is near impossible. In order to gain a better understanding of the nature of the underlying landscape, a measure of the systems "ruggedness" is taken, as defined by Barnett [2]. For mutation rate M, fitness function f, mean mutant fitness function μ, covariance and variance functions cov and var, Equation 3.1 details how to calculate ruggedness at a specific mutation rate, $\rho(M)$.

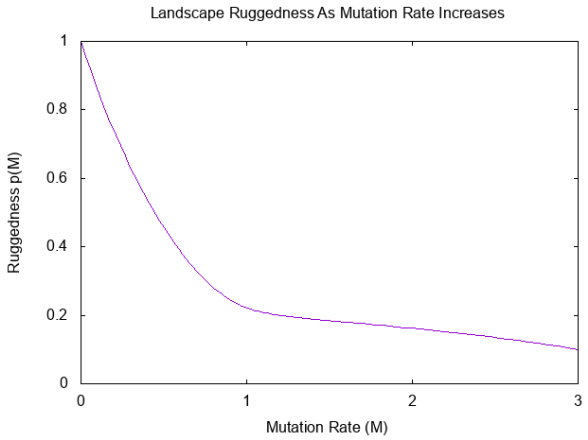


Figure 3.3: Landscape ruggedness at different mutation rates

$$\rho(M) = \frac{cov(f(g), \mu(g)))}{var(f(g))} \tag{3.1}$$

This exercise highlights the nature of the underlying landscape and the function traversing it. This ruggedness measure is equivalent to the correlation between an individuals fitness and the fitness of any offspring it produces[1]. Figure 3.3 outlines the relationship between the mutation rate and ruggedness. Even with a small number of mutations the correlation between parents and their children is abysmally small. As the mutation rate approaches 3 this correlation reaches a value of 0.1. This highlights the expected failure rate in an individual's children and the infrequency of viable individuals.

An ideal fitness landscape for genetic algorithms is one which has gentle hills which can be ascended, where aggressive mutation does little to render a potential solution inert. However, the landscape hinted at in Figure 3.3 is one which there are few effective mechanisms to navigate, the current best option is a highly targeted genetic algorithm.

3.2.5 Genetic Algorithm

The basis for the genetic algorithm follows similar work done by Thompson [42], but with a number of alternate avenues explored for problem specific improvements.

Parameter choices The basic genetic algorithm has a number of interplaying parameters which affect the genetic process in some interesting ways. Population size is one such parameter which we must consider; the number of individuals in each generation can drastically effect execution. As you grow the population you improve the probability of finding a solution by a given generation, but this has diminishing returns and makes the evaluation time considerably more laborious. A smaller population tends to be less diverse and is more easily dominated by a single effective solution and it's descendants.

Another important variable in the genetic process is the mutation rate, this defines the frequency with which we are to expect mutations in the offspring of a given generation. With a high mutation rate, one climbs the evolutionary ladder quicker and can make larger leaps across a chasm of low fitness. However, in less robust populations, a high mutation rate can eradicate fragile solutions, leading to a weak population never capable of climbing to the narrow peak of high fitness.

Selection mechanism The initial selection method will be rank based selection, which is by far the most popular selection method observed in pre-existing work; however, tournament selection will also be explored. Proportionate selection methods are ignored as they discriminate between individuals with great vigor initially, when the population is developing solutions improving from 15% to 50% accuracy, which it does with relative ease, this hurts population diversity. Later in execution, the proportionate selection methods, struggle to separate individuals performing with an accuracy of 96% and 100%.

[1] The genetic algorithm used to gather the data was calibrated with; a population size of 50, elitism, tournament selection of size 20, probability of crossover 0, a multiobjective-fitness function incorporating diversity weighted at 20% accuracy (see Section 3.2.5), and the mutation rate varied between experiments. This parameter configuration was arrived at after the experimentation in Chapter 4.

Rank-based selection is performed primarily by ordering the members of a population based on their fitness and then associating the probability of selection to a linear function on their position within the ranking. By adding a "skew", which shifts the probability function to one closer resembling a quadratic equation, one can emphasise the best performing individuals over the more balanced linear selection function.

A single parameter change allows us to change the relative weightings of high scoring and low scoring individuals in a population via Equation 3.2. Given population size n, individual x, the individual's position in the ranking r, and skew s (between 0 and 1, where 1 is linear selection and 0 is quadratic), the equation generates a score between 1 and the population size plus 1. By varying the skew value insight will be gained into the impact of a more or less discriminatory selection mechanism. Given the score generated by Equation 3.2, Equation 3.3 generates the probability than an individual is chosen during a single selection step.

$$S(x) = \frac{1-s}{n} r^2 + sr + 1 \tag{3.2}$$

$$P(x) = \frac{S(x)}{\sum_{i=0}^{n} S(x_i)} \tag{3.3}$$

Tournament selection involves randomly selecting k individuals from the population, and the one with the highest fitness is chosen to fill a place in the new generation. The process is repeated until the new generation is sufficiently populated. This adds another element of randomness to the genetic process, giving poorer solutions an opportunity to be in a tournament with other poor solutions and therefore has the ability to improve the diversity of the new population at the expense of running the risk that better solutions might be chosen for tournaments infrequently (or not at all).

Roulette wheel selection is not well suited for the task of binary addition within evolvable hardware. The fitness increases associated with a slight improvement in the design are huge, and this often leads to marginally better solutions eclipsing solutions which given more time could have evolved into equivalent (or even improved) FPGA configurations.

Elitism Selection will also incorporate elitism, this directly copies the best individual to the next population and ensures that the most fit member of the population is preserved unmolested by mutation. The fragility of potential solutions has been highlighted by Figure 3.3 and demonstrates the need to employ some preservation mechanism.

Multi-objective Fitness Function Extending the generic fitness function to a linear combination of multiple different functions applies evolutionary pressure to a multitude of useful different factors. For evolvable hardware the most important are often accuracy and size.

A small design is desirable as the smaller a design, the less power consumed by the device. To encourage smaller configurations a fitness functions can incorporate the number of cells inactive in a design into the evaluation procedure. However this functionality is not extensively tested.

Early testing, see Section 4.1, highlighted the need for a mechanism to cultivate population diversity. Too often did the system stumble into an evolutionary dead end and refuse to backtrack far enough to explore alternative routes.

Given the extremely rugged landscape demonstrated by Figure 3.3, a fitness function incorporating some measure of diversity seems especially prudent. [6] proposes using a distance measure from each individual to each other individual in the population to generate a score for the mean squared distance from a given member of the population to the rest of the population. This metric can be incorporated into the multi-objective fitness function to preserve designs based on novelty alone. These novel designs continue to experience evolutionary pressure and a series of distinct optima should exist within the population, improving the probability that the system avoids evolutionary dead ends and discovers a working solution.

The proposed diversity metric does not scale well as the population size increases. With a population size of n the evaluation step requires $O(n^2)$ individual distance measurements.

$$f(x) = w_{correct} \cdot f_{correct}(x) + w_{size} \cdot f_{size}(x) + w_{div} \cdot f_{div}(x) \tag{3.4}$$

Equation 3.4 summarises the multi-objective fitness function around which this evolvable hardware system is developed, where w_x is the weighting associated with the fitness function f_x. $f_{correct}$ is the original fitness function of the process, f_{size} is a measure of how small and efficient a solution is, and

f_{div} provides the mean distance from an individual to every other individual in the population. If a population has a high f_{div}, the population is diverse. The weightings are all configurable and allows for fine grain sculpting of the evolutionary process. The size metric ensures power efficient solutions, and the diversity encourages a genetic variety. These need to be balanced with the weight given to solutions to be correct, for example; it should be more important to be correct than small.

Crossover A mechanism for crossover is built into the platform. After selection the new population of strings will be randomly paired up, and given a probability, will undergo single point crossover. If crossover is to happen, a random point in the 32 byte string is chosen and the two paired individuals will swap all values beyond the randomly chosen point. Once the entire population has undergone a similar procedure the population experiences mutation.

This approach to crossover is not standard, but it is functionally identical to the mechanism outlined in Chapter 2. Normally, when a slot is to be filled in a new population one or two parents are selected, the latter arrangement chosen over the former with probability given by the crossover probability. Instead, however in this system a complete mating pool is constructed, paired at random, and then genetic material is exchanged with probability given by the crossover probability. This shuffling performs a function similar enough to the text book crossover definition [7] but allowed for more cleanly developed code with clear seperate sections for selection and crossover.

3.2.6 Coevolution

Coevolution will be implemented in a manner similar to the scheme outlined in [4]. This will allow coevolution to act as an effective method of employing selective evaluation and evolutionary pressure to a host population in the context of evolvable hardware. The selective evaluation will both act as a way to more effectively discriminate between FPGA configurations and also to reduce the size of the set of evaluation problems to improve scalability.

A separate population of identical size to the core population of FPGA configurations will be generated at random. Individuals in this population will consist of a small set of bitstrings of length of $2b$ where b is the number of bits required for each operand used in the addition (or subtraction). The first half of the problems will pertain to addition, and the second half represent subtraction problems. When using coevolution the evaluation step is slightly modified. To evaluate both the host (FPGA configurations) and parasite (list of problems) populations each host is randomly paired with a parasite, and the host is evaluated as before, save that the equations fed into the FPGA are defined by the parasite. The parasite gains a score of $m - f_{add}(x) - f_{sub}(x)$ where m is the maximum score possible.

Then the parasite population undergoes selection and mutation in an identical fashion to the host population.

Variable virulence By varying the parasite virulence one can reduce the intense discrimination which could lead to population disengagement and would be exasperated by the fitness landscape fragility. To this end, the ability to vary the parasite aggression is a central feature of the evolutionary system.

After the parasite is initially scored this score is then translated through a predefined function. The score is divided by the maximum parasite fitness so the all fitness exists between 0 and 1. Then given a virulence between 0.5 and 1.0 (where 1.0 is maximally virulent), the score is translated via Equation 3.5, where m is the new modified score, s is the previous score (normalised to be between 0 and 1), and v is the virulence. This equation is taken from the coevolutionary work done in [4].

$$m = \frac{2s}{v} - \frac{s^2}{v^2} \tag{3.5}$$

Improved scaling Coevolution has the potential to improve how an evolutionary hardware system scales. The literature has extensively explored how to tackle the ballooning search space size as you increase the scope of the problem being addressed, but little has been done to improve the scaling of the evaluation function itself. As the scale of an evolvable hardware problem is increased, the set of test cases grows at an exponential rate. This thesis suggests that by coevolving a population of problems against a population of FPGA configurations we can aggressively reduce the number of test cases used for a fitness evaluation, and therefore improve system scaling.

3.2.7 Fault Modelling

In order to explore the capacity of evolutionary algorithms to design fault tolerant circuitry and also the use of evolutionary hardware as a fault recovery mechanism in itself, the definition of what constitutes a fault in this context must be narrowed down. The primary faults explored are ones which are generated on a device as it is used, these faults are mainly caused by component wear out and unpredictable environmental effects [10]. To this end, the FPGA simulation will be extended to model faults in the internal logic of an FPGA cell. These faults will manifest as clamping the output of any logic operation to 0, 1, or 2 (the value reserved to represent an undefined value). The evolutionary algorithm is unaware of the faults, the only perceptible difference is in the evaluation of a population, which executes as expected and continues to encourage the population to improve in this new context.

For most of the testing, faults will be randomly generated at runtime; a neccessary feature due to the unpredictable nature of Darwinian search, but for specific fault recovery cases, expected faults can be specified. An interesting case arises when a model 2-bit adder is embedded in the population, and then a critical fault is induced. Observing the sharp recovery of the system indicates the potential of such a fault recovery mechanism, this is further explored in Subsection 4.2.1.

Along with observing how evolution can help a system recover from a fault, an analysis into how a system evolves in the presence of intermittent "sticky" faults could give insight into the uncomfortable world of irregular fault behaviour. The evaluation back end of the evolvable hardware platform can activate and deactivate faults as required. Evolving in the presence of such an evaluation mechanism should gradually encourage a population which can perform well irrespective of an active fault or not.

The reason the faults exist is of no consequence to the evolutionary process, but grounding this exploration in expected fault models will help to generate practical recovery methods [27].

3.2.8 Dynamic Weightings

The second set of dynamic problems being explored involves shifting the relative weightings applied to ADDs and SUBs in the evaluation step during execution. The automatic reweighting during execution will allow for the modelling of a system which has shifting execution priority. In order to maintain a balance between correctness, size, and diversity in the fitness function, the weighting associated with addition and subtraction are changed together; as one increases the other decreases. This means that once tuned to effective weightings, the exploration of a dynamically shifting problem is unburdened by the complications of a sensitive fitness function.

Irrespective of the weightings associated with addition and subtraction, the evaluation step collects complete information about the performance of each. This is then multiplied by both weightings and divided by the sum of the two weightings. This produces a value within a constant window. The extended fitness function which was used is given in Equation 3.6.

$$f(x) = \frac{w_{add} f_{add}(x) + w_{sub} f_{sub}(x)}{w_{add} + w_{sub}} + w_{size} f_{size}(x) + w_{div} f_{div}(x) \tag{3.6}$$

3.3 Implementation

All the flexibility described in the above design criteria is realised by modifying the headerfiles of my evolvable hardware source code. The implementation was written in C to ensure efficient execution and quick development cycles. The software can be cleanly separated into the GUI, the evolution front end, and the FPGA simulation back end. Collectively, they consist of more than 1500 lines of code, with an extortionate amount of flexibility, capable to scale the size of the FPGA and the problem well beyond reasonable limits.

3.3.1 FPGA Simulation

The header file for the simulation back end is contained in Appendix A, and shows the key data structures and functions exposed to the components performing evolution. The simulator can convert bitstrings of appropriate length into an FPGA of a given size, and given an FPGA the simulation can also perform evaluations and simulate faulty logic within any number of cells.

Converting a bitstring to an FPGA is achieved with the function `bitstring_to_fpga`, which takes a pointer to an `FPGA` struct and a pointer to a `char` array as arguments. The cells of the FPGA are iterated over left-to-right, top-to-bottom, to fill a shape defined by `FPGA_WIDTH` and `FPGA_HEIGHT` and for

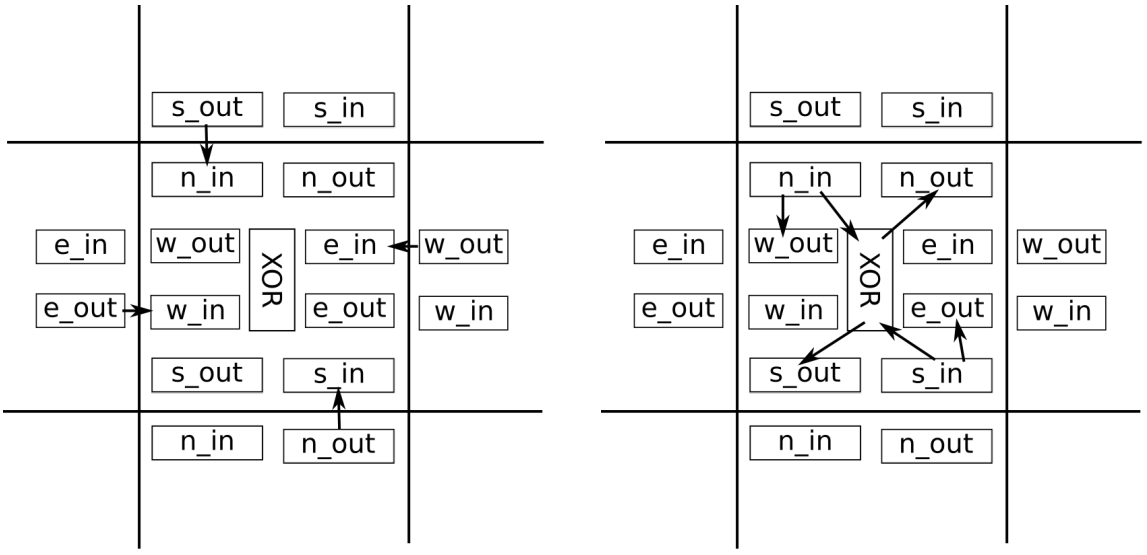


Figure 3.4: Dataflow for one complete compute cycle for an FPGA cell configured by the bitstring in Figure 3.1; (a) tick (b) tock.

each cell the next two bytes are read from the char array. The meaning of the bytes is parsed out and imbedded in the FPGA struct via an almost direct mapping (north is encoded as 00 in the bitstring, and is represented as the value 00 in the `enum` type `Direction` whenever a direction is stored on the FPGA).

When evaluating a given input the two values to be operated upon are stored in the array `FPGA.input[ FPGA_WIDTH ]`, then `evaluate_fpga` is called. Each variable and intermediate value (any value not defining the operation of the FPGA) is intialised to 2, to represent an undefined value. The simulation then performs a `tick` step, which iterates over each cell pulling data from the neighbouring cells, followed by a `tock` step, which performs relevant computation and stores output data in relevent output buffers to be read by the following `tick` step. These steps continue in turn until the output is constant (or an iteration cap is reached in which case it halts). At this point the function returns and the calling entity can inspect cell data and score accordingly.

The `tick` step populates each FPGA cell's `n_in`, `e_in`, `s_in`, and `w_in`, with the relevent data from each neighbouring cell. For example, when assigning `n_in` the southern output (`s_out`) of the cell to the north is copied in. This occurs for all cells except the ones on the edge, if trying to access a cell beyond the scope of the simulation the variable is assigned the value representing undefined, 2. This happens uniformly except in the case of the north most cells, in which cases their `n_in` is the relevant value from the `FPGA.input` array. Another exception to the rule comes in the form of a control signal input; the cell at the top left is fed a control signal by way of it's `w_in` variable, instead of being undefined it copies the binary value from `FPGA.control`. This is intended to distinguish an addition from a subtraction; for an addition the value will be set to either 0 or 1, and for a subtraction the value will be set to the other, the association has to be learned by the FPGA configuration during evolution. The way data is made available to an FPGA is captured by Figure 3.2

`tock` consults the variables storing information about the operation performed by the cell, and performs said operation with the relevant x`_in` variables, where x represents one of the cardinal directions. Then each cell's `n_out`, `e_out`, `s_out`, and `w_out` is assigned in accordance with the cells internal mapping (eg. `n_out`, the buffer the cell to the north copies for it's `s_in` value, could be directly copied from the function output, or any of the x`_in` variables).

The dataflow for a single CLB (configured by the bistring in Figure 3.1) during a complete tick tock cycle is demonstrated in Figure 3.4.

Deploying this simulation gave subsecond evaluation times for an entire generation of 4x4 FPGAs. These performance benefits, whilst removing the capacity to exploit strange undefined chip behaviour as experienced by Thompson [42], it did allow considerably quicker software development, and the creation of evolutionary hardware within the strict confines of defined behaviour which improved the ability to understand the machinations of their success.

3.3.2 Evolution

Beyond calls into the simulation back end and inspecting cells after the FPGA has been evaluated, the evolutionary front end is a distinct entity which operates almost solely on the abstract bitstrings which constitute the population(s) being evolved.

Initially a population of `POP_SIZE` bitstrings of length `STRING_LENGTH_BYTES` are randomly generated using the pseudorandom `rand()` syscall, which (although cryptographically insecure) is suitably random for seeding a random population. `STRING_LENGTH_BYTES` is defined as `2 * FPGA_HEIGHT * FPGA_WIDTH`. If `COEVOLUTION` is defined as `1` then a population of parasites of size `POP_SIZE`, is also randomly generated. Each parasite contains an array of size `PARASITE_SIZE` containing `char`s, each of which represents a challenge posed by the parasite.

The `evolve` function is passed the randomly seeded population(s) and curates the evolutionary process. First it randomly seeds an array of potential faults, these are stored in an array of `struct`s labelled `Fault`. `Fault` specifies the coordinates of the cell experiencing the fault, and some auxiliary information depending on the type of fault being simulated. Whenever an FPGA configuration is created, the fault information is copied into it. The function then begins on an unending `while` loop which conducts the evolutionary process until halted by the user. If coevolving, the parasite population is shuffled and then each member of the host population is sent with it's random parasite counterpart to the `evaluate` step. This function also requires a pointer to the entire host population. When this function returns the array `Individual.eval` has been filled. `eval[0]` contains the measure of the individuals accuracy (already modified by the weightings given to addition and subtraction), `eval[1]` denotes the number of cells on the configuration in the `OFF` position, and `eval[2]` is set to the diversity measurement. While each `Individual` is evaluated, the mean value for each fitness measurement is collected and the current most fit individual is kept track of. If coevolving the most fit individual is then exhaustively tested against all possible input data and the evaluation metrics are sent to the GUI handler to redraw the screen. Information about the population fitness, and best case fitness are logged in an external file. An iteration counter is incremented, the new population(s) are generated via the `new_pop` function and if `ELITISM` is defined as `1` the individual deemed most fit is then copied directly over the the 0th element in the new population array. Then the user input buffer is checked for keyboard input, and any relevent modification made; 'f' activates/deactivates faults, 'd' loads in a perfect adder and assigns highly targeted faults to act as the fault recovery demonstration, 'r' reseeds the population(s), the left arrow key shifts the add/sub weightings towards subtraction, and the right arrow key shifts it towards addition.

`evaluate` when passed an `Individual`, a `Parasite` and a pointer to the host population populates the evaluation variables in both the `Individual` and the `Parasite`. If not coevolving, a pointer to a dummy parasite is passed to the evaluation function. The evaluation is achieved by first generating the relevent FPGA via the `bitstring_to_fpga` function and then iterating over all possible inputs, calling `evaluate_fpga` and then reading the required bits from the bottom of the FPGA, one point is given for each correct bit. This process is completed for both addition and subtraction and then the weighted average of both is stored in `Individual.eval[0]`. Following the evaluation of accuracy, each cell is iterated over, for every cell who's function is denoted as `OFF`, the individual gets 1 point, and the total is stored in `Individual.eval[1]`. `ind_distance` gives a measure of the distance between two FPGA configurations, the distance from the `Individual` being evaluated and each other individual in the population is accumulated and the square root mean squared distance is stored in `Individual.eval[2]`, this step is an obvious scaling bottleneck but provides a diversity measurement which vastly improves the chances of successfully evolving a solution. If coevolving, the set of test additions is defined by the first half of the `Parasite` char array, and the set of test subtractions is defined by the second half of the array. The fitness of an individual is calculated as before (as the total number of correct output bits) and the parasite score is given as the maximum score minus the score given to the host individual.

The function `ind_distance` calculates the Hamming weight from one FPGA configuration to another. Two `Individual`s are passed in and converted into `FPGA`s via the `bitstring_to_fpga` function. Then for every different function and output routing on the configurations, the individuals are 1 further apart.

`new_pop` generates a new population of hosts and, if coevolving, parasites. When coevolving, everything done to the host population is mirrored independently on the parasite population. If using rank-based selection, the population is ordered based on it's (weighted) fitness; this ordering is conducted via a population-specific implementation of quicksort. Each population member is associated a value between 1 and $n + 1$ by the function `ind_prob` based on it's ranking in the population (low fitness individuals have lower rankings), this is the probability of each individual being selected for the next generation scaled to make it an integer. Then for each slot in the new population a random number is generated by `rand()` between 0 and the sum of all values generated by `ind_prob`. Starting from zero

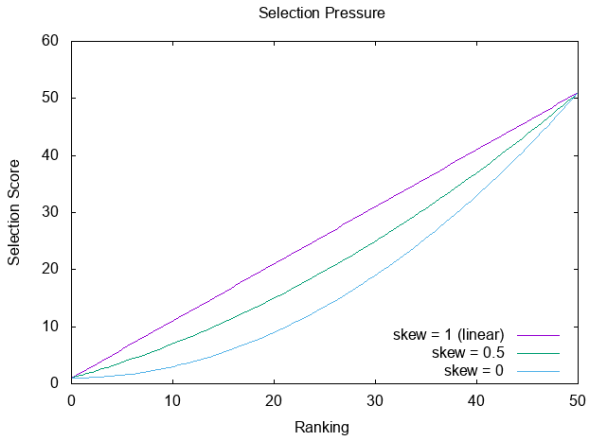


Figure 3.5: The score given to each individual in a ranking with various skew values.

and the lowest ranked individual, the value given to each population member is added to the total. If it exceeds the randomly generated number then that individual is copied into the open position in the new population. If it does not exceed the random number the next individual is checked by adding it's value to the running total, the process is repeated until the random number is exceeded. When all slots in a new population have been filled the population(s) are mutated. To this end, each member is iterated over and, given an expected mutation rate, defined by `MUTATION` and genetic material of length l, each bit in each individual is flipped with probability $\frac{1}{l}$`MUTATION`.

`ind_prob` implements Equation 3.2 to skew the ranking to be more or less discriminatory to highly performing individuals. The skewing parameter, given as s in the equation is defined by `PROB_SKEW` but this function simply returns it's argument passed through this quadratic equation as variable r. Figure 3.5 demonstrates the selection pressure applied to each member of a ranking with a series of different skew values.

In order to store information about the evolutionary process `log_data` writes information about each generation to a `log.dat` file, which later can be processed to develop graphs and better understand the quality of the evolutionary process. The information harvested can be easily modified and extended but for the most part it has consisted of the mean accuracy value and the accuracy of the best case individual. We are not overly concerned with the total weighted fitness function as it is used as a tool to better optimise for accuracy; diversity, for example, is useful for applying evolutionary pressure to preserving a varied ecosystem but means little for a correct answer and it's effects should be visible in measuring the correctness alone.

Crossover is performed after selection. The new population is shuffled, and for each two individuals a random number is generated. If this number is below the `CROSSOVER_PROB` crossover occurs. This involves switching the binary data from a randomly selected point in each individual's bitstring. Only the boundaries between bytes can be selected as crossover points.

If `TOURNAMENT_SIZE` is set to anything other than `POP_SIZE`, tournament selection is used as the selection mechanism. This is done by shuffling a population, taking the first `TOURNAMENT_SIZE` individuals and ordering this new set. The individual with the highest fitness is selected to fill the space in the new population.

All the evolutionary parameters can be defined at compile time to drastically alter the execution of the system, all the modifications can be done within the headerfile `evolve.h`.

3.3.3 Fault Implementation

Faults were implemented by doing a fault pass after each `tick tock` cycle in the function `evaluate_fgpa` and overwriting the relevent output of any faulty cells. Before the `tick` of the following cycle the predetermined faulty cells have their designated outputs clamped to a randomly selected value. The fault information is stored in two arrays within the `FPGA` struct, the data pertaining to fault location and effect is stored in an array of type `Fault`, this has `FAULT_NUM` entries and fully describes the operation of any fault. A second array of size `FAULT_NUM` consisting of integers contains binary data about whether or not a fault is active on the current evaluation and controls the execution of the fault pass.

Figure 3.6: GUI screenshot

Three types of fault are explored; stuck at 0, stuck at 1, and undefined. In the first two the result of any logic operation is stuck at a certain value, this is typical of some sort of short or wearout within a chip. The last type of fault, undefined faults, always produce a value of 2 (representing undefined) from the function. This means any output is indiscernible, a value of 2 is always produced by a function if either of it's inputs are 2s, this models the propagation of uncertainty and a 2 is always incorrect if read as the answer to a problem from the bottom of the FPGA.

All faults are randomly generated by the `evolve` function and can be programmatically activated or deactivated. If the value of `STICKY` is set to 1 faults automatically activate/deactivate every 1500 generations which allows for an in depth exploration of the dynamics of evolvable hardware as a mitigation technique for faults which are sometimes active or sometimes inactive. This procedure, if allowed to find an always perfect solution, also breeds a solution which works irrespective of this specific fault. If the fault being modelled is typical of this hardware this step can be used to develop highly specific fault tolerant designs.

3.3.4 Dynamic Weightings Implementation

Programatically, or by user input, the relative weightings of addition and subtraction can be shifted mid-execution. By pressing the right or left arrow keys the weighting is moved towards addition or subtraction respectively.

3.3.5 GUI

The user interface is controlled by the simulation backend, which exposes a series of simple functions to the evolution front end. These exposed functions allow the triggering of redrawing and reduces the required volume of information for easy redrawing of the GUI. `init_curses` sets up the environment, `redraw` called with correct arguments repaints the user interface, and `tidy_up_curses` cleanly tidies up the GUI.

The GUI was built with the NCURSES library, a free, open source curses emulation. It's used to create terminal user interfaces and was chosen due to the amount of experience with developing user interfaces with curses, not to mention a penchant for state-of-the-80s graphics technologies. Figure 3.6 is a screenshot of the GUI in action. It includes a diagram of the current best-case FPGA configuration along with it's performance metrics and information about the underlying population. The panel on the left and right denote correctly performed addition and subtraction respectively, for a given test if all 3 bits are correct the test is highlighted in green, if all 3 are wrong it is highlighted in red.

The GUI fully describes the functional components of the highest performing FPGA. The function performed by the cell is written in the centre, with the function inputs directly above (N, E, S, W representing the cardinal compass directions). For each neighbour, the cell has the value passed to that neighbour denoted by the letter directly next to the 'X' on the cell boundary. This value can be N, E, S, W, or F, representing the input from the neighbour to the north, south, east, west or the output of the function respectively.

3.3.6 Test suite

The apparatus to perform tests takes the form of a while loop encapsulating the evolutionary process, reading performance data and sculpting parameters as and when required (reseeding the population or changing weightings, for example). The bounds of the tests being executed are defined in the headerfile `evolve.h` (Appendix B). `TEST_SIZE` defines the number of samples to be generated, and therefore the number of times the test should be repeated; `TEST_LOOP` defines the size of the inner evolutionary while loop, and therefore the number of generations each test should be allowed to run for. During execution, information about the best performing individuals, execution time, and population averages are collected and averaged over each run. This gathered information is then written to two files `test.dat`, which involves the generation-by-generation fitness averages, and `summary.txt` which details the number of tests which generated a perfect answer, the average fitness by the end of execution, the average execution time, and the variance in final accuracies.

Chapter 4

Critical Evaluation

Using the automated testing suite to repeatedly run experiments, the data harvested for the next section was generated by averaging over 30 independent evolution runs each for 15000 generations. Each test was aimed at evolving circuitry capable of 2-bit addition (and later 2-bit addition and subtraction). Evaluation was performed as outlined in the previous chapter; for each of the 16 tests required for 2-bit addition the binary representation of each number was made available to the FPGA, and after evaluating the circuitry 3 binary values are read off the bottom of the grid, for each correct bit the configuration scored 1 point. For both 2-bit addition and weighted 2-bit addition and subtraction the maximum score is 48. Accuracy will be presented as a percentage compared to a perfect circuit. All solutions exist as FPGA configurations on an FPGA with 4x4 CLBs. An ideal hand-designed benchmark circuit is outlined in Figure 4.12(b) and requires less than half of the FPGA resources made available to it.

Much of the published work in genetic algorithms frames maximising the fitness function as the aim of genetic algorithm parameter choices. Here most of the data harvested pertains to the accuracy of the configuration; the number of correct bits read off the FPGA for each test given as a percentage. This is a measure of how good the configuration is, throughout execution the fitness function simply acts as an incentive to improve this value.

All experiments were conducted on a late 2013 15" Apple MacBook Pro with an Intel Haswell Core i7 processor clocked at 2.3 GHz and 16GB of RAM.

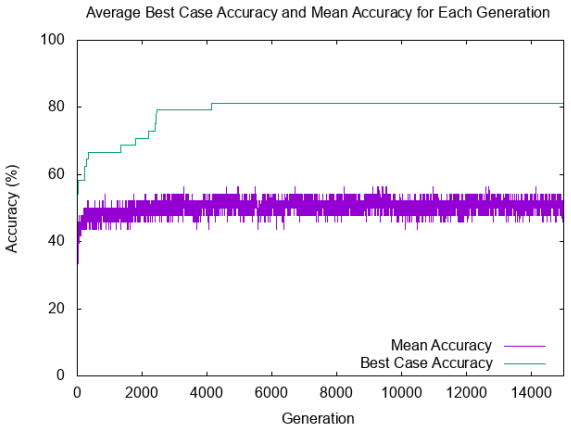
4.1 Genetic Algorithm Parameter Tuning

All initial genetic algorithm parameters were set mirroring the parameter choice of [42]; a population size of 50, with elitism, linear rank-based selection, the probability of single point crossover occurring set to 0.7, and the expected mutations per offspring set as 2.7. However the problem solved in [42] is dramatically different from the addition problem posed here so a great deal of domain specific tuning is required. This was chosen as the origin of parameter exploration as it is the paper in the existing literature which has taken the greatest pains to explain the experiment configuration and setup.

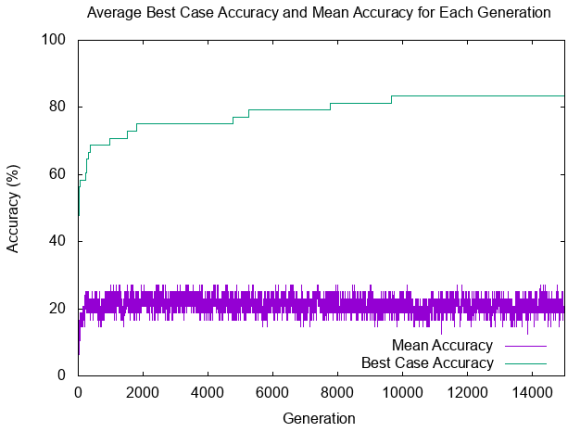
These parameters were evaluated and, as shown in Figure 4.1(a), the results of these initial tests are promising but show a great deal of room for improvement. Despite a healthy initial improvement in fitness, this quickly reached a plateau and across the duration of the tests no evolutionary run was capable of formulating a perfect solution. Visual inspection of this initial test as it ran highlighted a huge lack of diversity in the population of solutions, every improvement in fitness was an iterative change in the previous best case configuration. This occurs when a solution improves beyond it's peers and the boost to fertility leads to it dominating the population. Often a dominating solution will be mostly correct but due to wildly uneconomic uses of resources be completely incapable of evolving any further without dramatic backtracking. The user interface displays the current average diversity; the distance from one individual to each other in the population. Relatively early in execution when one member begins to dominate the diversity plummets. This information was discerned from the GUI during execution.

4.1.1 Improving Diversity

Such evolutionary dead ends were also faced by [6] who developed a method of employing multi-objective fitness to encourage population diversity and minimise waste in solutions. Using their multi-objective fitness function, a linear combination of diversity and size, to improve the population diversity and



(a)



(b)

	Fitness Function	Perf. Runs (%)	Avg. Exec. Time (s)	Avg. Best Accuracy (%)
(a)	Simple	0	235	81
(b)	Multi-Objective	0	267	83

Figure 4.1: Fitness function test results; population size 50, elitism, linear rank-based selection, single point crossover probability 0.7, mutation rate 2.7 and (a) a simple fitness function mirroring [42] in which the fitness is the accuracy, (b) an extended multi-objective fitness function incorporating a measure of diversity with the diversity weight set to 40% of accuracy.

reduce the probability of bloated solutions could provide the evolutionary pressure required to find a perfect solution. In this situation, the final size of a configuration is inconsequential, the size is bound to 4x4 and although a solution which underutilises the FPGA is ideal, such stretch goals are beyond the scope of this document.

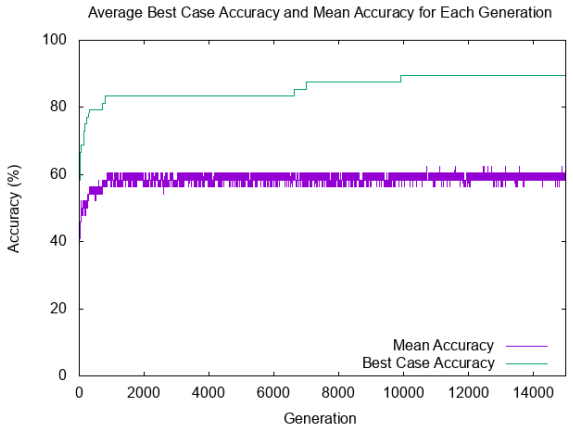
A trial was conducted incorporating a diversity measurement weighted at 40% of the accuracy weighting. This value was arrived at so that diversity, which exists in the rough range of 0-85, was weighted slightly less than accuracy, which reaches a maximum of 48.

Figure 4.1(b) depicts the results from this experiment. Averaged across repeated executions, there was a slight increase in execution time accompanied by a marginal improvement in the best case solution fitness after 15000 generations. However, a larger number of generations were required to match the accuracy of the simpler fitness function. Despite predictions to the contrary, another symptom of the more complex fitness function in this context is a reduced mean population accuracy, instead of easily beating 20 as happens in Figure 4.1(a), 4.1(b) maintains a disappointing average of 10. This is due to a shift in evolutionary emphasis, in the new scheme some individuals are given a large fitness due to their novelty alone and therefore maintained despite their low accuracy, which brings down the population average. This shift in emphasis also slows the pursuit of perfection; the initial trial in unencumbered with a fitness function attempting to maintain diversity, and so aggressively pursues improvements in fitness due to a larger proportion of the population which are offspring of a dominant individual. The slower intial gains in the second trial are indicative of the smaller segment of the population actively working on the current best configuration.

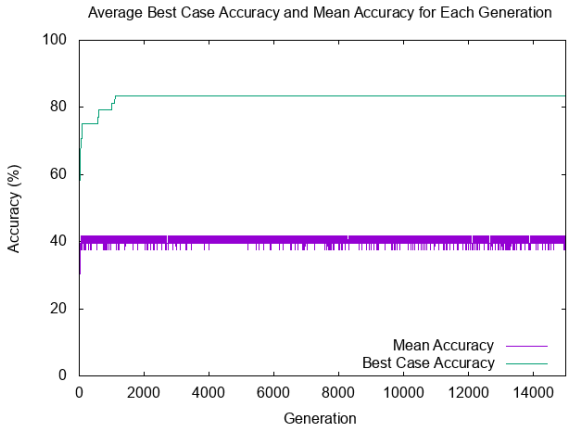
These findings disagree somewhat with [6], who found vast improvements across the board when extending the fitness function. However they did not experiment with population sizes as small as presented here; unlike our population size of 50, DeJong et al. used a population size of 1000. It could well be the case that the population size is too small to facilitate a series of semi-independently evolving subpopulations. They also had a mechanism which pruned the population of "dominated" individuals which would reduce the tendency for a population driven by a fitness function which incorporates diversity to maintain poorly performing individuals.

To explore this further an experiment with 2 variables was conducted, varying both the population size (50 and 500) and the incorporation (or not) of a multi-objective fitness function combining diversity and correctness. A maximum population size of 500 was used as a size matching that of DeJong et al. would not feasible in the context, as individual runs would take in excess of 2 hours.

Figure 4.2 contains the results from this experiment. The larger population size with the simple



(a)



(b)

	Fitness Function	Perf. Runs (%)	Avg. Exec. Time (s)	Avg. Best Accuracy (%)
(a)	Simple	0	2250	90
(b)	Multi-Objective	0	2857	83

Figure 4.2: Large population diversity trail; population size 500, elitism, linear rank-based selection, single point crossover probability 0.7, mutation rate 2.7 and (a) a simple fitness function mirroring [42], (b) an extended multi-objective fitness function incorporating a measure of diversity with the diversity weight set to 40% of accuracy.

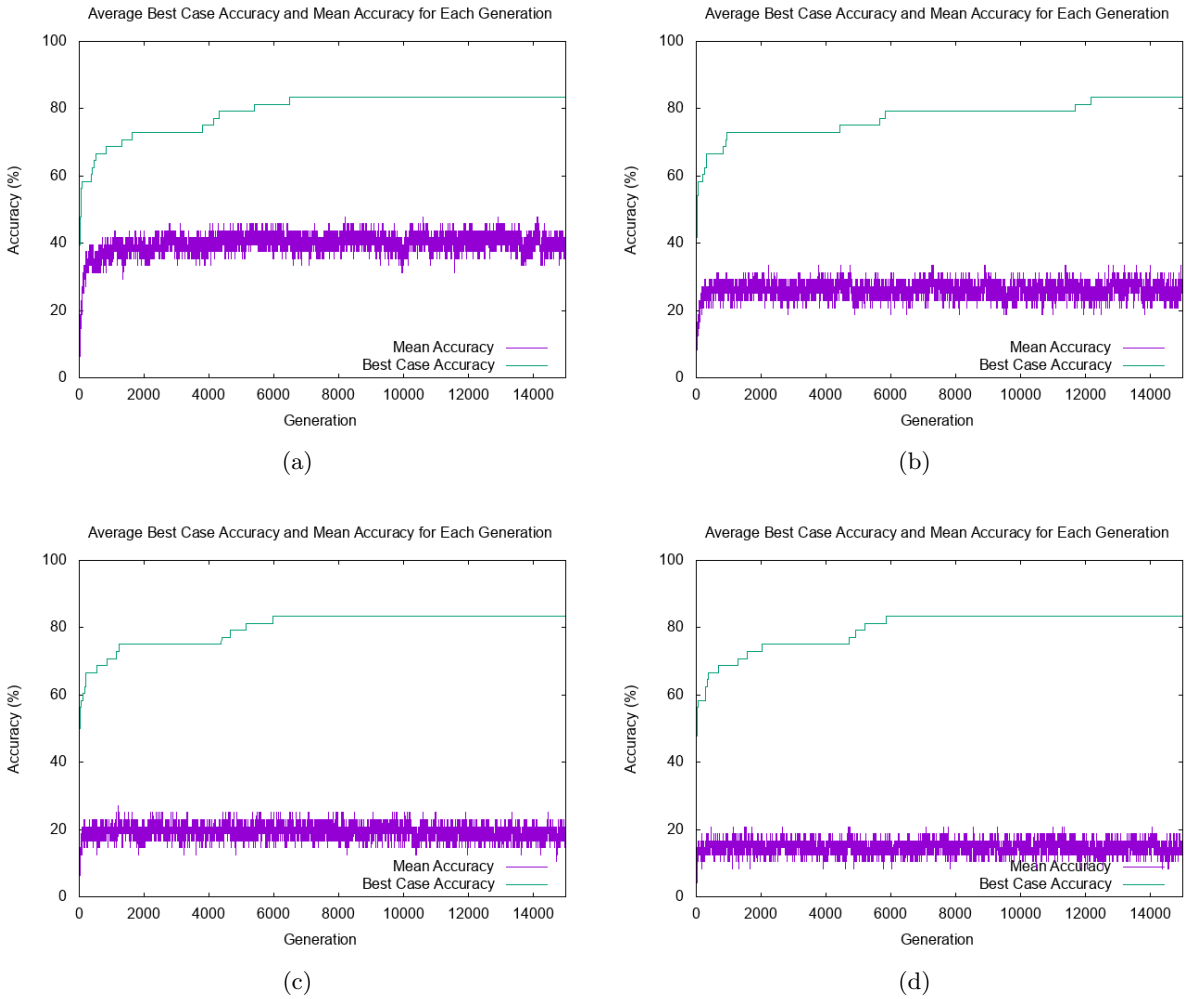
fitness function (Figure 4.2(a)) performed the best, the streamlined persuit of accuracy combined with a larger pool of configurations to draw from resulted in the best performance thus far. The extended function (Figure 4.2(b)) actually reduced the performance, again disagreeing with [6], and highlighting the effectiveness of their pruning strategy. This result could be because the larger population has natural diversity qualities, with a larger pool a probabilistic procedure such as evolution naturally produces variance, and the extended function distracted too much from the persuit of accuracy, and requires a pruning strategy to reduce waste exploration. The hypothesis that increasing the population size minimises the reduction in mean accuracy does ring true. With the smaller population the extended fitness function resulted in a mean accuracy of less than half the mean accuracy with the simpler fitness function, whereas with a larger population the extended function's trail showed a mean accuracy of more than 2/3rds that of the simpler one.

Despite the improved performance in Figure 4.2(a), moving forward further tuning will be performed on the parameters which produced Figure 4.1(b). The larger population size in the higher performing experiments produced an execution time which does not lend itself to exploration and iterative improvement. Population size will, however, be further explored in subsection 4.1.6.

4.1.2 Mutation Rate

There are two main factors influencing the choice in mutation rate; the fragility of solutions, and the space between viable individuals. If solutions to a problem are inherently fragile, as has been demonstrated in this case by Figure 3.3, then a high mutation rate can regularly destroy working solutions and cause the system to rarely rise above a fixed value. If the space between viable solutions is vast a mutation rate which is too small will, with low probability cause enough mutations to span the ravine, yet alone create them in correct places. These two factors are often at odds, no more so than in this context, where solutions are brittle and sparse. Elitism is employed to reduce the impact of fragile solutions by ensuring the best case solution is never lost, and the inclusion of the extended diversity fitness function encourages individuals to drift across ravines rather than have to make the journey in a single step. Therefore the impact of shifting the fitness function is unclear.

The experiments conducted in Figure 4.3 mostly agree with the findings in [42] with respect to the experimental results indicating and ideal mutation rate of around 2.7. However a slight accuracy increase in the best case individual is observed when shifting the mutation rate from 2.7 (Figure 4.1(b)) to 3 (Figure 4.3(c)). This is another example of domain specific experimental tuning for the binary arithmetic



	Mutation Rate	Perf. Runs (%)	Avg. Exec. Time (s)	Avg. Best Accuracy
(a)	1	0	245	83
(b)	2	0	264	83
Figure 4.1(b)	2.7	0	267	83
(c)	3	0	363	83
(d)	4	0	368	83

Figure 4.3: Mutation rate test results; population size 50, elitism, multi-objective fitness function with diversity weighting 40% of the accuracy, linear rank-based selection, single point crossover probability 0.7, and (a) 1 expected mutation per individual, (b) 2 expected mutations per individual, (c) 3 expected mutations per individual, and (d) 4 expected mutations per individual.

problem. The drastic reduction in the speed with which the best case individual is evolved occuring with a mutation rate of 2 (Figure 4.3(b)) is unexpected. This result could be due to the underlying topography of the solution landscape; viable solutions are rarely 2 changes apart, and are more frequently distanced at 1, 3, or 4 changes. This would explain this marginal reduction in evolutionary swiftness. Note the continuous decrease in population mean accuracy as the mutation rate is increased, the reasons are twofold; firstly, a higher mutation rate is more likely to induce a change in an accuracy-critical component of the solution. Secondly, with an extended fitness function maintaining solutions for novelty value alone, the more aggressive mutation rate is more likely to cast members of the population into even further unexplored corners of the fitness landscape giving them an even higher diversity score (and therefore improving their fitness and their staying power), these solutions can survive without a high accuracy score.

Because of the results demonstrated by Figure 4.3(c), namely slightly improved best-case evolution (at the cost of population accuracy), an expected mutation rate of 3 has been chosen. This was selected over the mutation rate 1 as the higher population accuracy is a symptom of a more localised less varied population which is cultivated by a gentler mutation rate.

4.1.3 Selection Mechanisms

The most popular selection methods in evolvable hardware, by a considerable margin, are rank selection and tournament selection. Thus far we have been using linear rank selection (mirroring [42]). Here the use of a variable skewed rank selection is employed to place higher evolutionary pressure on the better performing individuals. The variable controlling the linearity of the selection function can be set such that the probability of selection grows quadratically rather than linearly as you move up the rankings. To this end experiments setting this value to 0 (a quadratic selection function with no linear component), 0.5, and 1 (completely linear) will highlight any impact this has on the system.

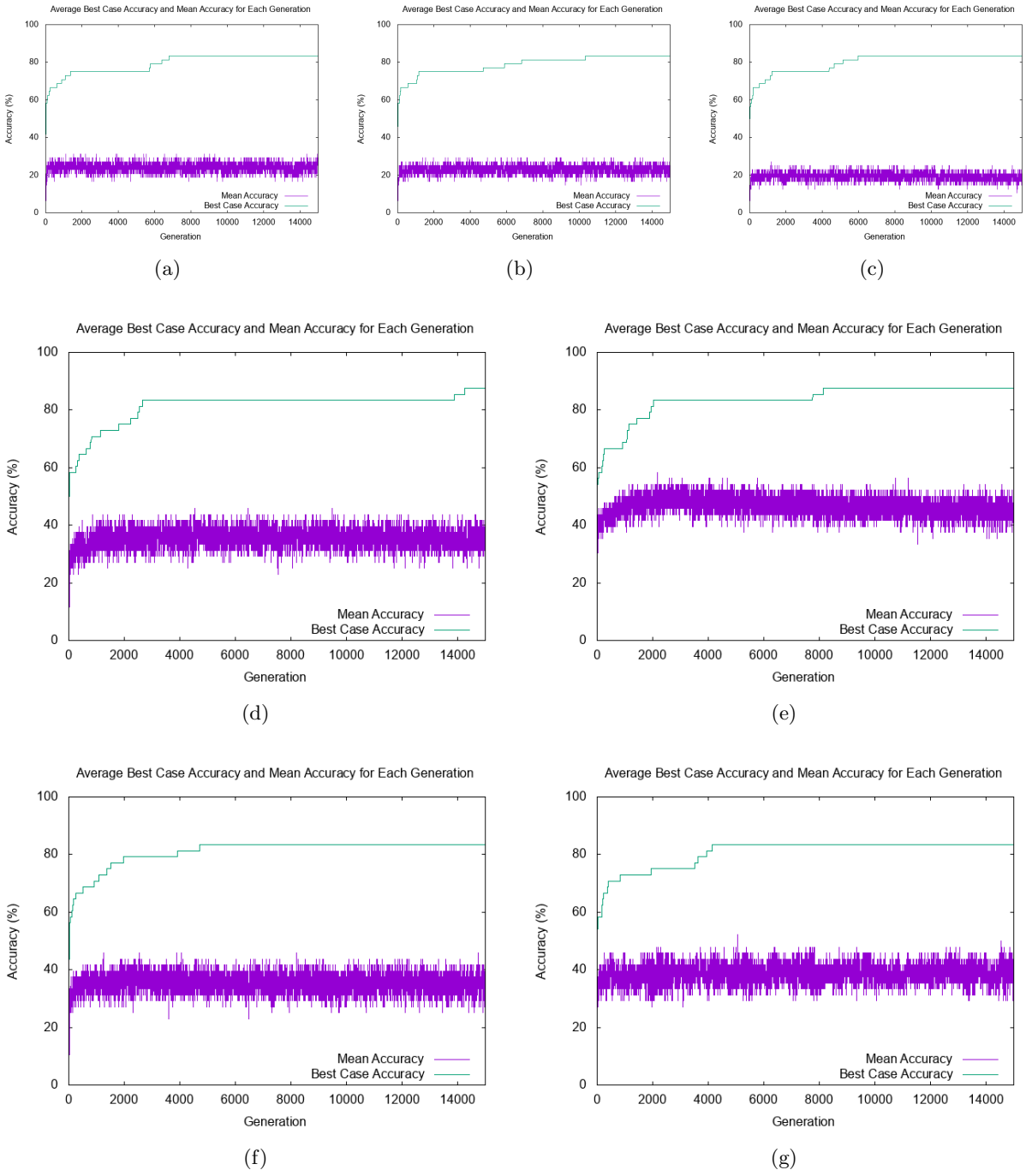
Along with this variation, an experiment comparing current best selection with tournament selection of varying sizes from 10 (20% the population size) up to 40 (80% the population size). The smaller tournament size will discriminate less against lower performing individuals, and the larger the tournament the fewer poorly performing configurations will be selected for the next generation. This interplay could dramatically increase performance, exploiting a balance to drastically improve performance. The performance improvement in this context was noted by [24].

The more aggressive discrimination against members of the population due to the variable skew is clear in Figure 4.4. The population under the more aggressive selection mechanism (Figure 4.4(a)) pursue the maximum accuracy somewhat quicker than the more linear rank selections (Figure 4.4(b) and Figure 4.4(c)), however this more aggressive discrimination frequently directs the population with something approaching single-minded determination, often to it's detriment. The average final fitness is clearly lower with a quadratic rank than with a linear rank.

Tournament selection with a tournament size of 20 is a clear improvement over any other selection methods thus far, and has had the single largest impact on the number of perfect solutions evolved of any parameter choice until now. 13% of the trails ended in a perfect solution with an accuracy score of 48. Individuals are chosen uniformly at random for a tournament, but within a tournament only the most fit individual wins. This interplay between minimally and maximally discriminatory selection schemes results in heavily directed evolutionary search which maintains a very diverse population. Each of the tournament schemes have a high population accuracy with a wider variance than any of the rank selection schemes, indicating the diversity maintenance. The smaller tournament size, Figure 4.4(d) is too random and does not put enough evolutionary pressure on finding an accurate solution. Despite maintaining a broad population the fitness is lower due to this lack of pressure. The larger tournament size Figure 4.4(g) discriminates too aggressively, in any selection round the weakest 80% of the population cannot be selected (as they are guaranteed to lose the tournament), this leads to a dominated and ultimately weaker population. Balancing these two facets of tournament selection, a tournament size of 20 allows for a search procedure directed enough to find optimal solutions and with enough random influence to maintain a wide population. This agrees with the findings of [24].

4.1.4 Crossover

Crossover works well in this context, the nature of the mapping between genotype and phenotype means any crossover can provide benefits to both mating pairs and inject some much needed diversity into the ecosystem. Three additional trial runs were executed, each at a different crossover probability rate; 0,



	Selection Method	Perf. Runs (%)	Avg. Execution Time (s)	Avg. Best Accuracy (%)
(a)	Rank (Skew 0.0)	0	304	83
(b)	Rank (Skew 0.5)	0	415	83
(c)	Rank (Skew 1.0)	0	363	83
(d)	Tournament (Size 10)	0	264	88
(e)	Tournament (Size 20)	13	231	88
(f)	Tournament (Size 30)	3	260	83
(g)	Tournament (Size 40)	0	244	83

Figure 4.4: Selection method test results; population size 50, elitism, multi-objective fitness function with diversity weighting 40% of the accuracy, single point crossover probability 0.7, mutation rate 3 and (a)(b)(c)[1] rank based selection with skew 0.0, 0.5, and 1.0 respectively , and (d)(e)(f)(g) using tournament selection with tournament size 10, 20, 30, and 40 respectively.

[1](c) duplicated from Figure 4.3(c) for better direct visual comparison

0.5, and 1. Crossover points only occur between bytes, never inside them; so when genetic material is transplanted it is always clean and individual cell operation is consistent before and after.

In Figure 4.5, none of the trails produced results rivaling the initial crossover probability of 0.7. This behaviour could be a result of how crossover influences the underlying population structure; never performing crossover and usually performing crossover performed better than always performing crossover and performing or not with an equal probability. An explanation for the performance at 0 probability could be that when the population develops without the threat of crossover there is an implicit encouragement to diversify as there is no punishment for creating a novel solution which crossover would destroy by mating with an incompatible partner. Poor performance at equal probabilities could be a symptom of the same behaviour; when crossover may or may not happen with equal probabilities a population cannot adapt either way. The poor performance at 100% probability would seem to disagree with this, unless you consider that crossover is ultimately a disruptive force. When crossover is guaranteed the population can only develop solutions durable in the face of random string swapping. 0.7 provides a good balance where a population can develop with the assumption of crossover but there is a chance solutions can develop which rely on crossover not happening.

4.1.5 Elitism

With a mutation rate as high as shown here the effect of elitism is heightened. The probability of a mutation crippling a design is relatively high and so a population without elitism is prone to climbing the evolutionary ladder just to be cast down by luck. Binary arithmetic is especially fragile, as demonstrated by Figure 4.6, where the evolutionary parameters are consistent with our highest performer so far, except elitism is turned off.

Clearly, as Figure 4.6 demonstrates removing elitism is crippling. The population rarely, if ever, performs better than an accuracy of 28. From here any path to a better viable solution is insurmountable, crippled by an aggressive mutation rate, which explored the search space with such vigor that no individuals performing well can survive due to their inherent fragility. This disagrees with [41] which claims evolution naturally discovers solutions resistant in the face of change; for binary arithmetic the solutions are too fragile to have such a resistance and require the support of elitism to prop up the best individuals. Resulting in solutions which do not have the same durability as a solution found without elitism, if one could ever be found.

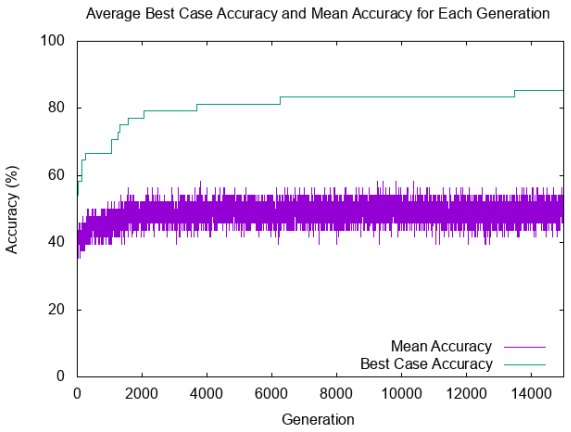
4.1.6 Tuning Population Size

With the desire to cultivate a diverse population at the forefront of the parameter selection reasoning enlarging the population would improve the spread of solutions, and allow the evolutionary pressure on diversity to simultaneously mature and develop a breadth of solutions. Obviously increasing the size of the population has objective benefits in a statistical sense, by casting the net wider there is a higher chance to stumble across the correct solution. But, this has diminishing returns; due to the diversity measurement incorporated into the fitness function the evalutation time grows with $O(n^2)$ complexity with the population size.

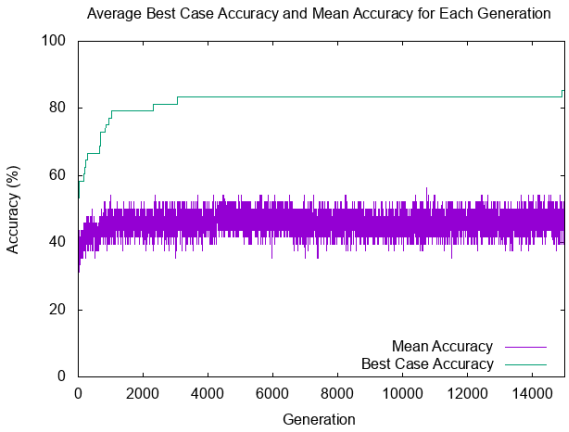
The increased population size from the experiments with a multi-objective fitness function could be beyond the point of diminishing returns. To explore this further trials varying the population size to explore how execution time and genetic algorithm performance is affected. Population sizes of 25, 50, 100, and 200 were selected to span the bredth of potential choices. For each the tournament size scaled accordingly, constantly set to 40% of the whole population (10, 20, 40, and 80 respectively).

Unsurprisingly the larger population has the highest average final fitness, and proportion of evolutionary runs ending in a perfect solution but Figure 4.7 highlights an interesting revelation; Despite the evaluation function scaling with complexity $O(n^2)$ the increase in execution time is almost linear with the population. This indicates the poorly scaling portion of the evaluation function measuring population diversity counts for an insignificant minority of the execution time. The most aggressive increases in time are due to the additional population members which need to be instantiated as FPGAs and fed the test data. This evidence against the claims made in Subsection 3.2.6 which suggested that the multi-objective fitness function would cause a scaling issue with a growing population and demonstrates that the problems are localised to FPGA evaluation.

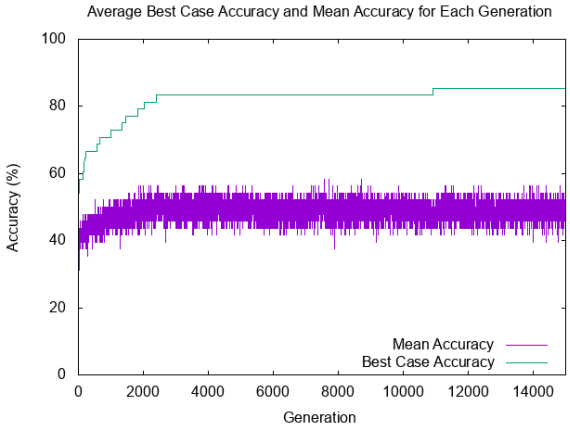
For all population sizes achieving a best-case accuracy of 36 occurred in a similar amount of time, however the larger populations were able to maintain this ascent up the evolutionary ladder better. This sudden insurmountable evolutionary barrier for the smaller population sizes implies that the smaller



(a)



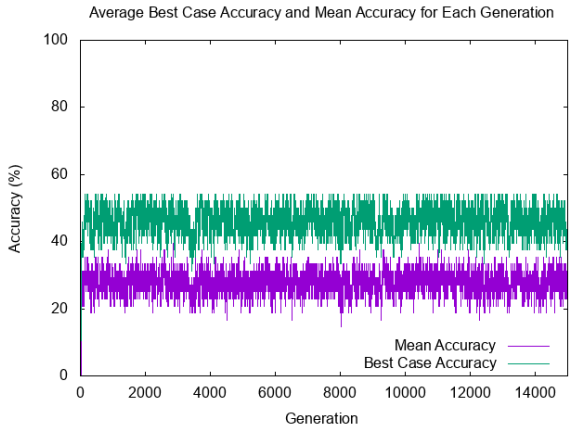
(b)



(c)

	Crossover Prob.	Perf. Runs (%)	Avg. Exec. Time (s)	Avg. Best Accuracy (%)
(a)	0.0	10	243	85
(b)	0.5	3	245	85
Figure 4.4(e)	0.7	13	231	88
(c)	1.0	3	249	85

Figure 4.5: Crossover probability experiment; population size 50, elitism, multi-objective fitness function with diversity weighting 40% of the accuracy, mutation rate 3, tournament selection of size 20, and probability of single point crossover set to (a) 0.0, (b) 0.5, and (c) 1.0.



Perfect Solutions (%)	Avg. Execution Time (s)	Avg. Best Accuracy (%)
0	281	40

Figure 4.6: Removing elitism test results; population size 50, no elitism, multi-objective fitness function with diversity weighting 40% of the accuracy, mutation rate 3, tournament selection of size 20, and probability of single point crossover set to 0.7.

populations were unable to cultivate a bredth of solutions early in execution and therefore encountered an evolutionary dead end, temporarily halting improvements.

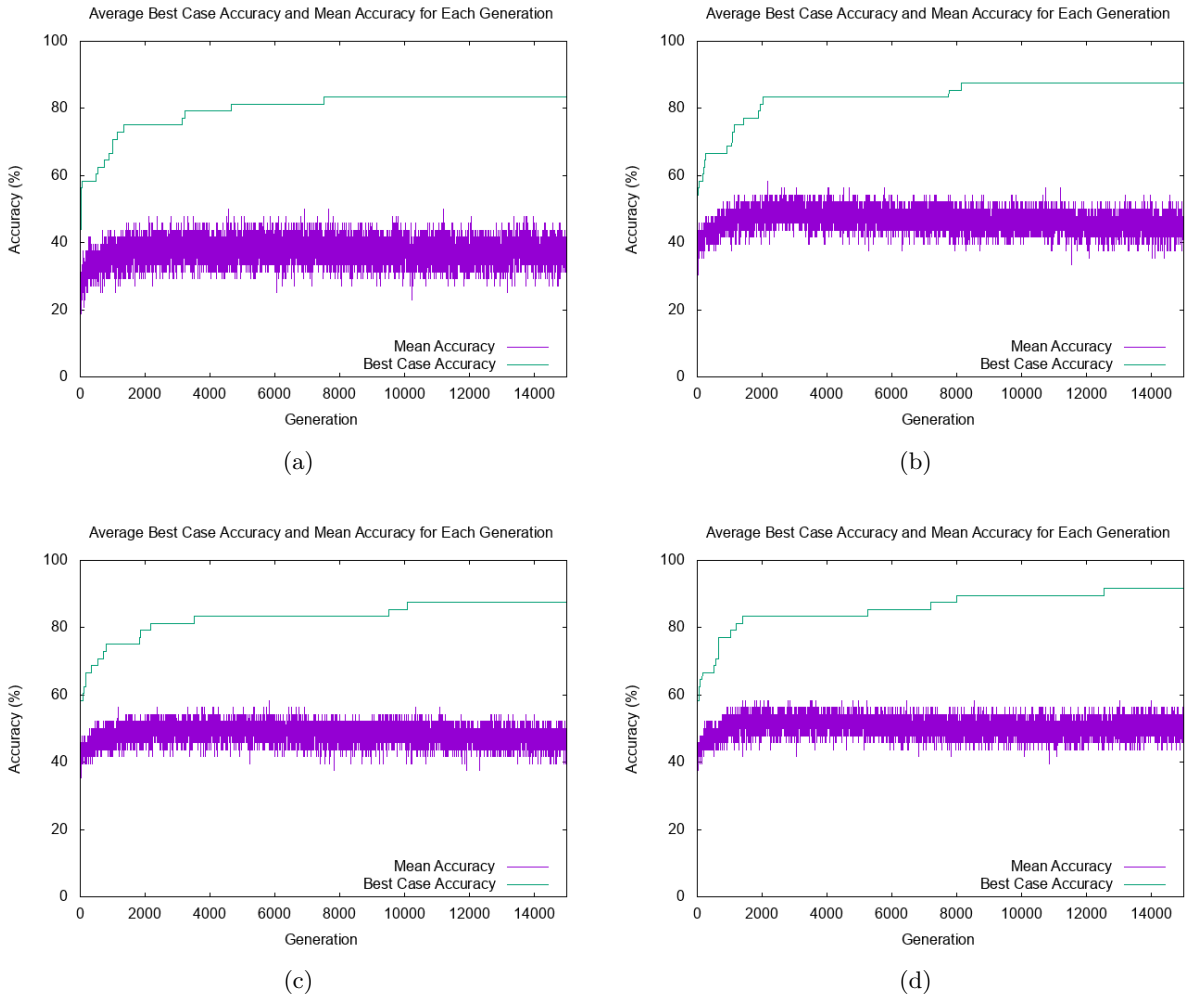
Moving forward a population size of 50 was selected. Clearly the population size of 200 is the best candidate, but the increase in percentage of perfect solutions is almost linear with the added execution time. In order to maintain momentum at this stage in the project the quicker execution was chosen, with the knowledge that one could quadruple the population to increase the execution time and percentage of perfect solutions by a factor of 4.

4.1.7 Tuning Diversity

One aspect of the evolutionary process hitherto unexplored is the relative weighting given to the diversity measurement in the fitness function. If set too high the genetic algorithm begins directing for novelty alone, accuracy is obscured and diversity reigns supreme. If set too low we re-enter the domain of single-minded persuit of accuracy, one which frequently falls into evolutionary dead ends gracelessly halting progress. Diversity weighting is defined here as a fraction of the weighting given to accuracy. Accuracy exists in the range 0-48 for the 2-bit binary arithmetic problem, and from experimental results diversity usual reaches a maximum value of around 85. So for any diversity weighting larger than half the size of the accuracy weighting an individual does well to be maximally diverse, accuracy occuring as an afterthought. Armed with this knowledge an experiment was framed running evolutionary trails at a number of key diversity weightings, 0.0, 0.2, 0.4, and 0.6 accuracy weight. This shifts evolutionary pressure from completely focussed on accuracy to mostly completely focussed on diversity.

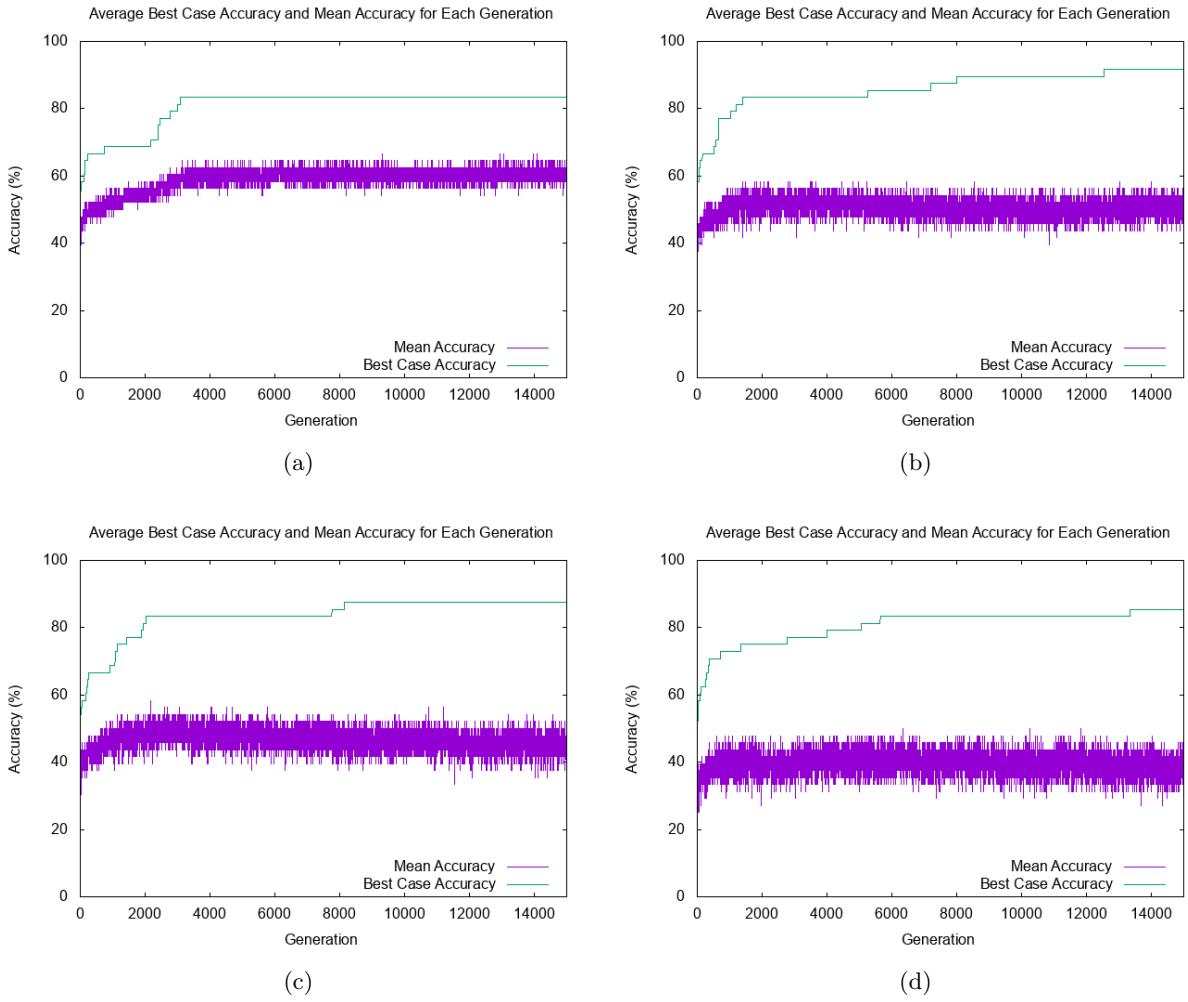
Figure 4.8 demonstrates the symptoms indicating existing on either side of balance of power within the fitness function between accuracy and diversity. With a weighting of 0 (Figure 4.8(a)) the population hits two evolutionary dead ends, one at accuracy 71%, which it eventually manages to surmount and another at accuracy 83% which it fails to improve beyond. The lack of diversity is also clear in the narrow variance in mean population accuracy. On the other end of the spectrum a weighting of 60% (Figure 4.8(d)) achieved worse perfect solutions but was able to reach an average ending accuracy of 85%, and all improvements to fitness occurred relatively regularly. This indicates a diverse population with only a small incentive to improve their accuracy. Balancing this, and improving on the previous best, decreasing the weighting from 40% to 20% seemed to better sit between the two extremes presented by Figure 4.8(a) and Figure 4.8(d).

The reduced effectiveness of diversity at higher weightings is somewhat at odds with the point of view presented by [6]. As previously mentioned however their implementation was coupled with an effective pruning mechanism to dissuade mindless diversification for the sake of diversification. Also the context



	Population Size	Perfect Solutions (%)	Avg. Execution Time (s)	Avg. Best Accuracy (%)
(a)	25	0	126	83
(b)	50	13	231	88
(c)	100	17	457	88
(d)	200	57	955	92

Figure 4.7: Population tuning test results; elitism, mutli-objective fitness function with diversity weighting 40% of the accuracy, mutation rate 3, tournament selection of size 40% the population size, probability of single point crossover 0.7, and (a) population size 25, (b) population size 50, (c) population size 100, and (d) population size 200.



	Diversity Weighting (% of Accuracy)	Perf. Runs (%)	Avg. Execution Time (s)	Avg. Best Accuracy (%)
(a)	0	16	235	83
(b)	20	23	238	94
(c)	40	13	231	88
(d)	60	10	247	85

Figure 4.8: Diversity weighting test results; population size 50, elitism, tournament selection of size 20, probability of single point crossover 0.7, and a multi-objective fitness function with diversity weighting set to (a) 0%, (b) 20%, (c) 40%, and (d) 60% the weighting associated with accuracy.

each multi-objective fitness function differs, evolutionary hardware and evolutionary programming are different animals each with distinct problems. Namely, [6] present theirs as a mechanism for avoiding the run-away bloat that can happen with evolutionary programming where the program needlessly gets longer, whereas here we are spatially capped at the size of the FPGA, looking for a solution within it's confines. It is a subtle difference but could explain the disparity in results.

4.1.8 Coevolution

Can we exchange the full problem testing system used thus far for a coevolved parasite population of evolutionarily targeted problems? In order to answer that question experiments employing coevolution were conducted on the evolutionary system developed thus far. Experiments were conducted with a parasite problem size of 8, 16, and 32; and the best performing parameter choice was repeated without elitism.

In a coevolutionary system there a series of problems that can disturb the gentle balance between host and parasite. In Figure 4.9 no coevolutionary system was capable of evolving a successful solution. Each had poor execution time and low final fitness.

Earlier in this document coevolution was touted as a potential solution to the scaling problem, however comparing Figure 4.9(b) and Figure 4.9(d) it is clear that without elitism the entire system becomes unable to produce a solution with even somewhat acceptable accuracy. This reiterates the findings in subsection 4.1.5 in this new context. In order for elitism to have any real impact with the binary addition problem members of the population have to be exhaustively tested against all possible problems to allow the elitism metric to keep the best performing individual regardless of the parasite they are paired against. This disrupts the notion that coevolution can be used to aid with the scaling issue and will be further discussed in Section 4.4. Somewhat surprising is the ability of coevolution with a parasite which only contains 8 problems (only 4 of which are addition problems) to provide a semblance of positive evolutionary pressure.

In all the experiments with elitism activated after a certain amount of time the average population fitness reaches a plateau and all improvements seem to come from the highest performing individual(s) with little impact in the overall population. Although similar to tests not using a coevolutionary system this behaviour could be indicative of a disengaged population, one in which the parasite population becomes too aggressive and becomes unbeatable, no longer discriminating in the host population between FPGA configurations.

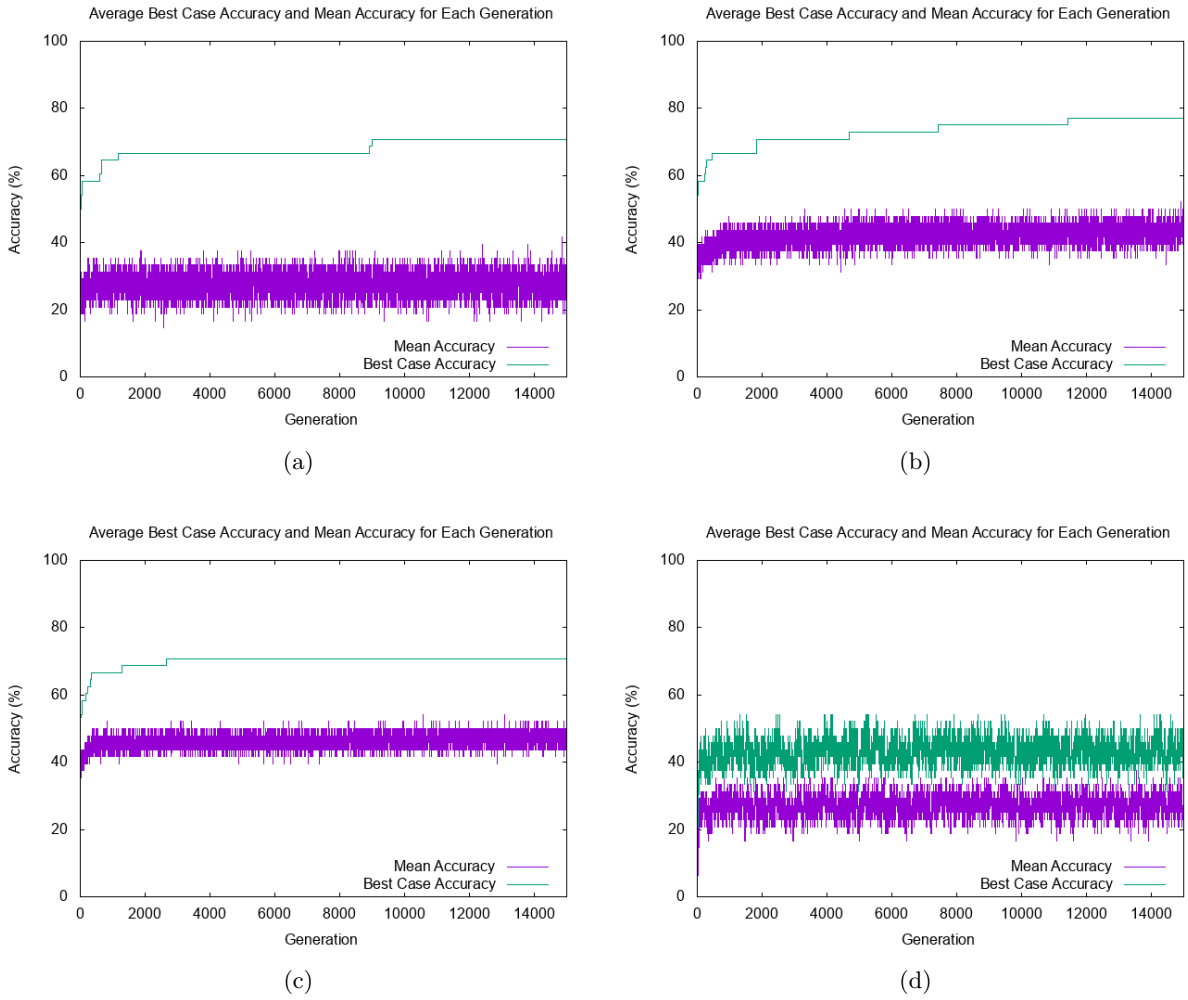
The evolvable hardware platform developed in parallel to this thesis is capable of modifying the parasite virulence but such exploration goes beyond the scope of this work.

4.1.9 Evolved 2-bit Addition Hardware

By now we have a relatively well performing evolutionary hardware system, tailored directly for the binary arithmetic problem. The tuned parameters are; population size 50, elitism, tournament selection of size 20, probability of single point crossover 0.7, and a multi-objective fitness function with a diversity weight set to 20% the accuracy weight. This can be thought of as a modernised, experimentally tuned version of the parameters used in [42]. With this configuration the evolutionary process takes, on average 238 seconds to span 15000 generations and 23% of executions conclude with a perfect solution to the 2-bit binary addition problem. One such perfect FPGA configuration is outlined in Figure 4.10(a), with all superfluous components not contributing to the answer manually pruned away. The graph detailing the average accuracy over time for this parameter choice is Figure 4.8(b). This novel solution calculates b_0 in the top right hand corner, and c in the bottom left, this could well be because crossover encouraged this sort of division of space and labour. The circuity calculating b_1 is spread throughout the device, highlighting the usefulness of crossover not being a certainty.

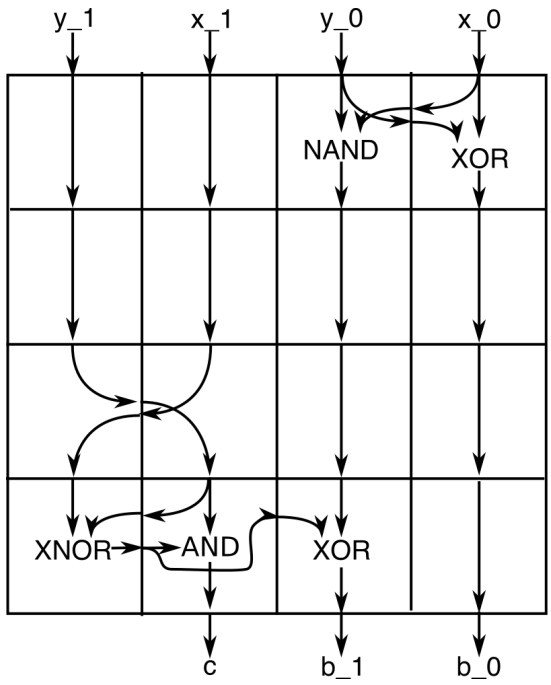
The histogram in Figure 4.10(b) shows the distribution of final accuracies during each of the 30 evolution runs, the maximum score is 48. The genetic algorithm clearly has a strong tendancy to create high performing solutions. The huge number of executions that finish with an accuracy of 44, however, indicates that it still suffers with dead end solutions. These solutions were clearly high performing, achieving an accuracy of 44, but they must have overutilised the FPGA and allowed no room for improvement. It is interesting to note that stumbling on and getting stuck on an accuracy of 44 is quite common. There could be a very stable local optimum amongst configurations achieving this score from which any further exploration struggles to make improvements.

Even from the initial experiments this particular failure case occurred frequently enough to be notable, the screenshot in Figure 4.11 was taken from one of these early trials. More research into why

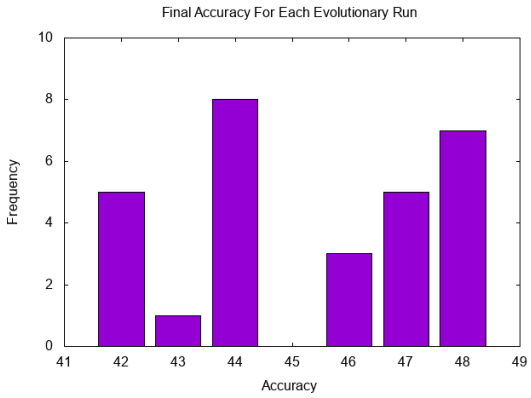


	Parasite Size	Perf. Runs (%)	Avg. Execution Time (s)	Avg. Best Accuracy (%)
(a)	8	0	740	71
(b)	16	0	605	77
(c)	32	0	703	71
(d)	16	0	535	50

Figure 4.9: Coevolution test results; population size 50, elitism, tournament selection of size 20, probability of single point crossover 0.7, a multi-objective fitness function with diversity weighting set to 20%, and a coevolved parasite population where each parasite is of size (a) 8, (b) 16, and (c) 32 tests. (d) has a parasite consisting of 16 tests but elitism is turned off.



(a)



(b)

Figure 4.10: Successful evolved solution for the 2-bit binary addition problem; (a) FPGA configuration scoring a perfect accuracy, (b) histogram of the final accuracies across each of the 30 evolution runs.

```
0 + 0 : 2/3 bits correct
1 + 0 : 2/3 bits correct
2 + 0 : 3/3 bits correct
3 + 0 : 3/3 bits correct
0 + 1 : 2/3 bits correct
1 + 1 : 3/3 bits correct
2 + 1 : 3/3 bits correct
3 + 1 : 3/3 bits correct
0 + 2 : 3/3 bits correct
1 + 2 : 3/3 bits correct
2 + 2 : 3/3 bits correct
3 + 2 : 3/3 bits correct
0 + 3 : 3/3 bits correct
1 + 3 : 3/3 bits correct
2 + 3 : 3/3 bits correct
3 + 3 : 2/3 bits correct
```

Figure 4.11: Common failure cases

configurations with an accuracy of 44, with these failure cases tend to be both easy to arrive at and hard to escape from is required to thoroughly understand this issue. There are few similarities between the failure cases, save for the fact that they are all only incorrect on one bit. The next step would be isolating the incorrect bit in each case to try and discern a pattern.

The configuration detailed in Figure 4.10(a) is clearly of novel design. Crossing over x_1 and y_1 before the XNOR and AND functions, only to have the XNOR result pass back through the cell performing the AND operation, for example. Such a design decision would probably never have been made by a human designer, but they can be produced by genetic algorithms because in this situation there is no evolutionary pressure to create a design which is intuitive.

A two sample t-test was used to discern if the parameter changes to the genetic algorithm actually had a substantial performance impact. Comparing the final algorithm (Figure 4.8(b), mean final accuracy 45, with variance 5.6) to the initial configuration (Figure 4.1(a), mean final accuracy 39, with variance 7.4) we get a t-value of 9.11. With 29 degrees of freedom (as the sample size for each is 30), and probability p=0.01 (the probability that the two distributions are actually from one population) our t-value has to be larger than 2.76 to be deemed significant, which it is. Therefore the changes arrived at in this section offer a significant performance improvement over the original parameter choices in the 2-bit binary addition problem.

With this platform we can explore a range of uses within the domain of dynamic problems.

4.2 Fault Tolerance

The dynamic problem with the largest immediate impact to consumer and industry alike is arguably that of a system experiencing faulty behaviour. With a tuned genetic algorithm, exploration into the capacity to dynamically adapt the configuration in the face of such a changed or changing problem provides the opportunity to thoroughly test evolutionary driven fault recovery and mitigation strategies.

Using the fault injection framework built into the FPGA simulation, a series of targeted or randomly generated faults will be introduced to an in-progress genetic algorithm.

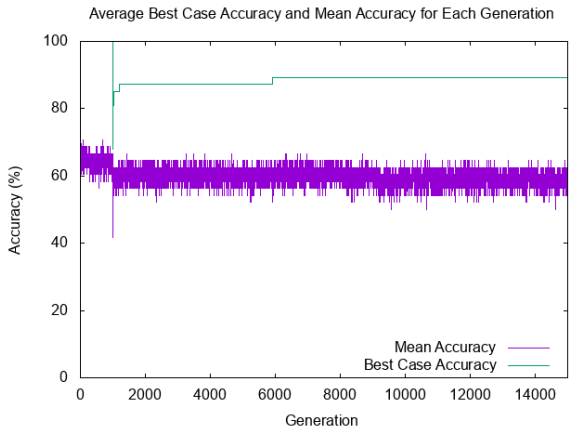
4.2.1 Simple Fault Recovery

By preloading the genetic algorithm with a hand designed 2-bit adder and then simulating a highly targeted critical fault within the function of a CLB, insight can be gained into the capacity of evolvable hardware to act as a fault recovery system in itself and provide a reliable method of recalibration should a device experience a fault which would otherwise render it useless.

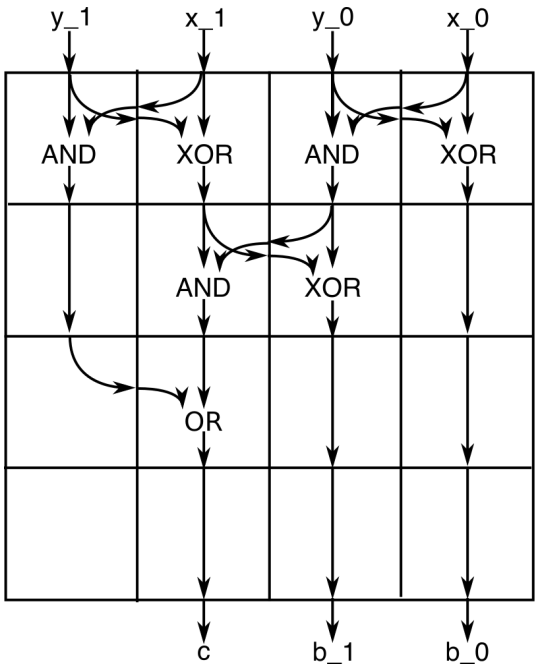
A textbook 2-bit adder was specified, as outlined in Figure 4.12(b). The population was seeded with this adder and the genetic algorithm allowed to operate. This constitutes the perfect accuracy experienced in Figure 4.12(a) until generation 1000. At this point a fault was injected in the CLB shaded red in Figure 4.12(c). This fault clamped the output of the binary function in the cell to an undefined value and therefore crippled the accuracy of the configuration. The system was then allowed to evolve as normal in the presence of such a fault.

At 1000 generations, when the fault is injected, the performance drops to levels well below the regular evolutionary search. However the percentage of executions ending with a perfect solution is much higher than evolution without the presence of the fault (Figure 4.4(e). This could be explained by examining the configuration which seeds the population; Figure 4.12(b) performs 2-bit addition using a fraction of the resources available to the FPGA. Only 7 CLBs are active and all of these are in the top half of the FPGA. This considerably more compact solution, even with an accuracy score of only 30, allows the genetic algorithm maximal space to experiment and improve fitness. Evolutionary dead ends are virtually eradicated by the efficient start point which provides the genetic algorithm a great deal of flexibility to orchestrate a solution from the bountiful resources. This leads to frequent perfect recoveries, and even if the solution evolved is not perfect, there are drastic performance improvements over the faulty chip almost immediately after the fault occurs. This experiment agrees with [14] who suggest deploying a genetic algorithm upon the detection of a fault.

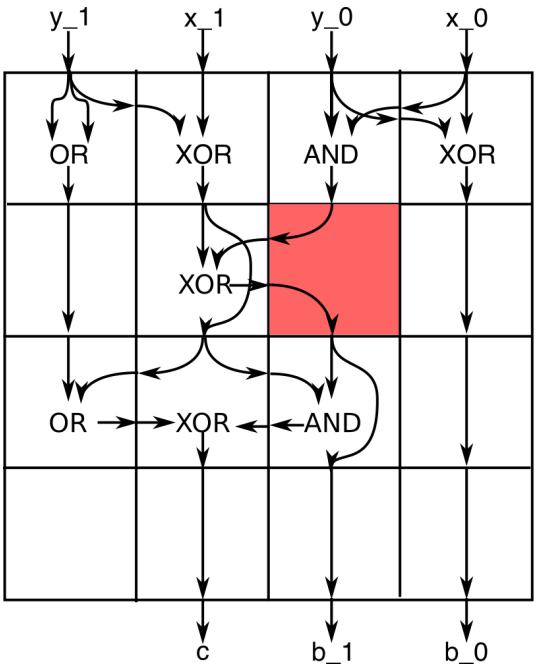
Once again it is worth noting the strange design choices made by the algorithm. Rather than pass data through a cell by routing an input to an output the cell in the top right ORs the northern input with itself and then passes this to the cell to the south. This is functionally equivalent and strange choices like these are commonplace in genetic algorithms where there is no evolutionary imperative to choose one solution over another.



(a)



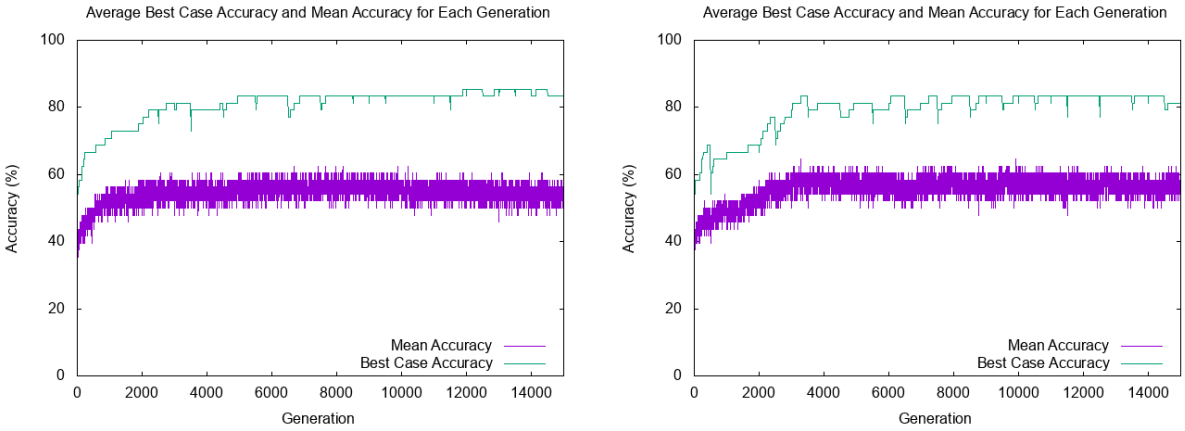
(b)



(c)

Perf. Runs (%)	Avg. Execution Time (s)	Avg. Best Accuracy (%)
43	237	88

Figure 4.12: Simple evolutionary fault recovery; (a) the accuracy over time, the fault is injected at generation 1000, (b) initial preloaded configuration, (c) one of the fault mitigation configurations evolved.



(a) (b)

Fault(s)	Perf. Runs (%)	Avg. Execution Time (s)	Avg. Best Accuracy
1	7	242	40
2	0	230	39

Figure 4.13: Evolutionary "sticky" fault recovery; faults oscillated between on and off every 500 generations, with (a) 1 randomly generated fault and (b) 2 randomly generated faults.

4.2.2 "Sticky" Fault Mitigation

"Sticky" faults occur when certain unpredictability environmental conditions cause a fault to occur, as such whether or not a fault is active oscillates seemingly at random. This unpredictablity makes diagnosis and mitigation difficult. To simulate a sticky fault, randomly generated faults were injected into the simulation, and activated or deactivated every 500 generations.

As one would anticipate the injection of a fault initially has an adverse effect on accuracy, as is clear in Figure 4.13, but eventually the damaging effects of the fault are lessened. With a single fault 7% of the evolutionary runs were capable of evolving a solution which performed perfectly with or without the fault. There are comparable results with 2 randomly generated faults, despite never evolving a perfect solution the limitations presented by the faults diminished over time as the system learned to cope.

The capacity to create solutions from scratch which are resilient to specific faults is clear. This agrees with [41] and [22], who use known fault models to generate hardware which can operate with high accuracy in the presence of a fault.

4.3 Dynamic Problem Optimisation

By introducing subtraction as a problem alongside addition, the relative weightings of a correct answer for each problem can be varied to explore dynamically changing problems. In the graph below the genetic algorithm starts with a perfect ADDer and 100% of the fitness weighting assigned to correct ADDs, every 200 generations this shifts by 10% towards SUBs.

Rather than continuous micro adjustments to maximise performance the behaviour indicated by Figure 4.14 is that of a system holding on to the initial configuration until extreme evolutionary pressure forces it to jump ship. It isn't until half way through execution, when ADDs and SUBs are weighted evenly that a major shift in performance for either occurs.

4.4 Scaling

The poor scaling qualities of evolvable hardware are a known issue, and unfortunately the system presented here is no different. Initial fears about the execution time of the evaluation function were assuaged by the experiments in Figure 4.7. Despite a diversity measurement which scales with complexity $O(n^2)$ as the population increases, the execution time grows linearly. This is because the regions of the algorithm scaling slowly are minimal and will not reach noticeable values until the population balloons to an

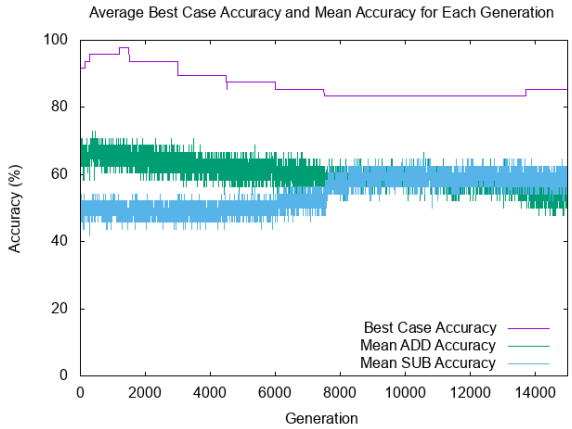


Figure 4.14: Accuracy in the presence of dynamic probelm weightings; every 1500 generations the accuracy weightings in the fitness function shifted 10% away from ADDs and towards SUBs.

FPGA Size (Width x Height)	Exec. Time (s)	Test Cases	Exec. Time per Test (s)
2x4	32	4	8
4x4	238	16	14
6x4	1473	64	23
8x4	8537	256	33

Figure 4.15: Execution time (s) for genetic algorithm operating on different FPGA sizes for a population of 50 over 15000 generations, the number of test cases is defined by the width of the FPGA.

unpractical size.

The main culprit for lengthy execution time is clearly therefore the instantiation and evaluation of FPGAs. Earlier in this document coevolutionary techniques were toted as a technique to minimise the number of evaluations, and therefore minimise the FPGA evaluation time; Figure 4.9 clearly demonstrates data to the contrary. Despite fewer tests in Figure 4.9(a) and Figure 4.9(b) the execution times are considerably lengthier than the previous non-coevolutionary searches. Initially it was thought that the culprit for these poor execution times was the incorporation of elitism, which relied on conventional scoring methods to maintain consistent growth in accuracy. Turning this off resulted in Figure 4.9(d), which despite executing quicker, performed terribly and was still slower than the conventional evolutionary counterpart.

An overlooked aspect of the coevolutionary system was the additional computation required to maintain this second population. This constitutes a huge execution overhead and ruins any capacity for coevolution to be used as a scaling mitigation strategy. These trails were, however, only conducted on relatively small problem sets. The 2-bit addition problem only has 16 trials for addition and 16 trails for subtraction.

Further exploring this scaling issue the size of the FPGA was varied. The evolutionary platform developed in conjunction with this document defines the scale of the problem being addressed by the size of the FPGA it is provided with. An FPGA of width w coincides with $\frac{w}{2}$-bit arithmetic. As such as the size of the FPGA scales in Figure 4.15 as does the problem, and therefore the number of test cases.

The ballooning execution time in Figure 4.15 highlights that the scaling issue is domineered by the number of test cases in each problem. A linear increase in FPGA size from 8 CLBs (2x4) to 32 CLBs (8x4) sees a 267 fold increase in execution time, but a 64 times increase in the number of test cases each FPGA is evaluated against. The execution time for each trail scaled slower than the FPGA size. This clearly identifies the scaling issue as a symptom of the number of test cases, despite minor growth in evaluation time for larger FPGAs. As evolvable hardware starts tackling larger and larger problems coevolution could be used to limit the number of test cases, if the dramatic performance disadvantages can be overcome.

Exploring the effects of a growing search space (as the FPGA size increases) is a well documented issue ([21][44][11]) but beyond the scope of this thesis.

Chapter 5

Conclusion

5.1 Summary of Achievements

My intention of recreating the genetic algorithm from Thompson's work on evolutionary hardware [42] and applying it to the binary arithmetic problem was successful, as outlined in Chapter 4. The initial genetic algorithm parameters in Section 4.1 mirror Thompson's exactly. When applied to 2-bit binary addition the algorithm executed quickly but in 30 repeated executions to generation 15000, failed to produce a perfect binary ADDer. The average final accuracy was 40 over 30 runs, out of a maximum of 48.

This baseline algorithm was then improved on by factoring in features from modern genetic algorithm and evolvable hardware literature. These features included; a multi-objective fitness function incorporating a diversity measurement, mutation rate variation, tournament selection, varied crossover probability, the absence of elitism, population size, multi-objective weightings, and coevolution (with varied virulence). Assessing each of these in turn a 2-bit binary addition specific genetic algorithm was devised with vastly improved performance. In repeated experiments almost 1/4 of executions to generation 15000 produced a perfect solution will 100% accuracy. This executed in comparable time to the initial parameters specified in [42], and achieved an average final fitness of 44, a clear improvement. Using a t-test this was shown to be a statistically significant improvement. The search space of FPGAs defined by 32 byte strings is huge. Clearly the scheme devised in this dissertation is a vast improvement over enumerated bruteforce search.

This system was then exposed to a series of dynamic problems; simple fault recovery, "sticky" fault mitigation, and shifting weighting optimisation. The system coped admirably with simple fault recovery; when a critical fault was injected in the FPGA simulation the accuracy dropped to a crippling low. In repeated executions, by 14000 generations evolving with the fault 43% had developed perfect solutions, and the average accuracy was 42. The success rate here is incredibly high when compared to the performance of the underlying evolvable hardware system which achieved perfect answers on 23% of repeated executions. With a single oscillating "sticky" fault on 7% the genetic algorithm was able to devise a solution which worked both in the presence of the fault and without it. Even with two faults clear improvements in how the system coped with shifting correctness criteria are visible.

The entire evolvable platform is available on GitHub. The FPGA simulation is a distinct component which exposes concise functionality to the evolutionary software component. The simulation can be downloaded independently and used not only as a component in other evolutionary systems but in any software project which could make use of a streamlined generic FPGA simulation of arbitrary developer defined size. The usefulness of the ability to convert binary strings into FPGAs and evaluate specified configurations extends beyond the context presented here and could form the backbone of a plethora of speculative hardware projects. This fits the FPGA simulation specification outlined in the project aims.

Exploration into the scaling problems of evolvable hardware outlined the key offenders contributing to extreme execution times. With simulated hardware the execution time scaled nicely with FPGA size, but the clear culprit in slow evaluation cycles was the sheer number of trails conducted for larger problems. Coevolution was proposed as a mitigation strategy, and would reduce the number of trials conducted; however, performance under the 2-bit problem was severely disappointing and was completely unsuitable as a replacement to conventional exhaustive testing. In order to achieve any sort of success elitism was non-optional and this required uniform exhaustive testing to function properly. Even without the extra computation required for elitism, the overheads maintaining a second distinct evolutionary population

lead to extreme average execution times.

A GUI was developed to provide visual feedback on the population health and genetic algorithm performance in an attempt to improve user understanding of the underlying process and drive parameter choices. This was somewhat successful, but the vast quantity of information on hand had to be reduced; the only detailed information on offer was for the current generation's best performing individual. For this an exhaustive list of test performance and a diagram of the FPGA configuration was provided to the user. The remaining population qualities were reduced to a handful of statistics; generation number, mean accuracy, and mean diversity. More innovation is required in this domain to truly offer a way to monitor a population and guide parameter choice.

There are a great number of limitations existing within this project, and evolvable hardware in a wider sense. 2-bit addition to most would seem trivial, and even though the execution time was painlessly short, directly extending this to 3-bit addition required 1473s and 4-bit addition 8537s. Designing an iterative design system around any larger problem than 2-bit binary arithmetic is simply not an option. Extending any assumptions here to useful applications requires solving a great number of challenges.

Visual inspection of many of the accuracy graphs in Chapter 4 could lead a reader to believe that many improvements are simply lucky leaps from a general background population which usually performs moderately; and the reader would not be wrong. Population quality tends to reach a certain value and then remain consistent, and best-case improvements become fewer and far between as execution continues. The only way a system is saved from backtracking is by the including of an elitism mechanic. These together indicate that the population improves until a certain value and then maintains a primordial soup, rarely performing well but on occasion generating an example with an improved performance. The cycle repeats but making improvements requires larger and larger leaps from this primordial soup.

The disparity between vanilla evolutionary hardware in Figure 4.8(b), only achieving a success rate of 23% and the fault injection examples in Figure 4.12 achieving 43% highlights the tendency for this evolvable hardware incarnation to stumble blindly into evolutionary dead ends, often dwelling needlessly on bloated hopeless solutions. These solutions take up too many of the resources allotted to the FPGA to achieve their mostly-good results to progress to perfection.

The system optimisation under dynamically shifting weightings is underwhelming, until the system is placed under real pressure the evolutionary tactic seems to be to maintain the current solution rather than perform minute tweaks to gain a performance advantage. The domain specific knowledge required to design hardware, through evolvable hardware, is replaced by domain specific knowledge for genetic algorithm design.

5.2 Project state

A fully flexible binary arithmetic evolutionary hardware platform works, with a plethora of options and configurable components. The FPGA simulator and GUI are a distinct self contained entity allowing for use in machine learning hardware design beyond genetic algorithms.

5.3 Further work

Given the self contained nature of the underlying FPGA simulation there is scope to extend this project with more machine learning mechanism to optimise the FPGA configuration for the binary addition problem. Further study into the underlying structure of the search space could provide insights which would allow further tuning of the genetic algorithm or, more likely, open the door to more esoteric specific implementations of other techniques, such as neural networks.

Remaining close to genetic algorithms the nature of the mutation function could shift. Rather than comprising random bitflips there could be a chance of a predefined operation with knowledge of the phenotype to genotype mapping, such as; cell translation/rotation, cell reordering, or mirroring cell connections. Such a modification would move this exploration to the fringes of genetic algorithm research, if not beyond the realm of, as genetic algorithms conventionally do not have any in-build understanding of what the genetic material represents (beyond the evaluation step).

If the plethora of issues surrounding system scaling could be addressed one could conceive of a hardware system similar to what was proposed in [14], save for a general purpose processor. Specific fault-prone units could be replaced with micro-FPGAs, each preloaded with the original component specification. Upon the detection of a fault the genetic algorithm begins. This iterates over the design finding ways to capitalise on the demonstrated fault recovery performance and devise a unit which works perfectly in the

presence of the fault. The vast majority of modern systems are multi-core so it is not unthinkable that should a component in one core fail any alternative cores could initiate and conduct the self-healing process. These systems could take advantage of evolvable hardware accelerator chips to create truly rugged circuitry. The evolutionary search could be conducted in CPU idling time, and even if it takes hours, a component which wouldn't otherwise work could be operational; ideal for satellites and/or operations working on a scale such that brief component downtime for the sake of longevity is not a huge issue.

If improvements in algorithm design leads to an evolvable hardware system which, when confronted by a dynamically shifting weighted problem, can perform prompt adjustments (at a speed faster than presented in this document) then a flexible pseudo-FPGA-CPU as outlined above could facilitate user/program specific fine tuning. Given contextual information (which shifts the desirability of certain functions and therefore the weightings) the processor could recalibrate to better suit it's function. Suppose, as a trivial example, user A performs a great deal of multiplication, and user B loves subtraction; given this information the processor should be able to optimise for each. Rather than trying to be accurate the processor could be assumed to be constantly accurate (other wise it will not change) and is instead trying to optimise for execution time. performs a lot of multiplication

Bibliography

[1] M.A. Almeida and Emerson Pedrino. Hybrid evolvable hardware for automatic generation of image filters. pages 1–15, 01 2018.

[2] Lionel Barnett. Ruggedness and evolvability-an evolution's-eye view. In *ALIFE*, page 748, 2008.

[3] Jürgen Branke and Hartmut Schmeck. *Designing Evolutionary Algorithms for Dynamic Optimization Problems*, pages 239–262. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[4] J. Cartlidge and S. Bullock. Combating coevolutionary disengagement by reducing parasite virulence. *Evolutionary Computation*, 12(2):193–222, June 2004.

[5] Jean-Francois Castet and Joseph H. Saleh. Satellite and satellite subsystems reliability: Statistical data analysis and modeling. *Reliability Engineering & System Safety*, 94(11):1718 – 1728, 2009.

[6] Edwin D. de Jong, Richard A. Watson, and Jordan B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, pages 11–18, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[7] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[8] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. volume 1 of *Foundations of Genetic Algorithms*, pages 69 – 93. Elsevier, 1991.

[9] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. Stageweb: Interweaving pipeline stages into a wearout and variation tolerant cmp fabric. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 101–110, June 2010.

[10] P. C. Haddow, M. Hartmann, and A. Djupdal. Addressing the metric challenge: Evolved versus traditional fault tolerant circuits. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 431–438, Aug 2007.

[11] M. Hartmann, P. K. Lehre, and P. C. Haddow. Evolved digital circuits and genome complexity. In *2005 NASA/DoD Conference on Evolvable Hardware (EH'05)*, pages 79–86, June 2005.

[12] Julien Henaut, Daniela Dragomirescu, and Robert Plana. Fpga based high date rate radio interfaces for aerospace wireless sensor systems. In *Systems, 2009. ICONS'09. Fourth International Conference on*, pages 173–178. IEEE, 2009.

[13] T. Higuchi, M. Iwata, I. Kajitani, H. Yamada, B. Manderick, Y. Hirao, M. Murakawa, S. Yoshizawa, and T. Furuya. Evolvable hardware with genetic learning. In *1996 IEEE International Symposium on Circuits and Systems. Circuits and Systems Connecting the World. ISCAS 96*, volume 4, pages 29–32 vol.4, May 1996.

[14] Tetsuya Higuchi, Masaya Iwata, Isamu Kajitani, Hitoshi Iba, Yuji Hirao, Tatsumi Furuya, and Bernard Manderick. Evolvable hardware and its application to pattern recognition and fault-tolerant systems. In Eduardo Sanchez and Marco Tomassini, editors, *Towards Evolvable Hardware*, pages 118–135, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[15] Tetsuya Higuchi, Masaya Iwata, D Keymeulen, Hidenori Sakanashi, Masahiro Murakawa, Isamu Kajitani, Eiichi Takahashi, Kenji Toda, N Salami, Nobuki Kajihara, and Nobuyuki Otsu. Real-world applications of analog and digital evolvable hardware. 3(3):220 – 235, 10 1999.

[16] W.Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*, 42(1):228 – 234, 1990.

[17] Jin-Hyuk Hong and Sung-Bae Cho. Meh: modular evolvable hardware for designing complex circuits. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 1, pages 92–99 Vol.1, Dec 2003.

[18] Greg Hornby, Al Globus, Derek Linden, and Jason Lohn. Automated antenna design with evolutionary algorithms. 1, 09 2006.

[19] Isamu Kajitani, Tsutomu Hoshino, Nobuki Kajihara, Masaya Iwata, and Tetsuya Higuchi. An evolvable hardware chip and its application as a multi-function prosthetic hand controller. In *AAAI/IAAI*, 1999.

[20] T. Kalganova and J. Miller. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pages 54–63, 1999.

[21] Tatiana Kalganova. An extrinsic function-level evolvable hardware approach. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming*, pages 60–75, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[22] D Keymeulen, Ricardo Salem Zebulum, Yili Jin, and Adrian Stoica. Fault-tolerant evolvable hardware using field-programmable transistor arrays. 49:305 – 316, 10 2000.

[23] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, February 2008.

[24] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. 9, 11 1995.

[25] Paul Layzell and Adrian Thompson. Understanding inherent qualities of evolved circuits: Evolutionary history as a predictor of fault tolerance. In Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, pages 133–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[26] Shyh-Chang Lin, Erik D. Goodman, and William F. Punch. A genetic algorithm approach to dynamic job shop scheduling problem. In *ICGA*, 1997.

[27] Jason Lohn, Greg Larchev, and Ronald DeMara. A genetic representation for evolutionary fault recovery in virtex fpgas. In AAndy M. Tyrrell, Pauline C. Haddow, and Jim Torresen, editors, *Evolvable Systems: From Biology to Hardware*, pages 47–56, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[28] Kalganova Miller, J. F. Miller, and N. Lipnitskaya. Multiple-valued combinational circuits synthesised using evolvable hardware approach. In *Proc. of the 7th Workshop on Post-Binary Ultra Large Scale Integration Systems (ULSI'98) in association with ISMVL'98*, page pp. IEEE Press.

[29] E. Monmasson and M. N. Cirstea. Fpga design methodology for industrial control systems; a review. *IEEE Transactions on Industrial Electronics*, 54(4):1824–1842, Aug 2007.

[30] R. N. Pedersen. Fpga-based military avionics computing circuits. *IEEE Aerospace and Electronic Systems Magazine*, 19(7):9–13, July 2004.

[31] Mitchell A. Potter and Kenneth A. De Jong. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evol. Comput.*, 8(1):1–29, March 2000.

[32] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. *SIGARCH Comput. Archit. News*, 37(3):93–104, June 2009.

[33] Eduardo Sanchez, Daniel Mange, Moshe Sipper, Marco Tomassini, Andres Perez-Uribe, and André Stauffer. Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware. In Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, editors, *Evolvable Systems: From Biology to Hardware*, pages 33–54, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[34] S. D. Scott, A. Samal, and S. Seth. Hga: A hardware-based genetic algorithm. In *Third International ACM Symposium on Field-Programmable Gate Arrays*, pages 53–59, 1995.

[35] Lukáš Sekanina. Image filter design with evolvable hardware. In Stefano Cagnoni, Jens Gottlieb, Emma Hart, Martin Middendorf, and Günther R. Raidl, editors, *Applications of Evolutionary Computing*, pages 255–266, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[36] Lukas Sekanina. Evolutionary hardware design. 8067, 05 2011.

[37] Jyothish Soman and Timothy M. Jones. High performance fault tolerance through predictive instruction re-execution. 2017.

[38] A. Stoica, D. Keymeulen, Vu Duong, R. Zebulum, I. Ferguson, T. Daud, T. Arsian, and Xin Guo. Evolutionary recovery of electronic circuits from radiation induced faults. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, volume 2, pages 1786–1793 Vol.2, June 2004.

[39] A. Stoica, D. Keymeulen, R. Zebulum, A. Thakoor, T. Daud, Y. Klimeck, R. Tawel, and V. Duong. Evolution of analog circuits on field programmable transistor arrays. In *Proceedings. The Second NASA/DoD Workshop on Evolvable Hardware*, pages 99–108, 2000.

[40] E. Stomeo, T. Kalganova, and C. Lambert. Generalized disjunction decomposition for evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 36(5):1024–1043, Oct 2006.

[41] A. Thompson. Evolutionary techniques for fault tolerance. In *Control '96, UKACC International Conference on (Conf. Publ. No. 427)*, volume 1, pages 693–698 vol.1, Sept 1996.

[42] Adrian Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, editors, *Evolvable Systems: From Biology to Hardware*, pages 390–405, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[43] Adrian Thompson. On the automatic design of robust electronics through artificial evolution. In *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware*, ICES '98, pages 13–24, London, UK, UK, 1998. Springer-Verlag.

[44] Jim Torresen. A scalable approach to evolvable hardware. *Genetic Programming and Evolvable Machines*, 3(3):259–282, Sep 2002.

[45] Andy M. Tyrrell, Eduardo Sanchez, Dario Floreano, Gianluca Tempesti, Daniel Mange, Juan-Manuel Moreno, Jay Rosenberg, and Alessandro E. P. Villa. Poetic tissue: An integrated architecture for bio-inspired hardware. In *Proceedings of the 5th International Conference on Evolvable Systems: From Biology to Hardware*, ICES'03, pages 129–140, Berlin, Heidelberg, 2003. Springer-Verlag.

[46] Zdenek Vasicek and Lukas Sekanina. Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. *Genetic Programming and Evolvable Machines*, 12(3):305–327, Sep 2011.

[47] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 193–204, New York, NY, USA, 2010. ACM.

[48] James Alfred Walker, James A. Hilder, and Andy M. Tyrrell. Evolving variability-tolerant cmos designs. In Gregory S. Hornby, Lukáš Sekanina, and Pauline C. Haddow, editors, *Evolvable Systems: From Biology to Hardware*, pages 308–319, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[49] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.

[50] P. Zhu, R. Yao, and J. Du. Design of self-repairing control circuit for brushless dc motor based on evolvable hardware. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 214–220, July 2017.

Appendix A

simulator.h

This shows the functionality exposed to the evolvutionary front-end by the FPGA simulator, and how internal data is organised.

```
#include <stdio.h>
#include <ncurses.h>
#include <curses.h>
#include <math.h>

#define FPGA_HEIGHT 4
#define FPGA_WIDTH 4
#define STRING_LENGTH_BYTES FPGA_WIDTH * FPGA_HEIGHT * 2
#define FAULT_NUM 1
#define FAULT_TYPE_CON 0

typedef enum {
        OFF,
        NOT,
        OR,
        AND,
        NAND,
        NOR,
        XOR,
        XNOR
} Gate;

typedef enum {
        NORTH,
        EAST,
        SOUTH,
        WEST,
        F
} Direction;

typedef struct {
        int x, y;
        Direction dir;
        unsigned char value;
} Fault;

typedef struct {
        Direction n_out, e_out, s_out, w_out;
        Gate gate;
        Direction g_in1, g_in2;
```

```
        //3 values: 0, 1, 2 (2 represents undefined)
        unsigned char n_in, e_in, s_in, w_in;
        unsigned char n_val, e_val, s_val, w_val;
} Cell;

typedef struct {
        Cell cells[ FPGA_HEIGHT ][ FPGA_WIDTH ];
        unsigned char control;
        unsigned char input[ FPGA_WIDTH ];

        Fault faults[ FAULT_NUM ];
        int active_fault[ FAULT_NUM ];
} FPGA;

/*
 * FPGA is defined by a bitstring of the following format:
 *              - the bitstring defines cells from left to right, top to bottom,
 *                row by row, one byte per cell
 *              - the least significant 2 bits define the value pushed to n_out
 *                (F,EAST,SOUTH,WEST)
 *              - the next 2 define the value pushed to e_out (NORTH,F,SOUTH,WEST)
 *              - the next 2 define the value pushed to s_out (NORTH,EAST,F,WEST)
 *              - the next 2 define the value pushed to w_out (NORTH,EAST,SOUTH,F)
 *              - the next 2 define where the first input for F comes from
 *                (NORTH,EAST,SOUTH,WEST)
 *              - the next 2 define where the second input for F comes from
 *                (NORTH,EAST,SOUTH,WEST)
 *              - the next 3 define the function F performs
 *                (OFF,NOT,OR,AND,NAND,NOR,XOR,XNOR)
 *              - the most significant bit is reserved
 */

void bitstring_to_fpga ( FPGA *fpga, unsigned char *bits );

void evaluate_fpga ( FPGA *fpga );

void init_curses ();

void redraw ( int test_loop, int iteration, FPGA fpga, int most_fit, int mean_fit,
int mean_div, int add_weight, int sub_weight );

void tidy_up_curses();
```

Appendix B

evolve.h

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include <time.h>
#include "simulator.h"

#define TEST_SIZE 30
#define TEST_LOOP 15000
#define WEIGHT_TEST 0
#define FAULT_INJECTION 0
#define POP_SIZE 50
#define MUTATION 3.0f
#define SIZE_WEIGHT 0
#define DIVERSITY_WEIGHT 2
#define ELITISM 1
#define FITNESS_WEIGHT 10
#define COEVOLVE 0
#define STICKY 0
#define LOG 0
#define PROB_SKEW 1.0f //between 0 or 1, 1 is linear
#define VIRULENCE 1.0f
#define PARASITE_SIZE 16
#define CROSSOVER 0.7f
#define TOURNAMENT_SIZE 20 //set to POP_SIZE for rank-based selection
#define RUGGEDNESS 0

int add_weight, sub_weight;

typedef struct Individual {
        unsigned char values[ STRING_LENGTH_BYTES ];
        int eval[ 3 ];
        int add_score, sub_score;
        FPGA fpga;
} Individual;

typedef struct Parasite {
        unsigned char values[ PARASITE_SIZE ];
        float score;
} Parasite;
```

Appendix C

Experimental Results

Figure	Pop. Size	Fitness Func.	Selection	Elitism	Crossover Prob.	Mutation Rate	Coevolution	Perf. Runs (%)	Avg. Exec. Time (s)	Avg. Best Accuracy (%)
4.1(a)	50	Simple	Rank (skew=1)	Yes	0.7	2.7	No	0	235	81
4.1(b)	50	Multi-Objective[2]	Rank (skew=1)	Yes	0.7	2.7	No	0	267	83
4.2(a)	500	Simple	Rank (skew=1)	Yes	0.7	2.7	No	0	2250	90
4.2(b)	500	Multi-Objective[2]	Rank (skew=1)	Yes	0.7	2.7	No	0	2857	83
4.3(a)	50	Multi-Objective[2]	Rank (skew=1)	Yes	0.7	1	No	0	245	83
4.3(b)	50	Multi-Objective[2]	Rank (skew=1)	Yes	0.7	2	No	0	264	83
4.3(c)	50	Multi-Objective[2]	Rank (skew=1)	Yes	0.7	3	No	0	363	83
4.3(d)	50	Multi-Objective[2]	Rank (skew=1)	Yes	0.7	4	No	0	368	83
4.4(a)	50	Multi-Objective[2]	Rank (skew=0)	Yes	0.7	3	No	0	304	83
4.4(b)	50	Multi-Objective[2]	Rank (skew=0.5)	Yes	0.7	3	No	0	415	83
4.4(d)	50	Multi-Objective[2]	Tournament[4]	Yes	0.7	3	No	0	264	88
4.4(e)	50	Multi-Objective[2]	Tournament[5]	Yes	0.7	3	No	13	231	88
4.4(f)	50	Multi-Objective[2]	Tournament[6]	Yes	0.7	3	No	3	260	83
4.4(g)	50	Multi-Objective[2]	Tournament[7]	Yes	0.7	3	No	0	244	83
4.5(a)	50	Multi-Objective[2]	Tournament[5]	Yes	0	3	No	10	243	85
4.5(b)	50	Multi-Objective[2]	Tournament[5]	Yes	0.5	3	No	3	245	85
4.5(c)	50	Multi-Objective[2]	Tournament[5]	Yes	1	3	No	3	249	85
4.5(c)	50	Multi-Objective[2]	Tournament[5]	Yes	1	3	No	3	249	85
4.6	50	Multi-Objective[2]	Tournament[5]	No	0.7	3	No	0	281	40
4.7(a)	25	Multi-Objective[2]	Tournament[5]	Yes	0.7	3	No	0	126	83
4.7(c)	100	Multi-Objective[2]	Tournament[5]	Yes	0.7	3	No	17	457	88
4.7(d)	200	Multi-Objective[2]	Tournament[5]	Yes	0.7	3	No	57	955	92
4.8(a)	50	Simple	Tournament[5]	Yes	0.7	3	No	16	235	88
4.8(b)	50	Multi-Objective[1]	Tournament[5]	Yes	0.7	3	No	23	238	94
4.8(d)	50	Multi-Objective[3]	Tournament[5]	Yes	0.7	3	No	10	247	85
4.9(a)	50	Multi-Objective[1]	Tournament[5]	Yes	0.7	3	Yes[8]	0	740	71
4.9(b)	50	Multi-Objective[1]	Tournament[5]	Yes	0.7	3	Yes[9]	0	605	77
4.9(c)	50	Multi-Objective[1]	Tournament[5]	Yes	0.7	3	Yes[10]	0	703	71
4.9(d)	50	Multi-Objective[1]	Tournament[5]	No	0.7	3	Yes[9]	0	535	50

[1]Diversity Weight 20%
[2]Diversity Weight 40%
[3]Diversity Weight 60%
[4]Size 10
[5]Size 20
[6]Size 30
[7]Size 40
[8]Parasite Size 8
[9]Parasite Size 16
[10]Parasite Size 32