Dissertation Type: research


DEPARTMENT OF COMPUTER SCIENCE

# Exploring Evolutionary Hardware and its Application to 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.

---

Monday 30th April, 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, Monday 30$^{th}$ April, 2018

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Listings

# Executive Summary

**A compulsory section, of at most 1 page**

This section should précis the project context, aims and objectives, and main contributions (e.g., deliverables) and achievements; the same section may be called an abstract elsewhere. The goal is to ensure the reader is clear about what the topic is, what you have done within this topic, *and* what your view of the outcome is.

The former aspects should be guided by your specification: essentially this section is a (very) short version of what is typically the first chapter. Note that for research-type projects, this **must** include a clear research hypothesis. This will obviously differ significantly for each project, but an example might be as follows:

> My research hypothesis is that a suitable genetic algorithm will yield more accurate results (when applied to the standard ACME data set) than the algorithm proposed by Jones and Smith, while also executing in less time.

The latter aspects should (ideally) be presented as a concise, factual bullet point list. Again the points will differ for each project, but an might be as follows:

- I spent 120 hours collecting material on and learning about the Java garbage-collection sub-system.
- I wrote a total of 5000 lines of source code, comprising a Linux device driver for a robot (in C) and a GUI (in Java) that is used to control it.
- I designed a new algorithm for computing the non-linear mapping from A-space to B-space using a genetic algorithm, see page 17.
- I implemented a version of the algorithm proposed by Jones and Smith in [6], see page 12, corrected a mistake in it, and compared the results with several alternatives.

TODO: include motivation straight away

The intersection between machine learning and hardware design is an oft unexplored area. Evolutionary hardware is one such field which explores the application of genetic algorithms to hardware design. This thesis explores how genetic algorithms can be improved in this context and the resulting evolutionary hardware systems applied to a series of dynamic problems.

Evolutionary hardware systems are built on FPGAs; to improve development time and remove execution bottlenecks a simulated FPGA will be constructed. This will provide the evaluation backend to the genetic algorithm.

My research hypothesis is that genetic algorithms can be used to construct efficient hardware systems suited for tackling dynamic problems. These problems include logic fault recovery or optimising in situe for a shifting problem.

The main achievements include:

- Building a highly specialised FPGA simulator (in C) to allow quick genetic algorithm development.
- Implimenting and improving on known genetic algorithm to create FPGA configurations. The implimented improvments include; multi-objective training, rank based selection, parameter changes, co-evolution with variable parasite virulence.

- Exploring fault recovery and dynamic optimisation with evolutionary hardware on the FPGA simulator.
- I created a clean user interface to allow for easy mid-execution user evaluation.
- Explore scaling bottlenecks and potential mitigation with coevolutionary techniques.

# Supporting Technologies

**A compulsory section, of at most 1 page**

This section should present a detailed summary, in bullet point form, of any third-party resources (e.g., hardware and software components) used during the project. Use of such resources is always perfectly acceptable: the goal of this section is simply to be clear about how and where they are used, so that a clear assessment of your work can result. The content can focus on the project topic itself (rather, for example, than including "I used LaTeX to prepare my dissertation"); an example is as follows:

- I used the Java `BigInteger` class to support my implementation of RSA.
- I used a parts of the OpenCV computer vision library to capture images from a camera, and for various standard operations (e.g., threshold, edge detection).
- I used an FPGA device supplied by the Department, and altered it to support an open-source UART core obtained from `http://opencores.org/`.
- The web-interface component of my system was implemented by extending the open-source WordPress software available from `http://wordpress.org/`.

- I used the ncurses C library to develop a GUI

# Notation and Acronyms

**An optional section, of roughly 1 or 2 pages**

Any well written document will introduce notation and acronyms before their use, *even if* they are standard in some way: this ensures any reader can understand the resulting self-contained content.

Said introduction can exist within the dissertation itself, wherever that is appropriate. For an acronym, this is typically achieved at the first point of use via "Advanced Encryption Standard (AES)" or similar, noting the capitalisation of relevant letters. However, it can be useful to include an additional, dedicated list at the start of the dissertation; the advantage of doing so is that you cannot mistakenly use an acronym before defining it. A limited example is as follows:

| | | |
|---|---|---|
| AES | : | Advanced Encryption Standard |
| DES | : | Data Encryption Standard |
| | ⋮ | |
| $\mathcal{H}(x)$ | : | the Hamming weight of $x$ |
| $\mathbb{F}_q$ | : | a finite field with $q$ elements |
| $x_i$ | : | the $i$-th bit of some binary sequence $x$, st. $x_i \in \{0, 1\}$ |

| | | |
|---|---|---|
| FPGA | : | Field Programable Gate Array |
| ASIC | : | Application Specific Integrated Circuit |
| EHW | : | Evolvable Hardware |
| Fitness | : | A measure of how well an individual fulfils it's fitness function |
| Correctness | : | Number of bits correct |

# Acknowledgements

**An optional section, of at most** 1 **page** It is common practice (although totally optional) to

acknowledge any third-party advice, contribution or influence you have found useful during your work. Examples include support from friends or family, the input of your Supervisor and/or Advisor, external organisations or persons who have supplied resources of some kind (e.g., funding, advice or time).

# Chapter 1

# Contextual Background

**A compulsory chapter, of roughly 5 pages**

This chapter should describe the project context, and motivate each of the proposed aims and objectives. Ideally, it is written at a fairly high-level, and easily understood by a reader who is technically competent but not an expert in the topic itself.

In short, the goal is to answer three questions for the reader. First, what is the project topic, or problem being investigated? Second, why is the topic important, or rather why should the reader care about it? For example, why there is a need for this project (e.g., lack of similar software or deficiency in existing software), who will benefit from the project and in what way (e.g., end-users, or software developers) what work does the project build on and why is the selected approach either important and/or interesting (e.g., fills a gap in literature, applies results from another field to a new problem). Finally, what are the central challenges involved and why are they significant?

The chapter should conclude with a concise bullet point list that summarises the aims and objectives. For example:

> The high-level objective of this project is to reduce the performance gap between hardware and software implementations of modular arithmetic. More specifically, the concrete aims are:
>
> 1. Research and survey literature on public-key cryptography and identify the state of the art in exponentiation algorithms.
> 2. Improve the state of the art algorithm so that it can be used in an effective and flexible way on constrained devices.
> 3. Implement a framework for describing exponentiation algorithms and populate it with suitable examples from the literature on an ARM7 platform.
> 4. Use the framework to perform a study of algorithm performance in terms of time and space, and show the proposed improvements are worthwhile.

Unlike conventional hardware design, evolvable hardware (EHW) takes advantage of machine learning techniques applied to flexible hardware platforms to explore the design space of potential circuit solutions to hardware problems. Genetic algorithms are the key machine learning tool which takes advantage of the hardware flexibility to navigate an inherrently problematic search space. Traversing the set of solutions is of immense difficulty; the fitness landscape is discontinuous and search by random means often proves the most fruitful. By combining the robust non-determinism inherent in genetic algorithms and hardware design one can autonomously develop application specific hardware.

TODO: add a stack of citations

## 1.1 Evolvable Hardware

Evolvable hardware has explored the application of genetic algorithms to the domain of hardware design for decades now. Genetic algorithms come from the field of natural computing and use Darwinian-inspired probabilistic procedure to improve a population of solution's performance given a fitness criteria with

evolutionary pressure. In the case of evolvable hardware, each individual is a bitstring which can be mapped onto a circuit design and the fitness critera is rooted in the circuit's ability to perform some predefined function.

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

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 construct a novel solution. Thus far there have been great successes deploying evolvable hardware to create user-specific prosthetic hand controllers [19], image filters [1], and creating arbitrary logic circuits [40]. Although current solutions are limited by scaling issues.

One of the advantages of genetic algorithms is also one of their largest weaknesses; they excel in generating strange and esoteric solutions, which can often outperform their more conventional hand-designed counterparts. An example of which (although outside the domain of circuit synthesis) comes from NASA, where an evolutionary algorithm designed an ironically alien antenna for use in space [18], this antenna performed better than any hand designed counterpart. Unfortunately the sometimes counter-intuitive solutions, seemingly arbitrary automatic design decisions and incomprehensible structure results in a high performing object with no ability to perform direct visual comparison to known solutions and great difficulty understanding how a system works. Analysis of the resulting hardware produced by an evolvable hardware project rarely extend beyond beyond an acknowledgement of functional correctness.

The function defining the correctness of an evolvable system can balloon to an unwieldy size. For the algorithm to take into account efficiency, fault tolerance, size or any other desired circuit characteristic the fitness function has to become progressively more complex. Leading to computationally intensive evaluation procedures.

Unfortunatley many applications of genetic algorithms are not as wildly successful as NASA's antenna. In many cases and by many design metrics genetic algorithms rarely produce results better than the hand designed alternatives. However, these hand designs require a huge amount of intuition from the 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 the most part been dormant in recent years. I hope to explore how modern genetic algorithm techniques can craft an algorithm capable of designing solutions to conventional hardware problems, such as simple arithmatic. TODO: sprincle a few genetic algorithm improvement citations through this

### 1.1.1 Field Programmable Gate Arrays (FPGAs)

Field Programable Gate Arrays 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 wealth of operations. Conventionally an FPGA design is specified in a HDL 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 neighboring cells, has some internal binary function, and sends distinct outputs to each of their neighboring cells. The values sent to the output can be the result of the function or any of the inputs. The function, the input(s) to the function and which values are sent to each output are completely configurable. Figure 1.1 shows the internal structure of a CLB and how they are 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 [25]. If properly configured (and containing an appropriate number of CLBs) an FPGA can be functionally identical 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 function their functionality can be modified at will. FPGAs share the power efficiency and execution proficiency of ASIC hardware, but the production costs do not scale as well as ASIC chips. However, ASIC development can be a lengthy and expensive process, one which often does not make financial sense, due to the scale

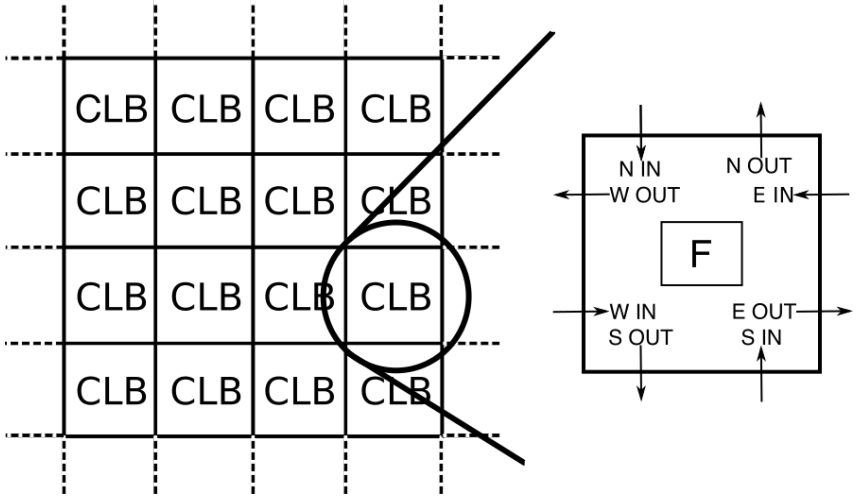



Figure 1.1: FPGA architecture

randomly initialise population $P$
**while** *no perfect solution* **do**
  $P' \leftarrow Select(P)$
  $Crossover(P')$
  $Mutate(P')$
  $P \leftarrow P'$
**end**

**Algorithm 1.1:** Basic genetic algorithm

of the chip requirements and associated small-scale manufacturing costs. This pressure for highly specific cost effective hardware on a small scale driven mainstream FPGA development.

These immensly 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 this case a team of engineers can develop software as the hardware is being developed in parallel by using an FPGA loaded with a beta version of chip. This cuts out 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 and crypto currency miners; this is because FPGAs configurations are cheeper to develop than ASIC hardware, highly application specific, and see huge performance and power consumption benefits over more general purpose hardware (GPGPUs for example). The flexibility also means itterative improvements can be made to coincide with new discoveries without the need to purchase new hardware. Similarly many industrial (and military) institutions require bespoke hardware solutions to highly specific problems; maybe 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 results in the widespread use of FPGAs in roles requiring an ASIC but without the market pressure to drive the development of one. An application specific design can be built on an FPGA, and approach peak performance and power efficiency without massive development and production costs. The military and aerospace sectors use FPGAs extensively for these reasons along with the built-in cryptographic and anti-tamper hardware obfuscation capabilities of 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, and mutation to move the population towards potential solutions to a given problem. 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.

The binary string associated to individual in the population is considered the genotype, and a phenotype what the string represents in the context of the problem. 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 (often an FPGA config-

uration). Selection (*Select*) generates a new population by evaluating each member of the population then randomly choosing individuals from the old population with a probability proportional to it's evaluated fitness. Crossover (*Crossover*) randomly pairs each member of the new population and given a probability swaps the genetic material from one individual with another from a random point. Given an expected mutation-per-individual $m$, and population size $n$ the *Mutate* function iterates over each bit in each bitstring flipping the bit with probability $\frac{m}{n}$. These simple processes coalesce into a high performance search procedure.

### 1.1.3 Genetic Algorithms with an 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 is the root of evolutionary hardware. The bottleneck for physical evolvable hardware is often the evaluation step, this involves uploading a series of configurations to the FPGA and extensively testing each. When each upload takes in the order of multiple seconds the evolution 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.

## 1.2 Dynamic Problems

A problem with evaluation critera which shift 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 tollerance.

### 1.2.1 Hardware Faults

One popular use of dynamic problems in evolutionary hardware is employed to breed fault tolerance into a design. This involves repeatedly turning on and off faults in the FPGA evaluation to create a design both robust to faults and not reliant on faults. Another fault resilience techniques requires evaluating the configuration against a fault-free FPGA and then combining the result with evaluation runs on FPGAs simulating known frequent faults. This extension of the fitness function (rather than mid-execution modification) removes this approach from the domain of dynamic problems, but it is worth noting as an alternate method to improve toughness.

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 benifits to improving the fault tolerance of devices. An extreme example of high cost of replacement can be found with satellites, survays of in-orbit satalites 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 strange 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 satellites into orbit set in the millions of pounds extending the lifespan of such devices would have significant economic impact.

A little closer to home, datacenters are vast structures contain thousands of servers. Each of these has an 8% probability of experiencing a failure each year [41]. Individually this is of no great concern, but when compounded across an entire datacenter server recovery and replacement becomes a primary concern for the management of such an establishment.

As previously stated predefined faults have been used to develop evolveable 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.

### 1.2.2 Dynamic optimisation

Another area of dynamic problems with evolutionary hardware that has seen success involves extending the evaluation function when an perfect solution has been found. 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 be as small 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 varying the relative benifits for correct answers to either has on the performance of 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 aims of this project is to develop an improved evolvable hardware platform capable of effectively addressing dynamic problems. More specifically:

- Apply the genetic algorithm from [37] to the binary arithmatic problem.
- Combine state of the art genetic algorithms to improve evolvable hardware performance on binary arithmatic.
- Explore the application of evolvable hardware to dynamic problems, including FPGA faults and weighted binary arithmatic.
- Develop a specialsied FPGA simulator to act as the evaluation backend 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/-subtraction).
- FPGA configurations for a variety of problems investigated here are very fragile, in the context of evolution, minor mutations could be disastrous.
- Moreso than in many evolutionary contexts the prospect of evolutionary deadends and dominating local optima will need to be addressed.

# Chapter 2

# Technical Background

**A compulsory chapter, of roughly 10 pages**

This chapter is intended to describe the technical basis on which execution of the project depends. The goal is to provide a detailed explanation of the specific problem at hand, and existing work that is relevant (e.g., an existing algorithm that you use, alternative solutions proposed, supporting technologies).

Per the same advice in the handbook, note there is a subtly difference from this and a full-blown literature review (or survey). The latter might try to capture and organise (e.g., categorise somehow) *all* related work, potentially offering meta-analysis, whereas here the goal is simple to ensure the dissertation is self-contained. Put another way, after reading this chapter a non-expert reader should have obtained enough background to understand what *you* have done (by reading subsequent sections), then accurately assess your work. You might view an additional goal as giving the reader confidence that you are able to absorb, understand and clearly communicate highly technical material.

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 they developed a robust directed non-deterministic procedure to navigate search spaces with non-continuous fitness. These search spaces are not well suited to conventional search methods and previously were only navigatable by random walks, or bruteforce search, and genetic algorithms provide a good general method to approach problems where such little information is known about a space of solutions. David Goldberg's book *Genetic Algorithms: In Search Optimisation & Machine Learning* [9] contains a good summary of the knowledge of genetic algorithms as it stood in the early 1990s after the vast majority of what we know had been published. 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 so 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** allows two strings in the new population to reproduce. Members of the breeding population are paired up at random, and there is a given probability that each pair performs crossover. For each

mating pair, 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 stage can be omitted for simpler evolution.

**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 unfeesable.

An interesting tagential application of FPGA technology to evolutionary algorithms comes as a physical evolution hardware accelerator [30], 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 in hardware. This highlights the oportunity 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 [10]. They compared proportionate selection, rank based selection, and tournament selection, amongst others.

**Proportionate selection** is the set of selection mechanisms which includes roulette wheel selection. All proportional selection schemes tie the probability of an individual being selected for the next generation to some function of their fitness function, $f$.

**Rank selection** orders each member of a population by their fitness. Then associates the probability of each individual being selected as an individual in the next generation 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 of individual fitness.

**Tournament selection** 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 influences the descision to incorporate them into an evolvable hardware platform. Proportionate selection discriminates heavily by the fitness of indervidiuals. Rank based discriminates less and cares more about who is better (rather than by how much they are better). Tournament selection discriminates intensley within a given tournament but the selection process to assemble the tournament is uniformally random, so depending on the tournament size this can be highly descriminatory (large tournament size) or minimally descriminatory (small tournament size). 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 used frequently.

### 2.1.2 The Fitness Function

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

By extending the fitness function to a weighted linear combination of a function measuring correctness (the original fitness functon) and any other measurements we can deploy evolutionary pressure to optimise for additional parameters [6]. For an individual $x$, a set of functions measuring distinct desirable aspects of individual $\{f_1, f_2, \ldots, f_m\}$, and a set of weightings associated with each function $\{w_1, w_2, \ldots, w_m\}$ 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.

### 2.1.3 Coevolution

Coevolution refers to the practice of evolving to populations in tandem, this can act as a way to divide labour (two populations solving different parts of a problem) [28], or as adversaries (one population of problem proposers and another of problem solvers). The former is used to improve the scalability of evolveable hardware to naturally decompose the problem into subproblems. The latter is often likened to a host/parasite or pray/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.

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 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 a maximal virulence.

In some settings the 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 some easy ones. 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 (TODO: insert figure) Bullock et al. [4] discouraged maximally virulent parasites. This encourages populations to stay engaged.

## 2.2 Evolvable Hardware

### 2.2.1 Theory

The foundational work in evolveable hardware is summarised by T. Higuchi et al. in their paper *Evolvable Hardware with Genetic Learning* [15]. 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 calibration bitstring; only a subset of the bitstring include bits relevant to the function of a specific region of the hardware. 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 a image pattern recognition system, and a welding robot controller which takes sensor information and traces a ditch.

TODO: add graph of gate level evolution, phenotype -¿ genotype mapping

They 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 Adrian Thompson [37]. 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.

TODO: include 10x10 grid schematic

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.

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 EHW schemes use simulations to improve training time), a series of test frequencies were fed into the device and an single cell's output was read. The fitness of a configuration was defined as the difference between the average output for each different input frequency (multiplied by a constant to avoid otherwise inescapable local optimum).

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 Thompsons 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*. 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 to nurture more than just correctness. 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 innactive. 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.

Evolving the size and shape of the circuitry involves defining how mutation and recombination of circuit size operates. The mutation step alters the number of rows and columns of a circuit with equal probability, and new cells are initialised randomly. Recombination is a mechanism for crossover of non-uniform circuit sizes which involves exchanging cells and way to preserve positional connections of cells in the case that the new circuit is too restrictive for direct copying. Both multi-objective fitness functions and evolved circuit structure apply evolutionary pressure to improving 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) [34]. 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 reduces the search space of solutions and improves search performance at the cost of reduced flexibility to construct novel circuitry.

A similar method proposed by J. Torrensen suggests allowing the evolutionary process itself to subdivide the process, increased complexity evolution [39]. 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.

TODO: insert image of "partitioned training vectors" and "partitioned training sets" from "A scalable

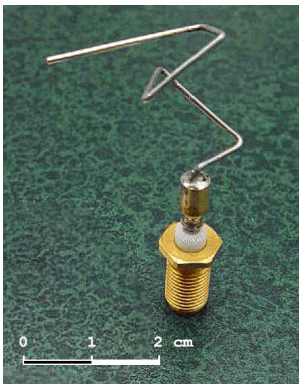



Figure 2.1: NASA evolved antenna [18]

approach to evolvable hardware"

These methods all deal with reducing the search space or dividing the problem into more manageable workloads. The problem remains with the 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 ($2^{2 \cdot 2}$) possible combinations of 2 2-bit numbers, whereas for a 3-bit adder this value grows to 64 ($2^{2 \cdot 3}$). For practical evolution of larger circuits this requires some careful thought.

TODO: Generalized Disjunction Decomposition for Evolvable Hardware [35] Addresses scaling issue, offers a new type of decomposition strategy.

TODO: Evolved Digital Circuits and Genome Complexity [14] Improving scaling issue my investigating genotypic complexity, also considers shrinking search space

TODO: Extrinsic vs intrinsic evolution

### 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 2.1 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 is long.

TODO: Evolutionary hardware design [31] talk about case studies, logic synthesis [40], image filter evolution

TODO: Hybrid Evolvable Hardware for automatic generation of image filters [1] Architecture for fast image filters - no need to use a bitstream to reprogram

A series of general purpose evolvable hardware chips have been developed for applications ranging from telecommunications circuits to robot controllers. Using tournament selection and elitism T. Higuchi et al. created a sensor driven robot controller, and developed an EHW chip capable of 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 [17]. 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].

TODO: Phylogeny, Ontogeny, and Epigenesis: Three Sources of Biological Inspiration for Softening Hardware

TODO: POEtic Tissue: An Integrated Architecture for Bio-Inspired Hardware

TODO: Routine duplication of post-2000 patented inventions by means of genetic programming

TODO: Toward automated design of industrial-strength analog circuits by means of genetic programming

TODO: Challenges of evolvable hardware: Past present and the path to a promising future [13]

TODO: MEH: Modular Evolvable Hardware for designing complex circuits

### 2.2.3 Arithmetic Circuits

TODO: MULTIPLE-VALUED COMBINATIONAL CIRCUITS SYNTHESISED USING EVOLVABLE HARDWARE APPROACH Genetic programing to add numbers via gate trees

TODO: Synthesis of Adder Circuit Using Cartesian Genetic Programming

TODO: Scalability problems of digital circuit evolution evolvability and efficient designs

## 2.3 Dynamic Problems

TODO: A little spiel about what dynamic are and why they are relevant

TODO: A self-organizing random immigrants genetic algorithm for dynamic optimization problems [35] Improves diversity and escaping from local optima, dynamic optimisation problem

TODO: A Genetic Algorithm Approach to Dynamic Job Shop Scheduling Problem. [26] Different sort of dynamic problem - varying input, constant fitness criteria

TODO: Designing evolutionary algorithms for dynamic optimization problems[3]

### 2.3.1 Fault Tolerance

TODO: Understanding inherent qualities of evolved circuits: Evolutionary history as a predictor of fault tolerance

Fault tolerance in regards to hardware design is defined as a systems capacity to remain functional despite component failure. This is usually in reference to post-fabrication faults which affect 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 do to environmental conditions or an aging device.

Genetic algorithms by their nature produce designs which are resilient in the face of change. More fragile designs are often mutated into obscurity whereas ones with natural fault tolerance endure. Amplifying the effects of this subtle pressure towards fault tolerant design is a mainstay of applied evolutionary hardware research.

TODO: Evolvable Hardware and its application to pattern recognition and fault-tolerant systems[16] learning an XOR circuit, discussion of 2 applications: pattern recognition, ditch tracer

TODO: Evolutionary techniques for fault Tolerance[36] Fault tolerant fitness function, fault tolerance by natural process, control for a robot evolved

TODO: On the Automatic Design of Robust Electronics Through Artificial Evolution Addresses problems of EHW only working at certain temperatures. Selection pressure for robustness added by executing fitness evaluation and different temperatures

One of the potential problems with fitness function driven exploration comes in the form of environment specific effects. Somefolk et al. explored the problem where the evolutionary process honed in to a solution which exploited temperature sensitive behaviours of a chip [38].

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 [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 fault 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 conventional genetic algorithm, along with tournament selection, and elitism J. 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 it.

TODO: Addressing the Metric Challenge: Evolved versus Traditional Fault Tolerant Circuits[12] Compairson peice

TODO: Evolviving Redundant Structures for Reliable Circuits[8] Novel redundancy structures, challenge to tune evolution to find these novel structures

TODO: Evolving Variability-Tolerant CMOS Designs[42] High failure rate in CMOS designs, GA vs CGP, unconventional designs with variability-tolerance

TODO: Implementation of a Power-aware Dynamic Fault Tolerant Mechanism on the Ubichip Platform[24] Built in self repair (BISR), dynamic fault tolerant system on the Ubichip platform

TODO: Machine Learing-based Anomaly Detection for Post-silicon Bug Diagnosis[7] Post-fab bug detection with ML

TODO: Self-healing router architecture for reliable network-on-chips[23] Redundancy based, rerouting

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 and protons causes huge damage.

Space technology 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. A. Sotica et al. propose using genetic algorithms to drive in-situ hardware reconfiguration to bypass faulty areas [33]. Experiments were performed on an FTPA 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 upto 350krad the system was usually able to recover functionality, and there were significant performance improvements over the unevolved comparison chip.

TODO: Design of self-repairing control circuit for brushless DC motor based on evolvable hardware[43] Radiation space damage, dynamic partial reconfiguration

TODO: Towards evolving fault tolerant biologically inspired hardware using evolutionary algorithms

TODO: Intelligent Fault Tolerance techniques

TODO: Requirement for extensive fault models to accurately inject faults

Fault tolerance in conventional hardware design requires redundancy in some form. Predictive preemptive instruction rescheduling on a different processor core [32] uses multi-core redundancy to mitigate the cost of faulty instruction execution. Cross-core redundancy is used by core salvaging [29] which shares execution pipeline resources between cores, in order to reroute execution pipelines around faulty components. This is similar in principal to StageWeb [11], 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 techniques do little to improve the performance of faulty architectural processor components and rely on the ability to route around them.

### 2.3.2 Dynamic Optimisation

It has been well established that flexible systems can provide huge boosts to, not only circuit efficiency, but also performance.

TODO: Configurable memory systems for embedded many-core processors[2] TODO: FPGA deep learning accelerators

# Chapter 3

# Project Execution

**A topic-specific chapter, of roughly 15 pages**

This chapter is intended to describe what you did: the goal is to explain the main activity or activities, of any type, which constituted your work during the project. The content is highly topic-specific, but for many projects it will make sense to split the chapter into two sections: one will discuss the design of something (e.g., some hardware or software, or an algorithm, or experiment), including any rationale or decisions made, and the other will discuss how this design was realised via some form of implementation.

This is, of course, far from ideal for *many* project topics. Some situations which clearly require a different approach include:

- In a project where asymptotic analysis of some algorithm is the goal, there is no real "design and implementation" in a traditional sense even though the activity of analysis is clearly within the remit of this chapter.
- In a project where analysis of some results is as major, or a more major goal than the implementation that produced them, it might be sensible to merge this chapter with the next one: the main activity is such that discussion of the results cannot be viewed separately.

Note that it is common to include evidence of "best practice" project management (e.g., use of version control, choice of programming language and so on). Rather than simply a rote list, make sure any such content is useful and/or informative in some way: for example, if there was a decision to be made then explain the trade-offs and implications involved.

The core of this project involves building a flexible evolvable hardware platform upon which novel genetic algorithm 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.

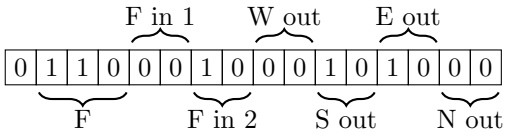
## 3.1 Project Management

The sourcecode is hosted online at `https://github.com/AlexDalt/MalleableHardwareAdder`, it's 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 reduced to evaluating various performance metrics of different and new evolvable hardware processes.

## 3.2 Design

The high level design choices have been made to improve development time and discern insights into evolutionary hardware with dynamic problems and the scaling issues involved.

TODO: talk about some problems common with GA design




| 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

### 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 [37] 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. 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 function $F$ is used as input instead. A description of the complete string can be found in Figure 3.1.

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, and 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).

Cells are defined from left to right, top to bottom on the FPGA. Therefore the binary strign 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 required by [37]. It allows concise expression of the FPGA. The order in which cells are defined also follows [37], and means any crossover occurring keeps the horizontal sections consistent with each other.

TODO: include mapping order diagram and potential crossover diagram

### 3.2.2 Problem Choice

The choice of problem is a key component in designing and configuring the 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. With a mind to hardware in a generic sense binary addition appeared to be an interesting 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 with weightings which can vary over time.

TODO: Talk about the complexity of the problem, the need to plan ahead, channels crossing over, individual simplicity but the need to chain together

### 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

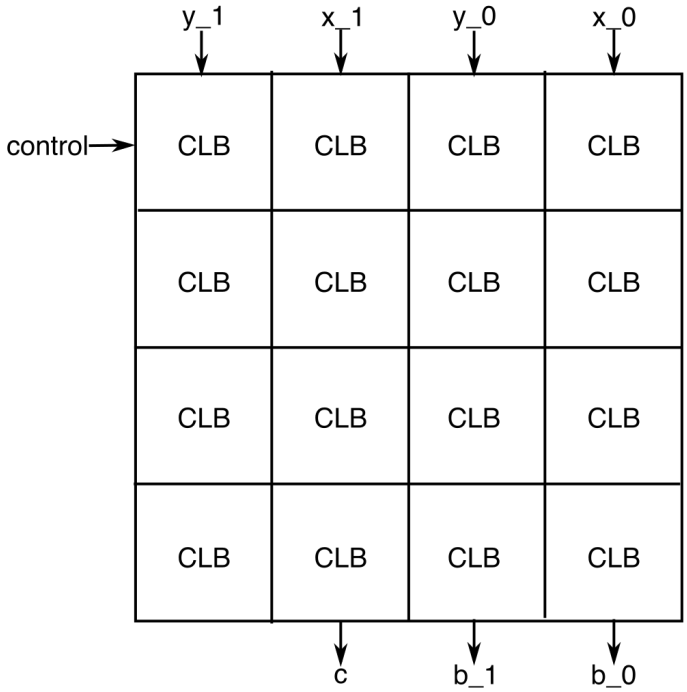



Figure 3.2: Diagram detailing how evaluation parameters are fed into the FPGA and answers read off for a 2-bit arithmetic problem

streamlined software FPGA simulator as the evaluation backend 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. When evaluating a binary addition or subtraction the FPGA is initialised to contain entirely undefined values (represented by a 2 internally), then the individual bits constituting the binary representation of the two numbers 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 the computation and storing the neccessary values in the output buffer associated with each neighbor. The tick tock cycle continues until the southern output of the bottom cells is no longer undefined and stabilises on a constant output; unless this doesn't happen within 64 iteration in which case the simulation halts. The output of the FPGA is read across the bottom row of cells. 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 arithmatic. Two 2-bit numbers, $x$ and $y$ are deconstructed into the constituent binary values where $x = x_0 + 2^{x_1}$ and $y = y_0 + 2^{y_1}$, these are made available to the top row of FPGA cells from most significant to least significant bit, 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 dude to the time taken for new values to propegate through the circuit. This leads to identical configurations which receive wildly different scores in two evaluations.

### 3.2.4 Fitness Landscape

TODO: evolution eye-view on ruggedness, redo this entire section

When tackling a genetic problem 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

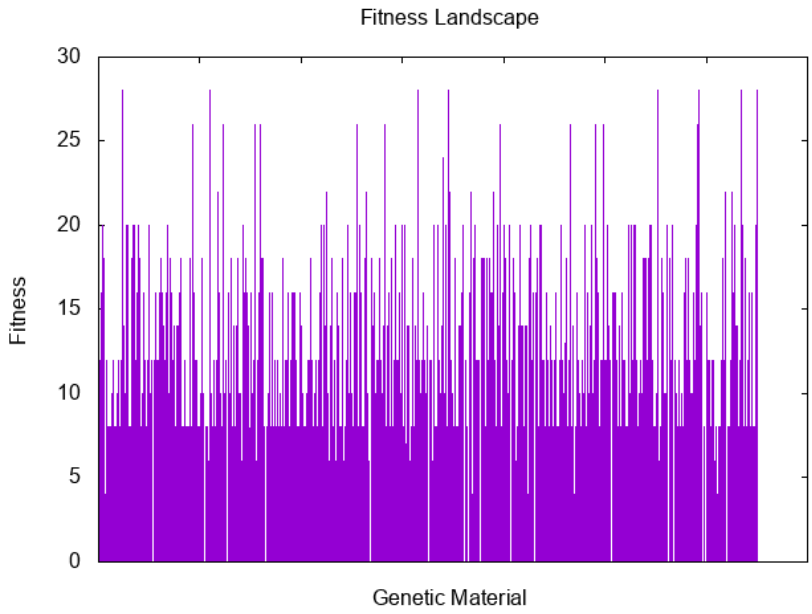



Figure 3.3: Sampled fitness landscape

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 timeframe. In order to gain an understanding for the cartography of the fitness landscape 13000 randomly generated configurations were evaluated and then ordered based their genetic material, the results are graphed in Figure 3.3.

TODO: find a reference talking about GAs being good when you have little understanding of underlying fitness landscape and topography, crazy hyper-dimentional shapes

In keeping with problems often tackled by genetic algorithms this landscape is discontinuous and random. Despite this however, the landscape is one which doesn't necessarily lend itself to easy searching via genetic algorithm. It is covered in local optima and the peaks are sharp, unpredictable, and not uniformly distributed. Therefore solutions will often be local optima, and successful solutions will be fragile, not taking much mutating to be cast back into the primordial soup.

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 ideal resting with a highly targeted genetic algorithm.

### 3.2.5 Genetic Algorithm

The basis for the genetic algorithm follows similar work done by Adrian Thompson [37], 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** TODO: why do we ignore stochastic selecton The primary selection method will be rank-based selection, which is by far the most popular selection method observed in preexisting work, however, tournament selection will also be explored.

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 the position within the ranking. By adding a "skew" by shifting the probability function to one closer resembling a quadratic one can emphasise the best performing individuals over the more balanced linear selection function.

$$P(x) = \frac{2}{n(n+1)}\left(\frac{1-s}{n}r^2 + sr + 1\right) \tag{3.1}$$

A single parameter change allows us to change the relative weightings of high scoring and low scoring individuals in a population via Equation 3.1. Given population size n, individual x, the individuals position in the ranking r, and skew s (between 0 and 1, where 1 is linear selection and 0 is quadratic) it generates a score between 1 and the population size plus 1. By varying the skew value I hope to gain insight into the impact of a more or less discriminatory selection mechanism.

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) solutions.

**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 highlights the need to employ some preservation mechanisms.

**Multi-objective Fitness Function** TODO: Reference forwards, early testing demonstrated the need for a diversity measurement, see page ...

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 correctness and size.

A small design is desirable as the smaller a design is 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.

Given the large volume of local optima and irregular peaks in the fitness landscape as demonstrated by Figure 3.3, it may be prudent to also incorporate some measure of diversity. [6] proposes a 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 deadends and discovers a working solution.

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

$$f(x) = w_{correct}f_{correct}(x) + w_{size}f_{size}(x) + w_{div}f_{div}(x) \tag{3.2}$$

Equation 3.2 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. 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, it should be more important to be correct than small, for example.

**Crossover** Early experiments will be conducted without any crossover mechanism, and therefore asexual reproduction (individuals will only have a single parents). But single point crossover will eventually be included and a range of crossover probabilities will be investigated.

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.

### 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 but also to reduce the size of the set of evaluation problems and 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 only the ones specified by each item in the set contained within the parasite. The parasite gains a score of $m - f_{add}(x) - f_{sub}(x)$ where $m$ is the maximum score possible ($bl$, where $b$ is the number of output bits for each problem and $l$ is the length of the set of problems contained within a parasite).

TODO: include parasite diagram

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

**Variable virulence** By varying the parasite virulence I hope to 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 3.3, 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].

$$m = \frac{2s}{v} - \frac{s^2}{v^2} \tag{3.3}$$

**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 you increase the scale of an evolvable hardware problem, the set of test cases grows at an exponential rate. I suggest 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. I primerily want to explore faults generated on a device as it is used, these faults are mainly caused by component wear out and unpredictable environmental effects, to this end I will extend the FPGA simulation 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 (which is 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 Darwinean 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.

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 backend 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.

TODO: add sticky fault state machine

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.

TODO: cite

### 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 modeling 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 the 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 3.4.

$$f(x) = \frac{w_{add} f_{add}(x) + w_{sub} f_{sub}(x)}{w_{add} + w_{sub}} + w_{size} f_{size}(x) + w_{div} f_{div}(x) \tag{3.4}$$

## 3.3 Implementation

All the flexibility described in the above design criteria is realised by modifying the headerfiles of my evolvable hardware sourcecode. 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 backend. 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 backend is contained in Appendix A and shows the key datastructures and functions exposed to the components performing evolution. The simulator can convert bitstrings of appropriate length into an FPGA of a given size, 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 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 relevent 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 cells `n_in`, `e_in`, `s_in`, and `w_in`, with the relevent data from each neighboring cell. For example, when assigning `n_in` the southern output (`s_out`) of the cell to the north is copied in TODO: include diagram of this happening. This occurs for all cells save 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 comes of 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 direction. Then each cell's `n_out`, `e_out`, `s_out`, and `e_out` is assigned in accordance with the cells internal mapping (eg. `n_out`, the buffer the cell to the north coppies for it's `s_in` value, could be directly coppied from the function output, or any of the $x$`_in` variables).

TODO: include diagram for this step too

Deploying this simulation gave subsecond evaluation times for an entire generation of 4x4 FPGAs. These performance benefits, whilst alienating us from the exploitation of strange undefined behaviour as experienced by Adrian Thompson[37] 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.

TODO: a bunch of relevent datastructure diagrams

### 3.3.2 Evolution

Beyond calls into the simulation backend 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.

TODO: talk about datastructures involved, individual, parasite etc. ind has space for the FPGA struct in it so it doesnt have to be reinstantiated throughout the process

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 labeled `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 coppied into it. The function then begins on an unending `while(true)` loop which conducts the evolutionary process until halted by the user. If coevolving the parasite population is shuffled and then the 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. If coevolution is not happening the function takes a pointer to a dummy parasite instead. When this function returns the array `Individual.eval` has been filled. `eval[0]` contains the measure of the individuals correctness (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 coppied 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 (if not coevolving, dummy) Parasite and a pointer to the host population populates the evaluation variables in both the Individual and the Parasite. If not coevolving this 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 correctness 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 ($bl$ where $b$ is the number of bits checked for each test, and $l$ is the length of the char array stored in the parasite) minus the score given to the host individual.

The function ind_distance calculates the hamming weight from one FPGA configuration to another. Two Individuals are passed in and converted into FPGAs 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 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 coppied 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 itterated over and, given an expected mutation rate defined by MUTATION and genetic material of length $i$, each bit in each individual is flipped with probability $\frac{1}{i}$MUTATION.

ind_prob implements Equation 3.1 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 $r$.

TODO: include graph of selection pressure for 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 correctness value and the correctness of the best case individual. We are not overly concerned with the total weighted fitness function as we are using that as a tool to better optimise for correctness; 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.

TODO: crossover

TODO: tournament selection

TODO: fancy diagrams

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.

TODO: complete function architecture diagram

### 3.3.3 Fault Implimentation

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.

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 500 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 Implimentation

Programatically or by user input the relative weightings of addition and subtraction can be shifted mid-execution.

TODO: more here

### 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 ?? 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 if test all 3 bits are correct the test is highlighted in green, if all 3 are wrong it is highlighted in red.

TODO: describe what the gui shows (cell breakdown)

### 3.3.6 Test suite

The aparatus to perform tests takes the form of a while loop encapsulating the evolutionary process, reading performance data and meddling with parameters 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 C). `TEST_SIZE` defines the number of samples to be generated, and therefor 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, and the average execution time.

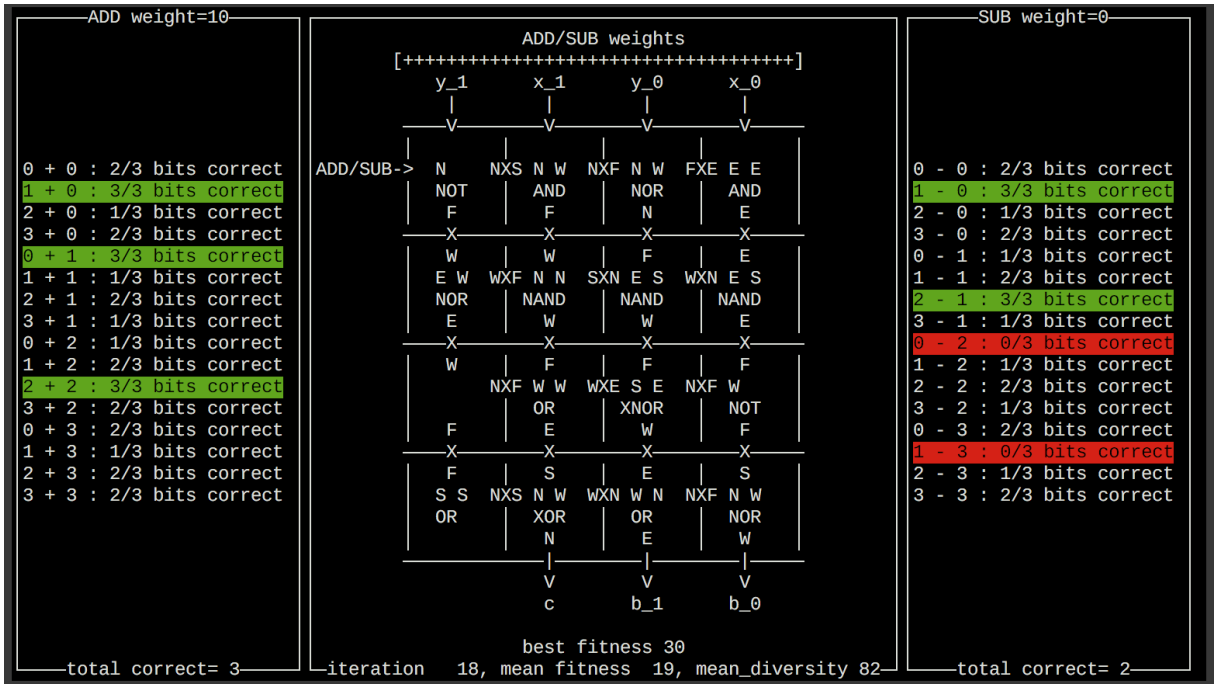



Figure 3.4: GUI screenshot

# Chapter 4

# Critical Evaluation

**A topic-specific chapter, of roughly 15 pages**

This chapter is intended to evaluate what you did. The content is highly topic-specific, but for many projects will have flavours of the following:

1. functional testing, including analysis and explanation of failure cases,
2. behavioural testing, often including analysis of any results that draw some form of conclusion wrt. the aims and objectives, and
3. evaluation of options and decisions within the project, and/or a comparison with alternatives.

This chapter often acts to differentiate project quality: even if the work completed is of a high technical quality, critical yet objective evaluation and comparison of the outcomes is crucial. In essence, the reader wants to learn something, so the worst examples amount to simple statements of fact (e.g., "graph X shows the result is Y"); the best examples are analytical and exploratory (e.g., "graph X shows the result is Y, which means Z; this contradicts [1], which may be because I use a different assumption"). As such, both positive *and* negative outcomes are valid *if* presented in a suitable manner.

Using the automated testing suite to repeatedly run experiments the data harvested for the next section was generated by averaging over 30 independant 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.

Much of the published work in genetic algorithm 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. This is a measure of how good the configuration is, throughout execution the fitness function simply acts as an incentive to improve this value.

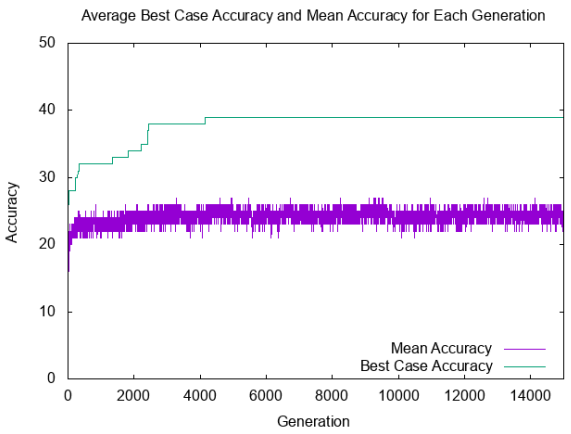
TODO: reference back a lot to other papers

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.

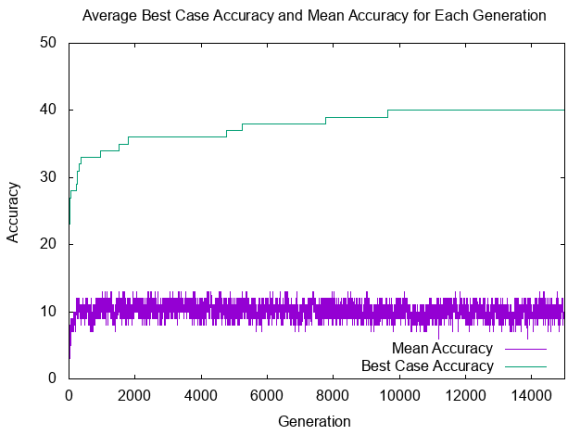
## 4.1 FPGA Simulator

The main metric by which to judge the quality of the FPGA simulation is the speed with which a configuration can be instantiated and evaluated.

TODO: actually this




(a)




(b)

| | Fitness Function | Perf. Runs (%) | Avg. Exec. Time (s) | Avg. Final Accuracy |
|---|---|---|---|---|
| (a) | Simple | 0 | 235 | 39 |
| (b) | Multi-Objective | 0 | 267 | 40 |

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 [37], (b) an extended multi-objective fitness function incorporating a measure of diversity with the diversity weight set to 40% of accuracy.

## 4.2 GA Parameter Tuning

All inital genetic algorithm parameters were set mirroring the parameter choice of [37]; a population size of 50, with elitism, linear rank-based selection, the probability of single point crossover occuring set to 0.7, and the expected mutations per offspring set as 2.7. However the problem solved in [37] 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 paper in the existing literiture which has taken the greatest pains to explain the experiment configuration and setup.

TODO: add more explicit parameters to figure

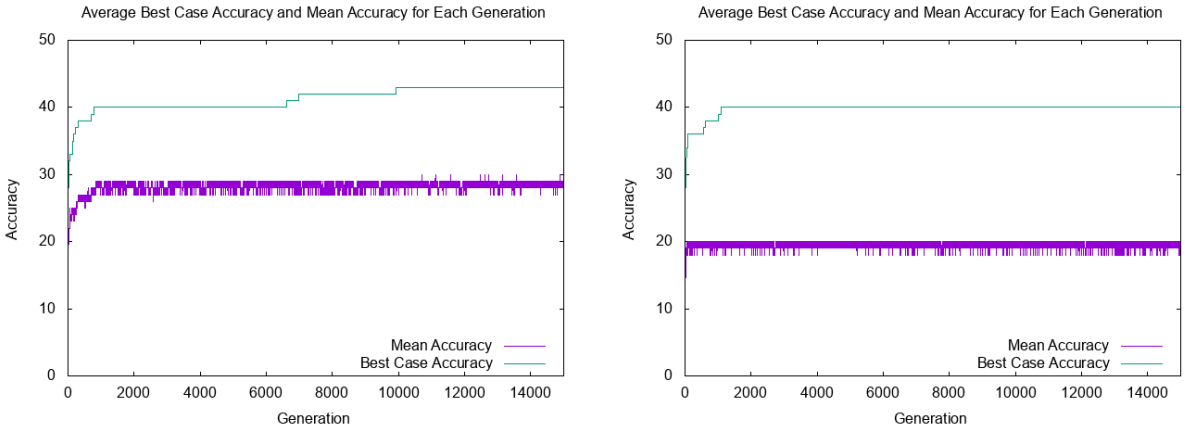
These parameters were evaluated and, as shown in Figure 4.1(a), the results of these initial tests are promising but show a great deal of room for improvement. Despite a healthy initial improvement in fitness, this quickly reached a plateau and across the duration of the tests no evolutionary run was capable of fomulating a perfect solution. Visual inspection of this initial test as it ran highlighed a huge lack of diversity in the population of solutions, every improvement in fitness was an itterative 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 incapabale 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 plumits.

TODO: include proof of plumiting diversity

The diversity in the population averaged (some value), which means that the average number of different components between an individual and the rest of the population was as little as 20. With an expected mutation rate of 2.7 this is indicative of a population completely dominated by a single (potentially bloated) solution as

### 4.2.1 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 reduce the probability of bloated solutions could provide the evolutionary pressure required to find a




(a) (b)

| | Fitness Function | Perf. Runs (%) | Avg. Exec. Time (s) | Avg. Final Accuracy |
|---|---|---|---|---|
| (a) | Simple | 0 | 2250 | 43 |
| (b) | Multi-Objective | 0 | 2857 | 40 |

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 [37], (b) an extended multi-objective fitness function incorporating a measure of diversity with the diversity weight set to 40% of accuracy.

perfect solution. In this situation the final size of a configuration is inconsiquential, 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 correctness 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) dipicts the results from this experiment. Averaged across repeated executions, there was a slight increase in execution time acompanised by a marginal improvement in the best case solution fitness after 15000 generations. However, a larger number of generations was 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 dissapointing 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 persuit of perfection; the intial trial in unencumbered with a fitness function atempting to maintain diversity, and so aggresively persues 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-idependently evolving subpopulations. They also had a mechanism which pruned the population of “dominated” individuals which would reduce the tendancy 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 feesable 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 fitness function (Figure 4.2(a)) performed the best, the streamlined persuit of accuracy combined with

a larger pool of people 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 the 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 itterative improvement. Population size will, however, be futher explored in subsection 4.2.6.

### 4.2.2 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 inherrently 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 moreso 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.

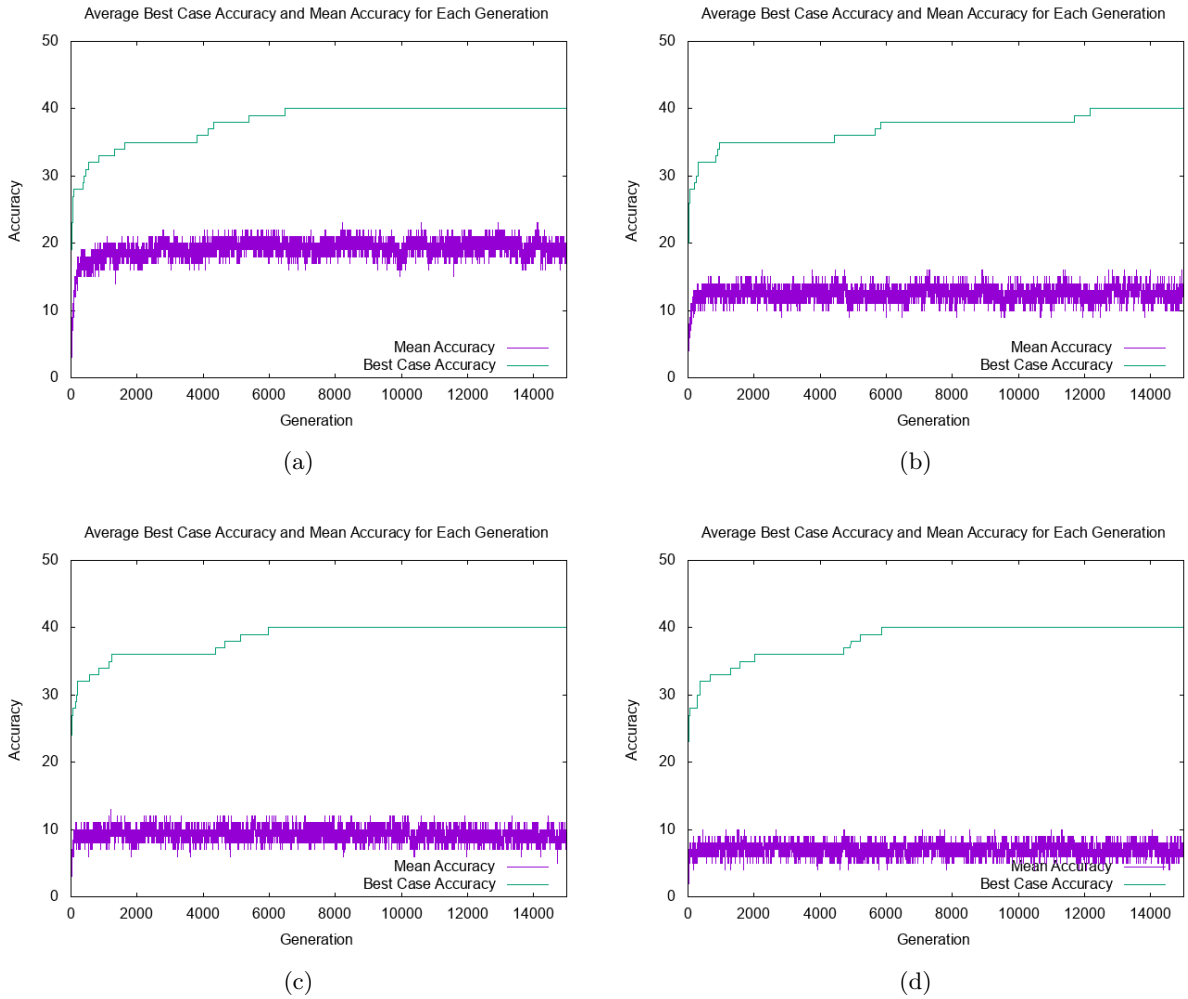
The experiments conducted in Figure 4.3 mostly agree with the findings in [37] with respect to the experimental results indicating and ideal mutation rate of around 2.7. However a slight accuracy increase in the best case individual is observed when shifting the mutation rate from 2.7 (Figure 4.1(b)) to 3 (Figure 4.3(c)). This is another example of domain specific experimental tuning for the binary arthmatic 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 aggresive 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 slighly 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.

### 4.2.3 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 [37]). Here the use of a variable skewed rank selection is emplyed 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) upto 40 (80% the population size). The smaller tournament size will discriminate less against lower performing individuals, and the larger the tournament




(a) (b)

(c) (d)

| | Mutation Rate | Perf. Runs (%) | Avg. Exec. Time (s) | Avg. Final Fitness |
|---|---|---|---|---|
| (a) | 1 | 0 | 245 | 40 |
| (b) | 2 | 0 | 264 | 40 |
| Figure 4.1(b) | 2.7 | 0 | 267 | 40 |
| (c) | 3 | 0 | 363 | 40 |
| (d) | 4 | 0 | 368 | 40 |

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.

the fewer poorly performing configurations will be selected for the next generation. This interplay could dramatically increase performance, exploiting a balance (TODO: much like cite cite cite) to drastically improve performance.

The more aggresive 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)) persue 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 uniformally at random for a tournament, but within a tournament only the most fit individual wins. This interplay between minimally and maximally discrimanatory 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 maintainance. 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.

### 4.2.4 Crossover

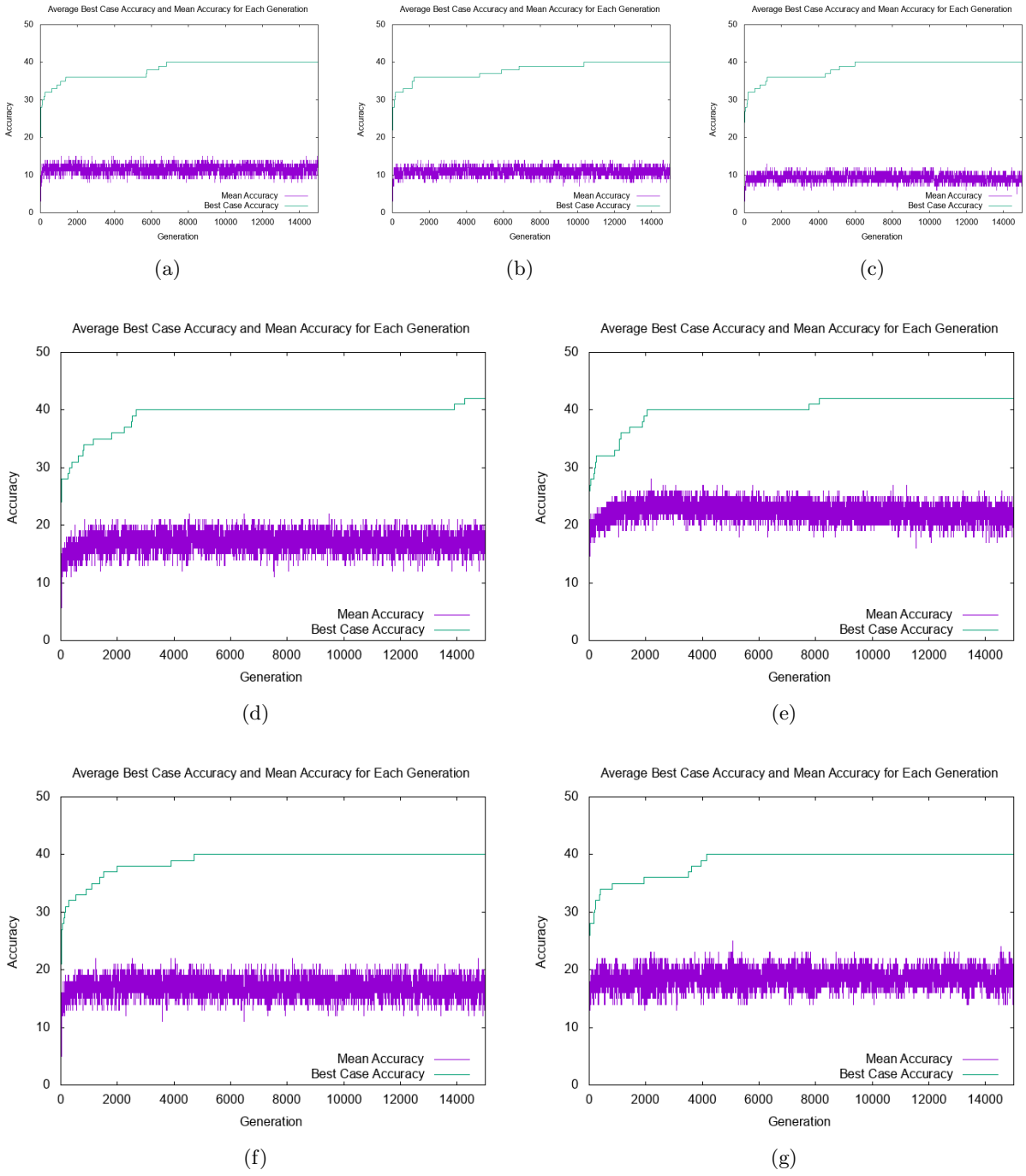
Crossover works well in this context, the nature of the mapping between genotype and phenotype means any crossover can provide benifits to both mating pairs and inject some much needed diversity into the ecosystem. Three additional trial runs were executed, each at a different crossover probability rate; 0, 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 an performing or not with an equal probability. An explination 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 uncompatible 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.

### 4.2.5 Elitism

With a mutation rate as high as shown here the effect of elitism is hightened. 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 arithmatic 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.

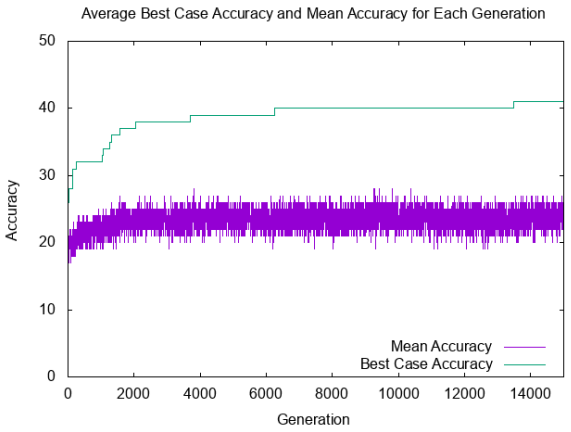
Clearly, as Figure 4.6 demonstrates removing elitism is crippling. The population rarely, if ever, performs better than an accuracy of 28. From here the any path to a better viable solution is insurmountable, crippled by an aggresive mutation rate, which explored the search space with such vigor that no individuals performing well can survive due to their inherrent fragility. This dissagrees with (TODO: cite the natural fault tollerence) which claims evolution naturally discoveres solutions resistant in the face of change; for binary arithmatic the solutions are too fragile to have such a resistance and require



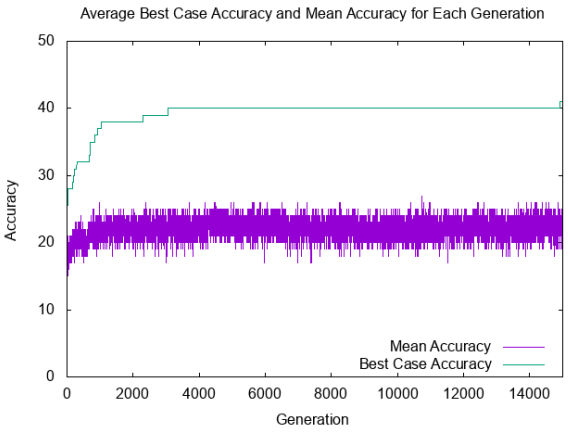

| | Selection Method | Perf. Runs (%) | Avg. Execution Time (s) | Avg. Final Fitness |
|---|---|---|---|---|
| (a) | Rank (Skew 0.0) | 0 | 304 | 40 |
| (b) | Rank (Skew 0.5) | 0 | 415 | 40 |
| (c) | Rank (Skew 1.0) | 0 | 363 | 40 |
| (d) | Tournament (Size 10) | 0 | 264 | 42 |
| (e) | Tournament (Size 20) | 13 | 231 | 42 |
| (f) | Tournament (Size 30) | 3 | 260 | 40 |
| (g) | Tournament (Size 40) | 0 | 244 | 40 |

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.

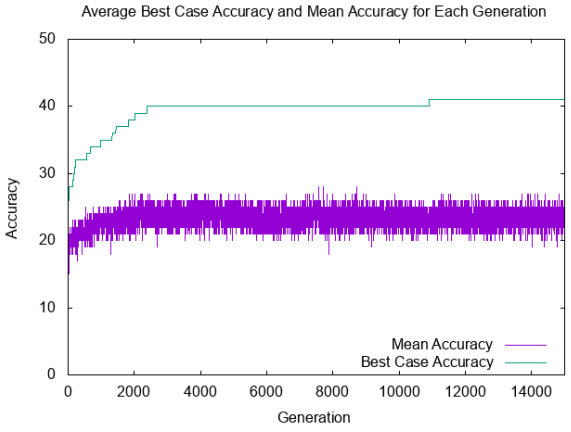
[1](c) duplicated from Figure 4.3(c) for better direct visual comparison




(a)



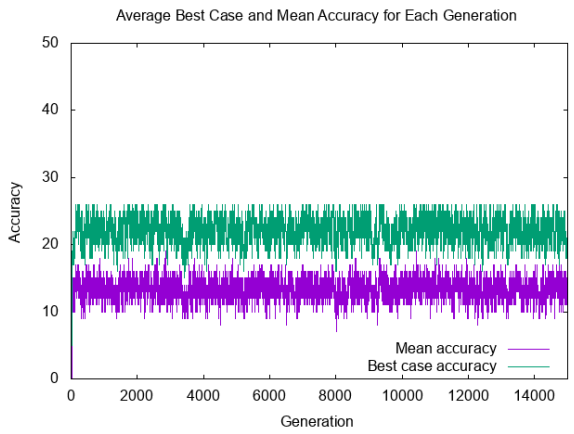

(b)




(c)

| | Crossover Prob. | Perf. Runs (%) | Avg. Exec. Time (s) | Avg. Final Fitness |
|---|---|---|---|---|
| (a) | 0.0 | 10 | 243 | 41 |
| (b) | 0.5 | 3 | 245 | 41 |
| Figure 4.4(e) | 0.7 | 13 | 231 | 42 |
| (c) | 1.0 | 3 | 249 | 41 |

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. Final Fitness |
|---|---|---|
| 0 | 281 | 19 |

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.

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.

### 4.2.6 Tuning Population Size

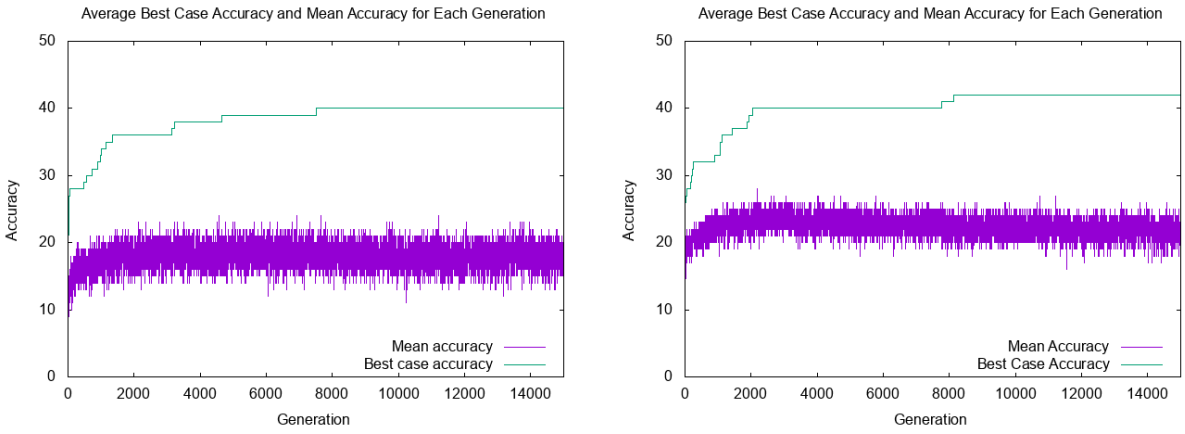
With the desire to cultivate a diverse population at the forefront of the parameter selection reasoning enlarging the population would improve the spread of solutions, and allow the evolutionary pressure on diversity to simultaneously mature and develop a bredth of solutions. Obviously increasing the size of the population has objective benifits in a statistical sense, by casting the net wider there is a higher chance to stumble accross 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 acordingly, 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 evalutation function measuring population diversity counts for an insignificant minority of the execution time. The most aggressive increases in time are due to the aditional population members which need to be instantiated as FPGAs and fed the test data. This evidence against the claims made in (TODO: ref earlier scaling claims) demonstrate that the scaling issues are localised to FPGA evaluation.

For all population sizes achieving a best-case accuracy of 36 occured in a similar amount of time, however the larger populations were able to maintain this ascent up the evolutionary ladder better. This sudden insummountable evolutionary barrier for the smaller population sizes implies that the smaller populations were unable to cultivate a bredth of solutions early in execution and therefore encountered an evolutionary deadend, temporarily halting improvements.

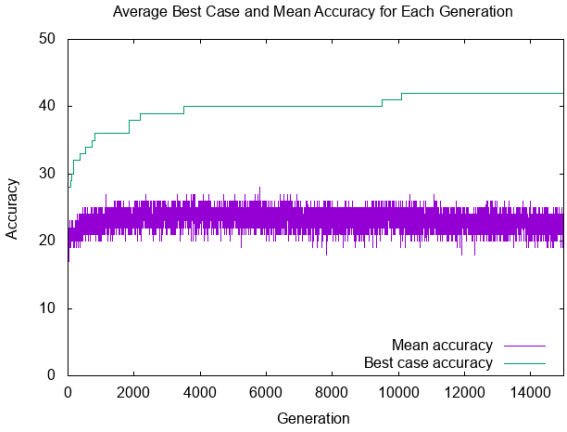
Moving forward a population size of 50 was selected. Clearly the population size of 200 is the best candidate, but the increase in persentage of perfect solutions is almost linear with the added execution time. In order to maintain momentum at this stage in the project the quicker execution was chosen,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



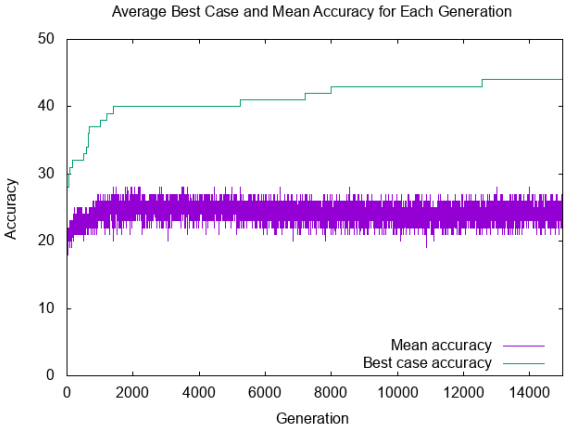

(a)




(b)




(c)




(d)

| | Population Size | Perfect Solutions (%) | Avg. Execution Time (s) | Avg. Final Fitness |
|---|---|---|---|---|
| (a) | 25 | 0 | 126 | 40 |
| (b) | 50 | 13 | 231 | 42 |
| (c) | 100 | 17 | 457 | 42 |
| (d) | 200 | 57 | 955 | 44 |

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)(duplicated) population size 50, (c) population size 100, and (d) population size 200.

of perfect solutions by a factor of 4.

### 4.2.7 Tuning Diversity

One aspect of the evolutionary proccess 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 suprime. If set too low we reenter 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 arithmatic 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 completly focussed on diversity.

TODO: prune away graphs placed on the same page

Figure 4.8 demonstrates the symptoms indicating existing on either side of balance of power within the fitness fuction between accuracy and diversity. With a weighting of 0 (Figure 4.8(a)) the population hits two evolutionary dead ends, one at accuracy 34, which it eventually manages to summount and another at accuracy 40 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 averge ending accuracy of 41, and all improvements to fitness occured 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 **??**.

The reduced effectiveness of diversity at higher weightings is somewhat at odds with the point of view presented by (TODO: cite the multi-objective). As previously mentioned however their implimentation was coupled with an effective pruning mechanism to dissuade mindless diversification for the sake of diversification. Also the context each multi-objective fitness function differs, evolutionary hardware and evolutionary programming are different animals each with distinct problems. Namely (TODO: cite) 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 spacially 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.

### 4.2.8 Coevolution

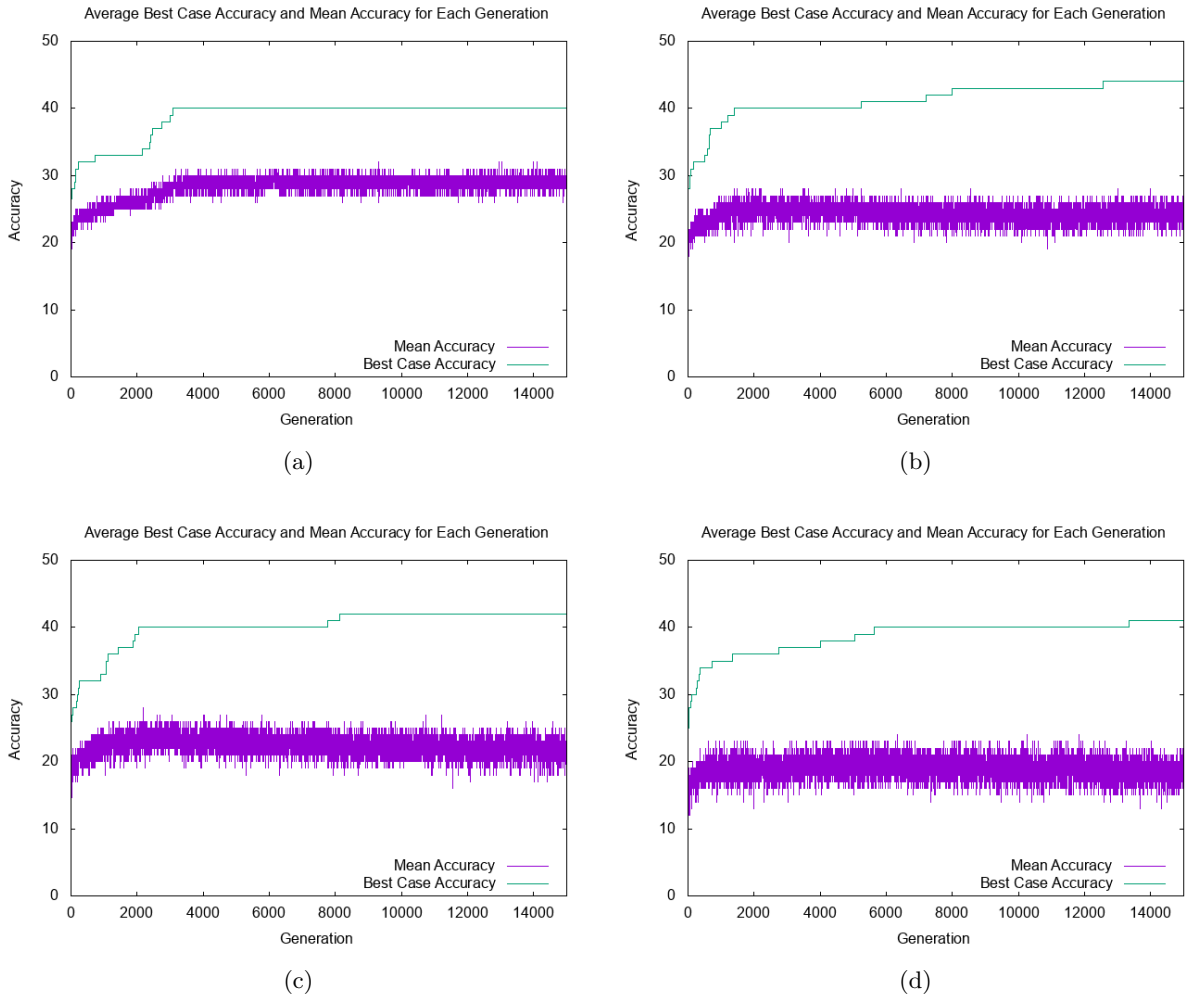
Can we exchange the full problem testing system used thus far for a coevolved parasite population of evolutionarily targeted problems? In order to answer that question experiments emplying coevolution were conducted on the evolutionary system developed thus far. Experiments were conducted with a parasite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

TODO: Talk about all the coevolutionary problems from Bullocks paper

In a coevolutionary system there a series of problems that can disturb the gentle ballance 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 rehiterates the findings in subsection 4.2.5 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 4.5.

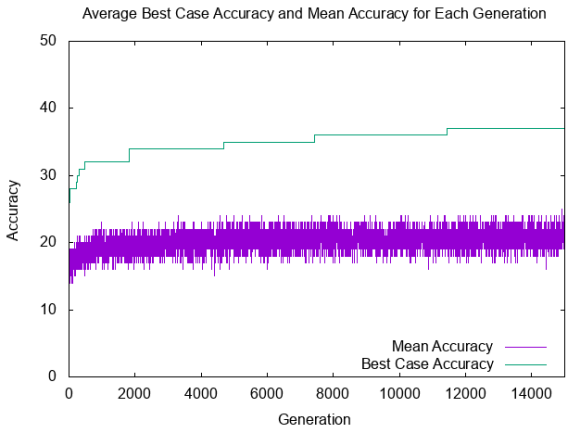
In all the experiments with elitism activated after a certain amount of time the average population fitness reaches a plateau and all improvements seem to come from the highest performing individual(s) with little impact in the overall population. Although similar to tests not using a coevolutionary system this behaviour could be indicative of a disengaged population, one in which the parasite population



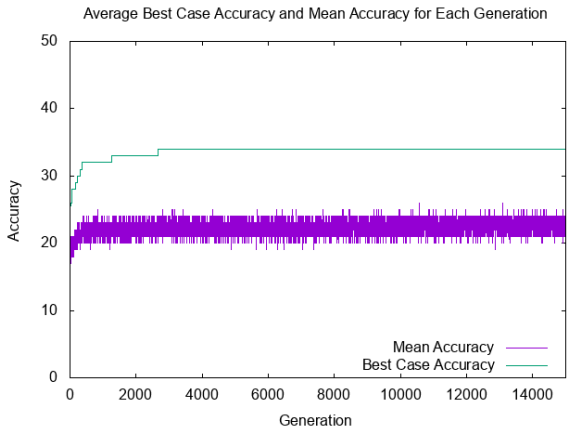

| | Diversity Weighting (% of Accuracy) | Perf. Runs (%) | Avg. Execution Time (s) | Avg. Final Fi |
|---|---|---|---|---|
| (a) | 0 | 16 | 235 | 40 |
| (b) | 20 | 23 | 238 | 44 |
| (c) | 40 | 13 | 231 | 42 |
| (d) | 60 | 10 | 247 | 41 |

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.

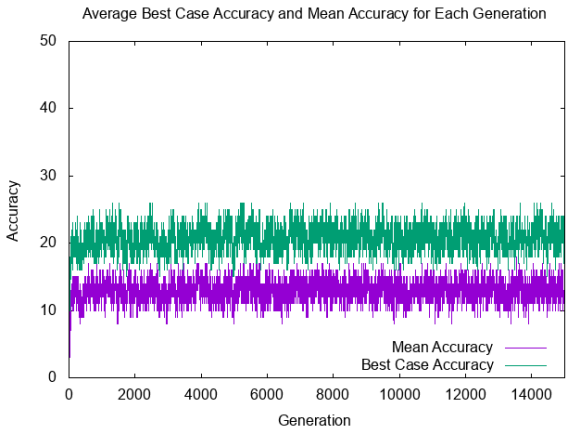
(a)




(b)




(c)




(d)

| | Parasite Size | Perf. Runs (%) | Avg. Execution Time (s) | Avg. Final Fitness |
|---|---|---|---|---|
| (a) | | | | |
| (b) | 16 | 0 | 605 | 37 |
| (c) | 32 | 0 | 703 | 34 |
| (d) | 16 | 0 | 535 | 24 |

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.

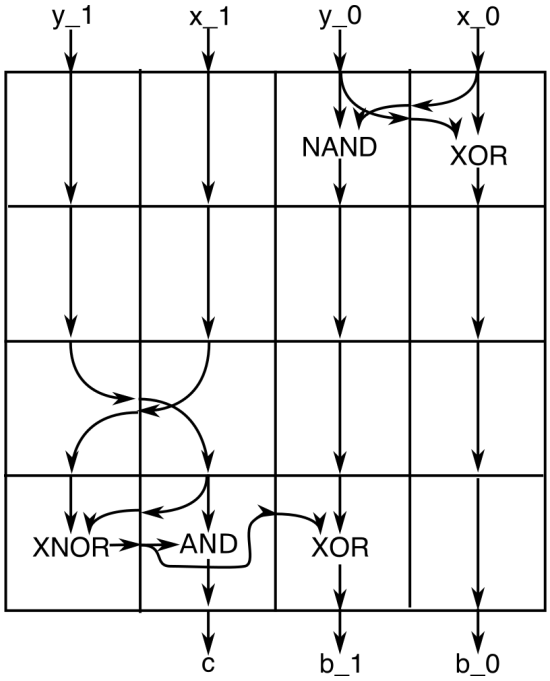



Figure 4.10: Successful 2-bit binary addition evolved hardware

becomes too aggresive and becomes unbeatable, no longer discriminating in the host population between FPGA configurations.

TODO: Change the virulence

### 4.2.9 Evolved 2-bit addition hardware

By now we have a relatively well performing evolutionary hardware system, tailored directly for the binary arithmatic problem. With this we can explore a range of uses within the domain of dynamic problems.

Talk about all the cool stuff, success rate etc analyise the given circuit beyond simple correctness. Diagram clearly one produced via crossover

## 4.3 Fault tollerance

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

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

### 4.3.1 Simple Fault Recovery

By preloading the genetic algorithm with a hand designed 2-bit adder and then simulating a highly targeted critical fault withing the function of a CLB insight can be gained into the capcity 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.

### 4.3.2 “Sticky” Fault Mitigation

Evolutionary hardware can be used to recover from otherwise devastating faults. In the example below the outputs of the operation F within the CLB marked in red were clamped to a value representing an undefined output. Starting with the ideal configuration on the left and activating the fault, the genetic algorithm recognised the current solution as suboptimal and generated the configuration on the right.

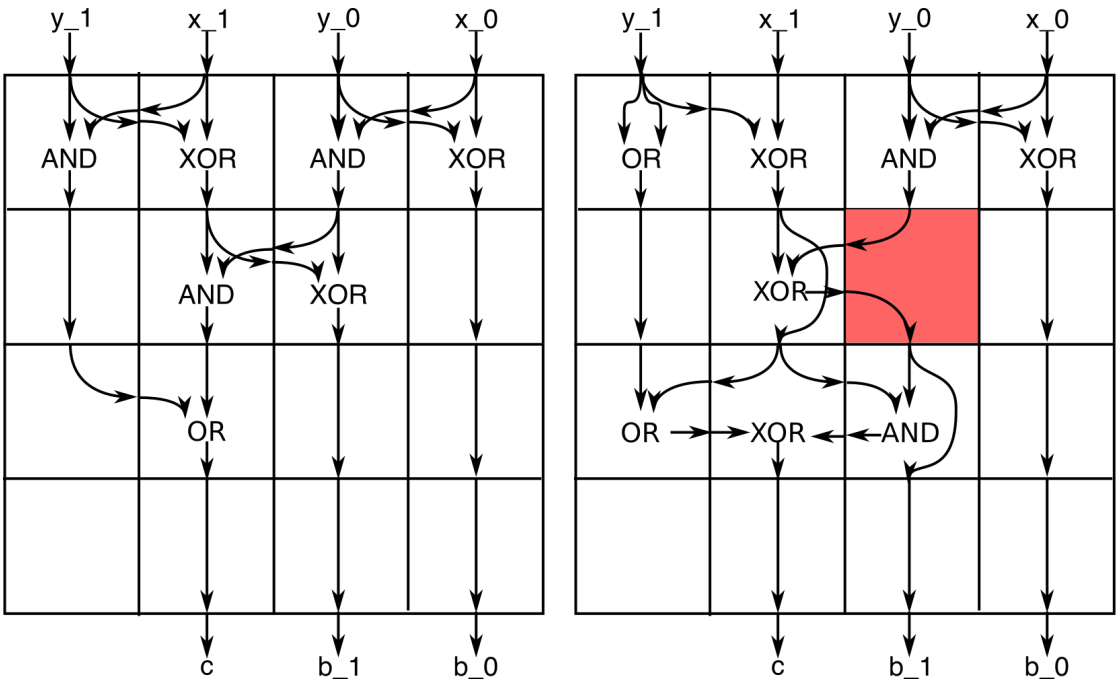



Figure 4.11: Fault recovery

Simulating a sticky fault by activating and deactivating a fault every 500 generations results in the fitness graph above on the right. When the fault is activated it has an adverse affect on fitness, but a solution is found which works when the fault is both active and inactive.

Sticky faults improve diversity and therefore improve training, maybe

Different from normal Fault tollerance - this is the sticky fault model to find a design okay for all, instead the evolution happens when the fault is detected to recover

## 4.4 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.

## 4.5 Scaling

In an effort to improve how an evolvable hardware system scales observing how the dynamics of a coevolutionary system vary as the number of parasites fluctuates could indicate a potential mechanism to reduce the number of test cases required per evaluation and therefore improve evaluation time.

Evaluating the execution time and correctness of the solutions generated by the genetic algorithm as we reduce the size of the parasite should quantify the feesability of this idea.

Scafolding vs non-scafolding, coevolution to improve scalability of finess evaulation (tests grow expontentiallly as input bits increase)

Table of timings

The trouble with using coevolition to improve scaling is to stop elitism every individual has to be tested against the whole suite of problems, and without elitism the entire system breaks down. A high mutation rate is required to effectivly persue the solution. Really bloody bad lol.

## 4.6 Evolution good

better than bruteforce mention things set out to do and address challenges

| FPGA Size (Width x Height) | Full test | Coevolve (Size 16) |
|---|---|---|
| 2x4 | | |
| 4x4 | 238 | 535 |
| 6x4 | | |
| 8x4 | | |

Figure 4.12: Execution time (s) for genetic algorithm opperating on different FPGA sizes with a full evaluation or a coevolved parasite population (without elitism).

## 4.7 Evolution bad

Intermittent leap from primordeal soup

# Chapter 5

# Conclusion

**A compulsory chapter, of roughly 5 pages** The concluding chapter of a dissertation is often underutilised because it is too often left too close to the deadline: it is important to allocation enough attention. Ideally, the chapter will consist of three parts:

1. (Re)summarise the main contributions and achievements, in essence summing up the content.
2. Clearly state the current project status (e.g., "X is working, Y is not") and evaluate what has been achieved with respect to the initial aims and objectives (e.g., "I completed aim X outlined previously, the evidence for this is within Chapter Y"). There is no problem including aims which were not completed, but it is important to evaluate and/or justify why this is the case.
3. Outline any open problems or future plans. Rather than treat this only as an exercise in what you *could* have done given more time, try to focus on any unexplored options or interesting outcomes (e.g., "my experiment for X gave counter-intuitive results, this could be because Y and would form an interesting area for further study" or "users found feature Z of my software difficult to use, which is obvious in hindsight but not during at design stage; to resolve this, I could clearly apply the technique of Smith [7]").

One can pre-load a chip with a working configuration, and when the system detects a fault which hurts performace, or space for optimisation the design is itterated over. Factoring in user data and contextual runtime information.

## 5.1 Summary of Achievments

## 5.2 Project state

Completed initial aims

## 5.3 Further work

Malleable HW, alternate ML algorithms, change mutation step to be context specific (translation/rotation of cells)

Be aggressivly critical, cannot be practical until this this and this is overcome talk about upload times

# Bibliography


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

[2] Daniel Bates, Alex Chadwick, and Robert Mullins. Configurable memory systems for embedded many-core processors. *CoRR*, abs/1601.00894, 2016.

[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] A. DeOrio, Q. Li, M. Burgess, and V. Bertacco. Machine learning-based anomaly detection for post-silicon bug diagnosis. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 491–496, March 2013.

[8] A. Djupdal and P. C. Haddow. Evolving redundant structures for reliable circuits - lessons learned. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 455–462, Aug 2007.

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

[10] 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.

[11] 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.

[12] 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.

[13] Pauline C. Haddow and Andy M. Tyrrell. Challenges of evolvable hardware: past, present and the path to a promising future. *Genetic Programming and Evolvable Machines*, 12(3):183–215, Sep 2011.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[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] K. Khalil, O. Eldash, and M. Bayoumi. Self-healing router architecture for reliable network-on-chips. In *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 330–333, Dec 2017.

[24] Kotaro Kobayashi, Juan Manuel Moreno, and Jordi Madrenas. Implementation of a power-aware dynamic fault tolerant mechanism on the ubichip platform. In Gianluca Tempesti, Andy M. Tyrrell, and Julian F. Miller, editors, *Evolvable Systems: From Biology to Hardware*, pages 299–309, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

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

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

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Adrian Thompson. On the automatic design of robust electronics through artificial evolution. In Moshe Sipper, Daniel Mange, and Andrés Pérez-Uribe, editors, *Evolvable Systems: From Biology to Hardware*, pages 13–24, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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 0
#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
 *              - the least significant 2 bits define the value pushed to n_out (F,I
 *              - 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,EA
 *              - the next 2 define where the second input for F comes from (NORTH,I
 *              - the next 3 define the function F performs (OFF,NOT,OR,AND,NAND,NOR
 *              - 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 iteration, FPGA fpga, int most_fit, int mean_fit, int mean_div, in

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 "simulator.h"

#define POP_SIZE 400
#define MUTATION 0.5f
#define SIZE_WEIGHT 0
#define DIVERSITY_WEIGHT 4
#define ELITISM 1
#define FITNESS_WEIGHT 10
#define COEVOLVE 0
#define STICKY 0
#define LOG 1
#define PROB_SKEW 0.0f //between 0 or 1, 1 is linear
#define VIRULENCE 1.0f
#define PARASITE_SIZE 16

int add_weight, sub_weight;

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

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

# Appendix C

# Experimental Results

TODO: some meaty tables here