A Hybrid Genetic Programming-Particle Swarm Approach for Designing Trading Strategies in Software and Hardware

An MEng undergraduate dissertation

United Kingdom, 2012–2013

Created by

Andreea-Ingrid Funie

Supervised by

Prof. Wayne Luk

Prof. Mark Salmon

Department of Computing (Masters in Artificial Intelligence)
Imperial College London

2013

Abstract

The aim of this project is to investigate the design and implementation of high frequency trading rules using genetic programming and swarm intelligence. The approach taken was to design, build and test two different versions of our basic approach: a genetic program which digs out relevant trading signal information from tick level Foreign Exchange market data and an adaption of this genetic program which contains a particle swarm optimisation on top of the original algorithm.

Because of their nature, evolutionary algorithms are generally considered to be slow. After observing that the fitness evaluation part of the algorithm is where the majority of the computation time is absorbed, we reduced this CPU original time by implementing an alternative version making use of the Field Programmable Gate Arrays (FPGAs). Thus, we show in the thesis how we implemented the trading algorithms on different architectures, while also showing the interaction between the CPU and FPGAs.

The application of the combination of genetic programming and swarm intelligence within a hybrid CPU/FPGA environment is as far as we know novel and has not appeared in the computational or financial literatures.

We obtained good robustness results and found significant profitability after performing a range of statistical tests and we concluded that our approach is reliable under stable market conditions and it does not provide spurious signals in the trading rules where there does not to be relevant information in the market data. We have also shown that the degree of predictability and hence profitability has decreased from 2003/4 to 2008 which is consistent with the rapid growth in Algorithmic Trading in Foreign Exchange Markets.

Acknowledgements

I would like to thank my supervisor, Professor Wayne Luk for his extended contributions and decision to send me on a Maxeler training day which gave me the best opportunity to quickly understand the tools I used. I further extend my gratitude to Professor Mark Salmon, for providing the inspiration for this project and for the innumerable suggestions he has given me through the course of the project. Also, I would like to thank both of them for their countless advices and hours spent on reading my dissertation drafts.

My thanks go out to the PhD students Maciej Kurek and Xinyu Niu for their quick explanations and suggestions to improve my approach to the hardware acceleration implementation part of the project.

I owe special gratitude to my family for teaching me not to avoid challenges. Without their support I would have not been able to pursue my dreams and to be the person I am today.

Last, but not least I want to thank to my friends for their unwavering support and encouragement during my years at Imperial.

This last months were challenging and I could not have made it without you!

Contents

Contents **iii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Idea . . . 2
1.3 Report Structure and Contributions . . . 3

2 Background Reading **5**
2.1 Financial Background . . . 5
2.1.1 Foreign Exchange Data . . . 5
2.1.2 Trading . . . 7
2.1.2.1 Technical indicators . . . 7
2.2 Machine learning techniques . . . 7
2.2.1 Genetic Algorithms . . . 8
2.2.2 Evolving from Genetic Algorithms to Genetic Programming . . . 9
2.2.3 Swarm intelligence . . . 9
2.2.3.1 Particle swarm optimisation . . . 10
2.3 Numerical Tests . . . 10
2.3.1 One sample T-Statistics . . . 10
2.3.2 Anatolyev-Gerko test: Economic significance . . . 11
2.4 Hardware acceleration . . . 12
2.4.1 Field-Programmable-Gate-Arrays . . . 12
2.4.2 Maxeler toolchain . . . 13
2.4.3 Hardware Platform . . . 14
2.4.4 Floating Point vs Fixed Point Precision . . . 15
2.5 State-of-the-Art . . . 16
2.6 Problem identification and proposed solution . . . 18
2.7 Summary . . . 19

3 Trading Algorithm in Software **21**
3.1 Genetic Algorithm approach . . . 22
3.2 Genetic Programming . . . 24
3.2.1 Design . . . 24

3.2.1.1 Terminals and Functions 25
3.2.1.2 Initialisation 26
3.2.1.3 Genetic operators 28
3.2.1.4 Fitness Evaluation 29
3.2.1.5 Selection 31
3.2.1.6 Tree structure execution and memory 32
3.3 Particle swarm optimisation 33
3.3.1 Approach 37
3.4 Implementation Details 38
3.4.1 C++ implementation 38
3.4.2 Market Data Parser 39
3.4.3 Running the statistical and robustness tests 39
3.5 Summary 40

4 Trading Algorithm in Hardware **43**
4.1 FPGA Design 44
4.1.1 Design Specifics 44
4.1.2 Device-Independent Analysis 46
4.2 FPGA Implementation 47
4.3 Performance Evaluation 50
4.3.1 Double/Single Precisions vs Fixed Point Resource Usages 50
4.3.1.1 Double Precision floating point solution on FPGA . . . 50
4.3.2 Single Precision floating point solution on FPGA 50
4.3.3 Fixed Point precision solution 51
4.3.4 Acceleration Measurements 52
4.4 Challenges and Further Improvements 54
4.4.1 Challenges 54
4.4.2 Further Improvements 54
4.5 Summary 58

5 Testing for Robustness **59**
5.1 Robustness 59
5.2 Individual testing 61
5.3 Aggregate testing 64
5.4 Rejection-Acceptance testing 65
5.5 Algorithm behaviour in noisy data presence 68
5.5.1 Results 69
5.6 Summary 70

6 Further Statistical Results **71**
6.1 GP vs Hybrid GP/PSO 71
6.1.1 Individual Returns comparison between GP and GP/PSO 72
6.1.2 T-statistics tests 73
6.1.3 Anatolyev-Gerko tests: Economic significance 76

6.2 Summary 78

7 Conclusions and Future Work **79**
7.1 Conclusions 79
7.2 Future Work 80

Appendix A **83**
.1 Technical Indicators 83
.2 FPGA specifics 86

List of Figures **87**

References **89**

Chapter 1

Introduction

1.1 Motivation

The advent of high frequency trading in electronic markets calls for new trading strategies and the management of new risks. There are two elements driving trading, the traditional economic forces of demand and supply as captured by the order book of a market as well as forces that arise simply from within the structure of the market itself as liquidity moves to balance the structure of risks in the market. Although it is much more straightforward to analyse simple mechanical trading rules, researchers have made some progress in looking at more complex, pattern-based rules: looked for "head-and-shoulders" patterns in currency markets, while continued their study in the context of equity markets [24].

Many studies have been made regarding possible ways of discovering successful trading strategies. Some of the machine learning related research present different solutions related to genetic algorithms, neural networks, particle swarm optimisation or decision trees, but all of the enumerated methods are very computationally expensive, thus limiting the testing and evaluation process. Also, no one found a strategy which has a good stable performance under many different market conditions, thus the need for real- time adaptation to market characteristics. This adaptation implies learning aspects added to the original machine learning algorithm, and for the adapt- tation to perform at its best it requires frequent market conditions feedback achieving its maximum performance at real-time evaluation. Thus the need of a more powerful speed optimisation comes into place and the conclusion that the advent of machine-based trading algorithms is due in no small part to the capacity to analyse big streams of data in real time using advanced hardware and software.

Recently, hardware acceleration tools provided by companies like Maxeler Technologies help big investment firms accelerate their trading strategies making use of reconfigurable hardware. For example, run-time reconfig- uration for a reconfigurable algorithmic trading engine has been shown to provide flexibility in algorithm design, enabling implementations to react to changes in market conditions rapidly. We know

that hardware acceleration is a useful optimisation added for trading, and this has been shown for financial market simulation as well.

With these in mind, the idea behind my research is to seek profitable trading patterns, or reliable predictive structures that can be employed with confidence in the market in real time, making use of artificial intelligence optimisations, hardware acceleration and pattern recognition techniques such as genetic programming.

We will test our approach on Foreign Exchange data and one important aspect we will keep in mind when interpreting our results is whether some patterns work in some market conditions only (different market conditions lead to different regimes in financial terms).

1.2 Idea

We have built an evolutionary hybrid genetic algorithm which use aspects of swarm intelligence (particle swarm optimisation) and seeks reliable and profitable trading patterns that can be further used in the trading strategies.

By their nature, evolutionary methods are quite time-consuming and thus, to reduce execution time and to be able to dig out for more advanced trading rules we have used hardware acceleration techniques which allowed us to obtain significant computation speedup.

Although we followed the route of data mining, we tried to analyse what happens when we get negative results as well, because we think that some of the events might happen for a reason. Furthermore we tested our program for robustness, profitability, reliability and predictability, aspects which are very important for the employment of the discovered trading rules in the real market trading.

The project was both challenging and interesting and the application of the combination of genetic programming and swarm intelligence within a hybrid CPU/FPGA environment is as far as we know novel and has not appeared in the computational or financial literatures. In an industry where every millisecond of latency reduction can have a considerable impact on profit, it is worthwhile to examine alternative approaches that may be used to speedup the computation time required to accomplish the objective of our project.

From a regulator's point of view, our project will be beneficial because it helps them identify any possible market behaviour which can appear under different market conditions (regimes). The regulators can give further importance to the research aspect that they could gain from the project, by being able to identify trading rules which could cause significant market changes (e.g: the GBP price to go up all of a sudden, despite non obvious economical reasons). Also, the speed with which the regulators obtain feedback with regards to the market trading rules that appeared is very important since this can further help them avoid significant risk taking or change of markets behaviour.

This project is also important since, from a financial institution point of view they would be able to use it to identify useful patterns, given the historical analysis, that can help the company maximise its profits from trading.

It is also important to note that our approach has produced just a working prototype of a version that could eventually, with the power of hardware acceleration, be able to offer feedback in real-time about the high frequency trading market behaviour for both financial institutions and regulators.

1.3 Report Structure and Contributions

Having introduced the main motivation behind the Individual Project as well as the most important objectives, I present the contributions made throughout this dissertation and how they relate to the thesis structure:

- `Background Reading`
  Firstly, I explain the basic details of the Foreign Exchange market, introducing the concept of trading that will be used throughout the project. Additionally I introduce machine learning concepts such as genetic programming and swarm intelligence. Furthermore I give some details about the main statistical tests used for evaluating the results of our approach. In the end, I explain hardware methods for accelerating computation, with our main focus on using Field Programmable Gate Arrays. The chapter concludes with an overview of the most related work in the domain, discussing how it is associated with the dissertation.

- `Trading Algorithm in Software`
  I introduce the first approach we had to identify possible trading patterns which is related to a simple genetic algorithm that makes use of already existent technical indicators. Afterwards I introduce the main approach which uses a genetic program that is not limited to already existent technical indicators, rather it searches for different trading rules that seem to follow some patterns in the market data. Furthermore I explain our new addition to the field, represented by the particle swarm optimisation and how it fits with the genetic program overall. In the end I conclude with briefly describing some implementation details of our approach.

- `Trading Algorithm in Hardware`
  I describe the hardware design for the fitness evaluation part of our genetic program and briefly introduce different acceleration measurements related to speedup details. Then I introduce implementation details for an initial solution and subsequently consider various performance optimisations with regards to our FPGA application. Special attention is given to memory and resource utilisation on the FPGA device and modifications presented in order to reduce it. Furthermore, I do a comparison between my CPU and FPGA execution time results and conclude with remarks concerning my experiences throught the design process.

- `Testing for Robustness`
  In this chapter I detail the numerous mathematical tests such as aggregation and rejection-acceptance testing performed in order to check for our algorithm robustness. I add further details about how the main genetic programming approach behaves in noisy data presence and explain our conclusions through results.
- `Further Statistical Results`
  Continuing from the detailed explanation of the robustness tests we present and interpret the results for some well-known statistics tests, such as "T-statistic" and predictability econometric tests such as the "Anatolyev-Gerko" test.
- `Conclusions and Future Work`
  In the last chapter I conclude with a brief summary of my thesis and I revisit the project objectives and discuss what has been achieved. I also present future work and explain how the project could be expanded and which areas are good candidates for further development.

Chapter 2

Background Reading

In this chapter we will briefly introduce the most important information,, necessary to understand the project, with the key financial concepts being explained, assuming little prior knowledge. The topics that will be discussed include:

- `Financial Background:` The basics of the Forex market, covering some basics trading concepts and technical indicators.
- `Machine Learning Techniques:` Offers an introduction in the Machine Learning field, covering the basics of genetic programming and swarm intelligence.
- `Numerical Tests:` Presents the basic statistical tests that we used for evaluating our program's results.
- `Hardware Acceleration:` A brief discussion of hardware acceleration and basic notions about FPGAs.
- `State-of-the-Art:` A summary of the most relevant related work in domain.

2.1 Financial Background

2.1.1 Foreign Exchange Data

Firstly, we will consider the Foreign Exchange Market is the market in which *currencies* are traded.

Currency Trading is the worlds largest market consisting of almost trillion in daily volume being in a continue growing. FX is not only the largest market in the world, but it is also the most liquid, differentiating it from the other markets. Traders include large banks, central banks, institutional investors, currency speculators, corporations, governments, other financial institutions, and retail investors.

In addition, there is no central marketplace for the exchange of currency, but instead the trading is conducted over-the-counter. Unlike the stock market, this decentralisation of the market allows traders to choose from a number of different dealers to make trades with and allows for comparison of prices. Typically, the larger a dealer is the

better access they have to pricing at the largest banks in the world, and are able to pass that on to their clients. The spot currency market is open twenty-four hours a day, five days a week, with currencies being traded around the world in all of the major financial centres.

All trades that take place in the foreign exchange market involve the buying of one currency and the selling of another currency simultaneously. This happens because the value of one currency is determined by its comparison to another currency. The first currency of a currency pair is called the "base currency", while the second currency is called the "counter currency". The currency pair shows how much of the counter currency is needed to purchase one unit of the base currency and are associated with a single unit that can be bought or sold. When purchasing a currency pair, the *base currency is being bought*, while the *counter currency is being sold*. The opposite is true, when the sale of a currency pair takes place. There are four major currency pairs that are traded most often in the foreign exchange market. These include the EUR/USD, USD/JPY, GBP/USD, and USD/CHF.[14]

Financial instruments existent on the FX Market are:

1. `Spot`: transaction is a two-day delivery transaction (except in the case of trades between the US Dollar, Canadian Dollar, Turkish Lira, EURO and Russian Ruble, which settle the next business day), as opposed to the futures contracts, which are usually three months.
2. `Forward`: a buyer and seller agree on an exchange rate for any date in the future, and the transaction occurs on that date, regardless of what the market rates are then.
3. `Swap`: two parties exchange currencies for a certain length of time and agree to reverse the transaction at a later date.
4. `Future`: standardised forward contracts (avg. length of 3 months and are usually traded on an exchange created for this purpose.
5. `Option`: a derivative where the owner has the right but not the obligation to exchange money denominated in one currency into another currency at a pre-agreed exchange rate on a specified date. *The options market is the deepest, largest and most liquid market for options of any kind in the world.*

Characteristics of the FX market which make it unique are:

- geographical dispersion;
- low margins of relative profit compared with other markets of fixed income;
- huge trading volume representing the largest asset class in the world leading to high liquidity;
- continuous operation: 24 hours a day except weekends;
- variety of factors that affect exchange rates;
- use of leverage to enhance profit and loss margins and with respect to account size.

Many studies were made on the FX market, some of the researchers found a similarity between the Equity market and the FX market, based on the existence of mean reversion and momentum phenomena, present in both of them[29].

2.1.2 Trading

Definition 1. *Trading means performing a transaction that involves the selling and purchasing of a security.[17]*

There are different types of trading, such as *daily trading* or *high frequency trading*:

Day trading the practice of speculation in securities, specifically buying and selling financial instruments within the same trading day, such that all positions are usually closed before the market close for the trading day.

High-frequency trading is the use of sophisticated technological tools and computer algorithms to trade securities on a rapid basis.

2.1.2.1 Technical indicators

Technical trading is a broader style, and is not even necessarily limited to trading; it can indicate a much broader philosophy or approach to investing. In general, it involves looking back in history using the recognisable patterns of past trading data to try to predict what might happen to stocks in the future. However, we all know how poor their forecasts can be. We can only hope that this technical analysis will do more than a little bit better.

The challenge of technical analysis is that there are literally hundreds of technical indicators available and there is no single indicator that can be considered universally best, as each particular indicator, or group of indicators, may be applicable only to specific circumstances. Some technical indicators may be useful for certain industries, others only for stocks of a certain classification (e.g. stocks within a certain range of liquidity or market capitalisation). Because of the unique patterns that highly traded stocks might exhibit throughout history, some indicators may be relevant only to certain individual stocks.

Examples of such technical indicators are: simple moving average, exponential moving average, average directional index, relative strength index, stochastic oscillator, bollinger bands, etc. Further mathematical details about all of the technical indicators implemented can be found in the appendix 7.2.

2.2 Machine learning techniques

Machine learning is about the construction and study of the systems that can learn from data. This field is growing and its applications are becoming more and more important nowadays. One example would be a machine learning system which can be trained on pictures to learn to distinguish between happy and said faces. After learning, it can be used to classify new picture by it's portrait mood.

There are many algorithms related to this field, but the ones we are interested in at the moment are evolutionary algorithms, support vector machines. Machine learning is a branch of the artificial intelligence field and thus we can easily combine different artificial intelligence techniques (e.g: genetic algorithms can be optimised with the use of swarm intelligence properties).

2.2.1 Genetic Algorithms

A *genetic algorithms* is a search heuristic that mimics the process of natural evolution. This heuristic is routinely used to generate useful solutions to optimisation and search problems.[22] Genetic algorithms (GAs) belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimisation problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.[12]

In a genetic algorithm, a population of candidate solutions (individuals) to an optimisation problem is evolved toward better solutions. Each candidate solution has a set of properties (chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.(e.g: Gray coding)

The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, the more fit individuals are stochastically selected from the current population, and each individual's genome is modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

A typical genetic algorithm requires:

- a genetic representation of the solution domain,
- a fitness function to evaluate the solution domain.

Once the genetic representation and the fitness function are defined, a GA proceeds to initialise a population of solutions and then to improve it through repetitive application of the mutation, crossover, inversion and selection operators. The steps of a genetic are simply described below:

Initialisation initially many individual solutions are (usually) randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. The initial population may be traditionally generated randomly, or given.

Selection during each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions measured by a fitness function are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions.

Crossover is a process of taking more than one parent solutions and producing a child solution from them. There are methods for selection of the chromosomes such as: Roulette wheel selection, Tournament selection, Boltzmann selection, Rank selection, etc.

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of algorithm chromosomes to the next in a similar manner as biological mutation. In mutation one or more gene values in a chromosome are altered from its initial state and the solution may change entirely from the previous solution. Hence GA can come to better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability which should be set low, because if set too high the search will turn into a primitive random search.

Termination in this case typically either a solution was find to meet the minimum conditions, or the maximum number of generations has been reached.

2.2.2 Evolving from Genetic Algorithms to Genetic Programming

In artificial intelligence, *genetic programming* (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. Essentially GP is a set of instructions and a fitness function to measure how well a computer has performed a task. It is a specialisation of genetic algorithms (GA) where *each individual is a computer program*. It is a machine learning technique used to optimise a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task [36]. The overall algorithm stays the same, just the structure of the individuals is now different. If in the case of GAs we usually used fixed-size bit streams, GPs actually allow us to use variable size programs which ar usually represented as trees (as we will further see in our implementation).

2.2.3 Swarm intelligence

Swarm intelligence is the collective behaviour of decentralised, self-organised systems, natural or artificial.

Swarm intelligent systems are typically made up of a population of simple agents interacting locally with one another and with their environment. The inspiration often comes from nature, especially biological systems. The agents follow very simple rules, and although there is no centralised control structure dictating how individual agents should behave, local, and to a certain degree random, interactions between such agents lead to the emergence of "intelligent" global behaviour, unknown to the individual agents (e.g: ant colonies, fish schooling).

There are many algorithms related to the swarm intelligent systems behaviour and their characteristics, such as: ant colony optimisation, firefly algorithm, mmulti-swarm optimisation, particle swarm optimisation, etc.

2.2.3.1 Particle swarm optimisation

Particle swarm optimisation (PSO) is a global optimisation algorithm for dealing with problems in which a best solution can be represented as a point or surface in an n-dimensional space. Hypotheses are plotted in this space and seeded with an initial velocity, as well as a communication channel between the particles which then move through the solution space, and are evaluated according to some fitness criterion after each time step. Over time, particles are accelerated towards those particles within their communication grouping which have better fitness values.

The main advantage of such an approach over other global minimisation strategies such as *simulated annealing* is that the large number of members that make up the particle swarm make the technique impressively resilient to the local minima problem.

2.3 Numerical Tests

2.3.1 One sample T-Statistics

A *t-test* is any statistical hypothesis test in which the test statistic follows a Student's t distribution if the null hypothesis is supported. It can be used to determine if two sets of data are significantly different from each other, and is most commonly applied when the test statistic would follow a normal distribution if the value of a scaling term in the test statistic were known. When the scaling term is unknown and is replaced by an estimate based on the data, the test statistic (under certain conditions) follows a Student's t distribution.

In testing the null hypothesis that the population mean is equal to a specified value 0, one uses the statistic:

$$t = \frac{x - 0}{s/\sqrt{n}} \tag{2.1}$$

A *one-sample location test* is used whether the mean of a normally distributed population has a value specified in a null hypothesis.

2.3.2 Anatolyev-Gerko test: Economic significance

Let r_t be the observed log-returns and $\hat{r}_t$ be their forecasts for $t = 1, ..., n$ which depend on the past information $\mathcal{F}_{t-1} = \{r_{t-1}, r_{t-2}, ...\}$.

Let the trading rule of the investor be based on the forecast variable $\hat{r}_t$, in particular, the investor takes a long position if $\hat{r}_t \geq 0$ and a short position if $\hat{r}_t < 0$.

Thus, the one-period return from using the trading strategy is $R_t = \text{sign}(\hat{r}_t) \cdot r_t$.

The null hypothesis is conditional mean independence so that

$$H_0 \colon E(r_t | \mathcal{F}_{t-1}) = \text{const}$$

or that $\hat{r}_t$ and r_t are independent.

The expected one-period return $E(R_t)$ can be consistently estimated under the null by two estimators:

$$A_n = \frac{1}{n} t \sum R_t \tag{2.2}$$

and

$$B_n = \left(\frac{1}{n} t \sum \text{sign}(\hat{r}_t) \right) \left(\frac{1}{n} t \sum r_t \right). \tag{2.3}$$

A_n estimates the average return from using the trading strategy whereas B_n estimates the average return from using the benchmark strategy that issues buy/sell signals randomly with probabilities corresponding to the proportion of buys and sells implied ex post by the trading strategy.

When r_t is predictable investing in the trading strategy will generate higher returns than the benchmark and the difference between A_n and B_n will be sizeable.

The excess profitability statistic is then given by

$$EP = \frac{A_n - B_n}{\sqrt{\hat{V}}} \sim N(0, 1)$$

where

$$\hat{V} = \frac{4}{n^2} \hat{p}_{\hat{r}} (1 - \hat{p}_{\hat{r}}) t \sum (r_t - \bar{r})^2$$

with $\hat{p}_{\hat{r}} = \frac{1}{2} \left(1 + \frac{1}{n} t \sum \text{sign}(\hat{r}_t)\right)$.

2.4 Hardware acceleration

In computing, hardware acceleration is the use of computer hardware to perform some function faster than is possible in software running on the general-purpose CPU. Normally, processors are sequential, and instructions are executed one by one. Various techniques are used to improve performance; hardware acceleration is one of them.

The main difference between hardware and software is *concurrency*, allowing hardware to be much faster than software. Hardware accelerators are designed for computationally intensive software code.

2.4.1 Field-Programmable-Gate-Arrays

In the field of computing, we are accustomed to dealing with problems in two different ways, software and hardware.

The software approach is relatively simple and flexible, with the programmer being able to make modifications to the code quickly. However, this method does not provide the best possible performance.

On the other end of the spectrum we have hardware, i.e. integrated circuits or ASICs. These on the other end are considerable increase in performance, but at the cost of greater development expense and time.

Once a chip has been fabricated you cannot issue a bug fix like with software, the hardware has been preconfigured and will do what it was designed for.[19]

One type of a hardware acceleration is by making use of the so called *Field-Programmable-Gate-Arrays* (FPGAs):

FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" somewhat like many (changeable) logic gates that can be inter-wired in (many) different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.[1]

The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) (circuit diagrams were previously used to specify the configuration, as they were for ASICs, but this is increasingly rare) and its main advantage is that they usually exhibit better power efficiency and performance than software implementations, but this comes at a price of increased complexity for developing the designs.

As running algorithms on the FPGA eliminates the overhead of an operating system, these configurations can offer vast performance increases[31]. Furthermore, it has been speculated that financial institutions have achieved notable speedup utilising these tech- nologies [3]. FPGAs have also been the topic of research in the field of computational finance [25], together with other many machine learning techniques.

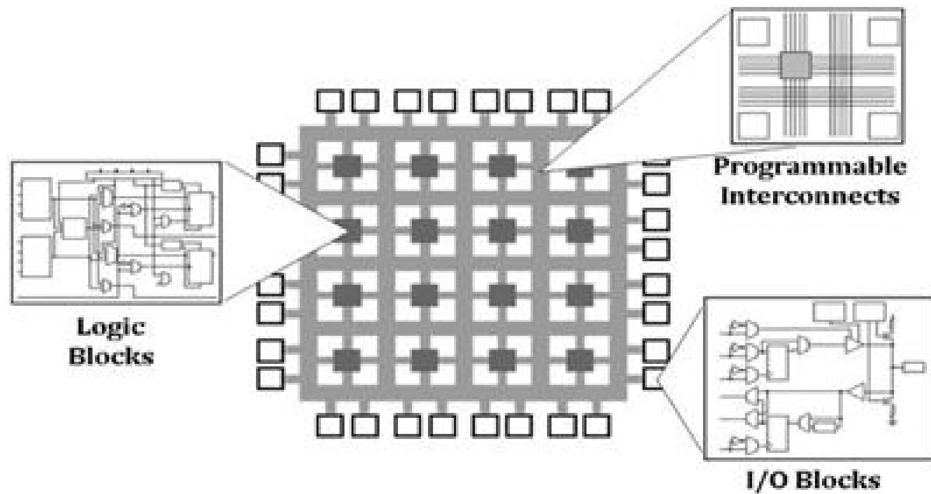

Figure 2.1: FPGA Basic Architecture Organization

2.4.2 Maxeler toolchain

The tool-chain and platform 2.2 that we use for implementing the FPGA application is the one developed and maintained by Maxeler Technologies. In consists of the MAX3 cards that contain a Xilinx Virtex-6 chip and up to 48GB of DD3 DRAM. These cards can be programmed through MaxCompiler which provides a Java-compatible object-oriented API allowing the developer to specify the data-flow graph, contained in what is called a *kernel*. MaxCompiler will then insert buffers which will introduce the appropriate delays in the design that will ensure the correct values will reach the appropriate stages in the pipeline at the correct clock cycle (this process is called scheduling). After the MaxCompiler tool has finished generating, the VHDL code control is given to the Xilinx ISE tools which will afterwards synthesise, place and route the design and generate a bit string that can be used to configure the FPGA [32].

The generated bit-stream is included in the *maxfile* which contains meta-data information about the design (named memory, register names, runtime parameters, etc.). The maxfile can then be linked against a normal C/C++ application using standard tools such as: gcc, ld, etc. The interaction with the FPGA is performed by MaxCompilerRT(low-level runtime) and MaxelerOS (driver layer).

The kernels in MaxCompiler have input streams that are pushed through a pipelined data-flow graph and some of them are output from the kernel. A hardware stream is seen from a programmatic point of view, analogous to a variable (which value potentially changes each cycle) in conventional programming languages.

The MaxCompiler code is written in a Java-like language called MaxJ which provides overloaded operators for hardware streams (e.g: $+$, $-$, $*$, etc.). There are also kernel designs which form part of a MaxCompiler design. The developer specifies a manager that describes the streaming connections between the kernels and it can also be used

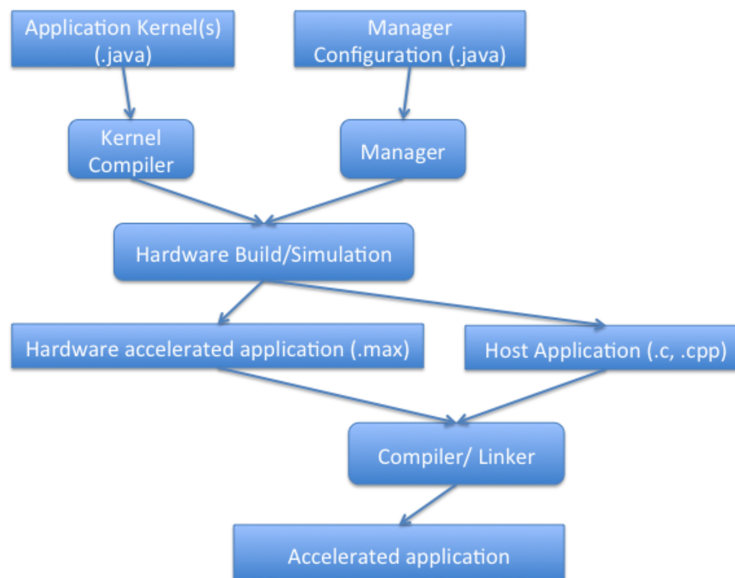

Figure 2.2: Maxeler toolchain diagram

to configure a design to stream data to and from the host through PCIe or from the DRAM that is attached to the FPGA. The developer can instantiate multiple kernels and connect them up.

2.4.3 Hardware Platform

The host machine can communicate with the card through the PCIe bus using the MaxCompilerRT API.

The FPGA provides a number of resources that can be used to specify a design. The MAX3 card that we use contains four specific elements:

LUTs : LookUp tables which are small combinatorial logic elements that can be configured to implement any logical function.

Flip Flops : elements which can be used as registers to implement pipeline stages, etc.

BRAMs : Block RAMs which are memory cells present on the chip itself and possible to be accessed with very low latency.

DSPs : custom tuned elements used for fast multiplication.

The card is connected to the host machine via a PCIe bus which is a popular standard component present on most modern motherboards 2.3.

Chip resources are a very important feature of the FPGAs that we need to take in consideration when building the FPGA design. We thus need to be careful not to use more resources than the chip offers, because then our design will fail the hardware build. We usually aim to use less than 90% of the resources.

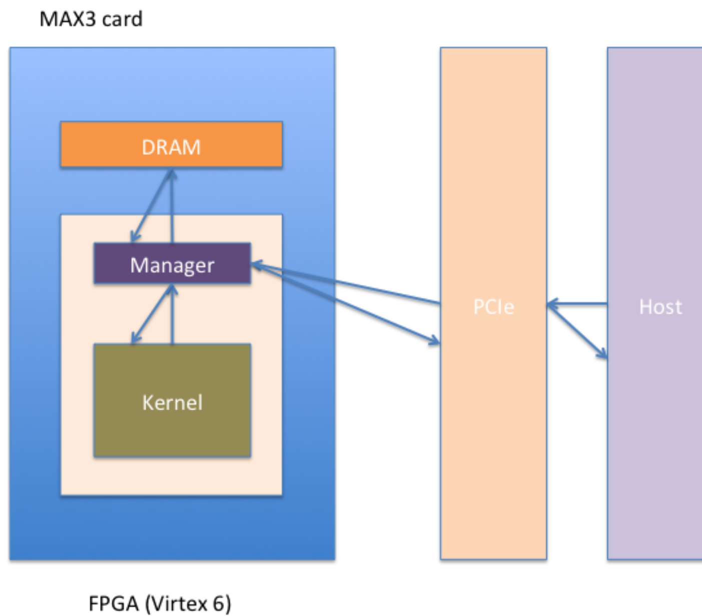

Figure 2.3: Diagram of the hardware parts of the MAXCard showing the relationship between the PCIe, DRAM, Host and FPGA

2.4.4 Floating Point vs Fixed Point Precision

The most common representation for real number in modern architectures is the IEEE-754 floating point representation. With this representation we store the real number using three fields: the sign bit, the mantissa and the exponent. We have 32-bit floating point numbers which have a 8 bits exponent and 24 mantissa (as shown in figure 2.4), as well as 64-bit floating point numbers with a 11 bits exponent and 53 mantissa.

Figure 2.4: 32-bit Floating Point Representation conforming to the IEEE-754 standards

In general, a floating point number with N bits for the exponent can represent a range from $\pm 1 * 2^{-2^N/-2}$ to $\pm 2 * 2^{2^N/-1}$. The decoding of a floating point number involves multiplying the mantissa by 2 raised to the power of the exponent, which is typically an expensive operation.

As we are working with custom hardware, we can use an alternative representation to floating point numbers. We can store a real number as an integer and a fractional part at fixed offsets. This approach makes the arithmetic much simpler and faster, but

it does sacrifice range and accuracy. Fixed point representation is used in applications when the inputs are of limited precision. Faster in hardware terms, means fewer cycles necessary to perform an operation, which further translates to pipeline stages.

2.5 State-of-the-Art

A lot of research has been done in this field, since finding optimal trading strategies is a well hunted area. Thus, it was a challenge to find an area that can add further value, either with an optimal strategy or with a relevant conclusion for the user (i.e: that some strategies do not work under the tested circumstances).

Firstly, we investigated the nature of the markets we are interested in: foreign exchange and futures, and tried to understand what causes traders and computer trading to make profit/losses under certain economic circumstances[11]. We also looked into the distinction between high-frequency trading rules and daily trading rules in those markets.[7]

Secondly, we looked into some existent trading strategies[10] and understand the nature of the technicalities we encounter[40]. We read about trading patterns such as *Logistic Regression*, *Momentum*[18], *Mean-value analysis* etc.

Since this patterns are relatively easy to implement, every trader/strategist can do this, thus nowadays we can't have a competitive advantage by relying solely on this existent "mathematical tricks" of finding similarities inside given trading data. Therefore, we decided to move further down the line and search for machine learning techniques applied in our field.

We found out that a couple of tries have been made in this area, either by using simple classifiers (e.g k-Nearest-Neighbour) to identify any unknown possible patterns in the historical data, or by using learning techniques such as: neural networks or decision trees. A number of papers related to the scope of this project: to build a trading strategy making use of a genetic algorithm, were published to date and we will describe some of the most relevant ones:

An interesting paper related to our work titled *"Genetic Algorithm: An application to Technical Trading System Design"*[33] studies the problem of how can GA be used to improve the performance of a particular trading rule by optimising its parameters, and how changes in the design of GA itself can affect the solution quality obtained in context of technical trading system. The results of the experiments based on real time-series data demonstrate that the optimised rule obtained using the GA can increase the profit generated significantly as compare to traditional moving average lengths trading rules taken from financial literature.

A further contribution to the field of market prediction is made by the paper *"Prediction of Stock Market Indices using Hybrid Genetic Algorithm/Particle Swarm Optimisation with Perturbation Term"*[2]. The paper proposed a new hybrid genetic algorithm/particle swarm optimisation model with perturbation term inspired by the

passive congregation biological mechanism to overcome the problem of local search restriction in standard models. This perturbation term is based on the cooperation between different particles in determining new positions rather than depending on particles selfish thinking which enables all particles to perform the global search in the whole search space for finding new regions with better performance. Experiment study carried out on the most famous stock market indices in both long term and short term prediction shows significantly the influence of the perturbation term in improving the performance accuracy compared to standard models. Even though this paper follows a similar approach to ours, it is not the same exact thing we do in our thesis. This paper and other similar ones aim to predict the actual stock market indices, while our aim is to find optimal trading rules which follow systematic patterns in market data.

The study *"A Genetic Programming Based Stock Price Predictor together with Mean-Variance Based Sell/Buy Actions"*[27] is talking about the use of genetic programming to help develop a virtual stock market environment. This had successful results, thus this paper certainly proves one more time that the genetic algorithms can add value to the finance stock markets not only for prediction but for simulation as well.

As previously remembered, work has been done in term of predicting the financial markets, by using the k-Nearest Neighbours classifier. The paper titled *"Classification of Stock Index Movement using k-Nearest Neighbours (k-NN) algorithm"* [23] talks about how the stock index movement of the popular Indian Stock Market indices BSE-SENSEX and NSE-NIFTLY are investigated with the use of this classifier, which forecasted the daily movement of the indices. Its performance was compared to that of the logistic regression model and the results were better.

The work is followed up by *"Genetic Algorithm for Trading Signal Generation"* [34] which talks about a supervised learning approach to generate trading signals (buy/sell) using a combination of technical indicators. Proposed system uses a genetic algorithm along with the principle component analysis to identify a subset of technical indicators which detect rise and fall of the market with greater accuracy and generates realistic trading signals. The performance of this algorithm was tested using trading data obtained from National Stock Exchange, India. Simulation shows the enhanced profitability for the proposed trading strategy.

One important work, on which we based our decision is called *"High Frequency Foreign Exchange Trading Strategies Based on Genetic Algorithms"*[15]. In this paper the authors used genetic algorithms to generate the most profitable trading strategy based on technical indicators on the foreign exchange market. The trading strategies with neutral position generated by genetic algorithms have an annualised return of 3.7% during test period which is better than the trading strategies with neutral position. Thus, we think that we can optimise this since a lot of potential comes into place, by building a hybrid evolutionary algorithm which has characteristics of swarm intelligence.

We also think that Support Vector Machines might come into place at some point, if time permits us, and we think this will add further improvements on the predictions,

since studies made in this area proved that profits can be made. For example, a paper called *"A Hybrid Machine Learning System for Stock Market Forecasting"*[5] proposed a GA-SVM algorithm for stock market prediction. The results show that the hybrid GA-SVM system outperforms the standalone SVM system. One other paper related to the use of support vector regression for financial technical analysis, is *"An Incremental Learning Approach for Stock Price Prediction using Support Vector Regression"*[26]. In this study, online support vector regression is employed for predicting the stock price, as it is required to incrementally learn the daily updates effectively without relearning the historic data again. The performance of the trained model has been evaluated and found that the online support vector regression model produced 96% accuracy with threshold 1.

One other piece of work that we found interesting is *"Optimising Intraday Trading Models with Genetic Algorithms"*[6]. This article deals with the use of GAs in the parametric optimisation of financial time series trading models over the models' parameter space. It also lays the groundwork for the use of evolutionary optimisation by performing search over the model space itself and the results obtained show that GAs are definitely improving the prediction algorithms one created with other techniques.

We also found out about how great genetic programming can be used even in real-time adaptive trading systems. In their work, the authors of *"A real-time adaptive trading system using genetic programming"*[9] used a GA to provide new trading rules maybe undiscovered before. Despite the individual indicators being generally loss-masking over the data period they tested, the best rule found by the developed system is discovered to be modestly, but significantly, profitable in the presence of realistic transaction costs.

One paper called *"Exploring Algorithmic Trading in Reconfigurable Hardware"*[39], along with a conference publication of the same title[30] provide perspective into the field of accelerating electronic trading using the FPGA architecture. The approach shows very promising results, with the hardware implementation being approximately 377 times faster than the software version and a reduction in latency of 6.25 times.

2.6 Problem identification and proposed solution

The recent studies have shown that one of the most researched aspects in terms of trading strategies is related to identifying systematic trading rules signals inside of the market data and also in predicting the stock market prices. Many pattern recognition techniques have been employed and many approaches have been applied in trying to find a reliable and profitable solution to this problem.

After doing our researched we decided to implement a genetic program with particle swarm optimisation on top of it, which will search for any mathematical potential trading rule that can lead us towards a trading signal pattern identification. As far as we are concerned, studies have been done but only for stock market prediciton, our approach is to actually identify any particular movements or trends inside the market

data. We consider that the trends are very important as if we can predict well the *buy*, *sell* signal, then we can further make predictions about what the actual stock price will be.

Evolutionary algorithms tend to be very time-consuming until they can actually offer us good results back, thus the need of more computation power and execution time speedup. After noticing that the fitness evaluation part of the genetic program is the most expensive part, we decided to make use of hardware acceleration techniques for obtaining a runtime speedup of our application. So, in our best case scenario we would have significantly reduced the execution time for the fitness evaluation and we are remained only with the genetic operators and other specific genetic programming aspects to be computed. Recent studies have shown that genetic algorithms implementation on FGPAs obtain an execution time speedup of $60x$ [21].

Therefore, our approach will not only try to find better results in terms of the trading rules identified as profitable by our program, but we will also be able to obtain results faster than in a CPU normal approach.

2.7 Summary

In this chapter we have presented the most important background information necessary to understanding the work carried out throughout the project.

We began by giving an overview of the foreign exchange and futures markets and the key financial concepts together with some trading basics. Furthermore, we presented some machine learning techniques which provide deeper inside in the techniques we will use in building the trading strategy.

Additionally, we provided an overview of hardware acceleration methods, concentrating mostly on Field Programmable Gate Arrays.

We concluded with an exploration of the state-of-the-art, i.e. the most relevant related work. It was interesting and exciting to notice how much work has been done in the field of financial markets prediction.

Chapter 3

Trading Algorithm in Software

After doing our research in what machine learning techniques proved successful in the prediction of financial markets, we decided to build an evolutionary algorithm which given a number of market price values, numerical constants and operators (mathematical, binary, logical) as input, under certain economical given conditions will be able to choose the best trading rule consisting of a combination of those inputs.

One trading rule will be considered as being best either at minimum loss or after giving the best profit, considering our minimum gain stopping criterion.

We noticed that the use of evolutionary algorithms offered a good rate of success in terms of profitability, over the normal trading technical indicators. There were some issues in not being very reliable under unpredictable change in market conditions [20], and we aim to find with our genetic program not only good trading rules but the states of the market when our genetic program stops performing as expected.

In this chapter we will briefly introduce the most important information related to the design and implementation of our program. The topics that will be discussed include:

- `Genetic Algorithm approach:` Shows the approach we took before introducing genetic programming into place;
- `Genetic Programming:` Explains our solution for the trading rules pattern identification;
- `Particle Swarm Optimisation:` Shows our optimisation added on top of the genetic program which consists in the novel way of identifying any trading rules patterns;
- `Implementation Details:` Briefly describes the implementation details of our solution.

3.1 Genetic Algorithm approach

In the beginning we started by designing and implementing a genetic algorithm which given a set of already existing technical indicators (e.g: momentum, linear regression, stochastic oscillator, etc) will choose the best combination of those technical indicators regarding some given profitability rules.

A typical genetic algorithm follows the following pseudo-code implementation:

input : n, X, m
output: a optimal population

`Initialise generation 0`;
$k \leftarrow 0$;
$P_k \leftarrow$ a population of n randomly generated individuals;
`Evaluate` P_k;
Compute `Fitness(`i`)` for each $i \in P_k$;
repeat
  `Create generate` $k+1$;
  `1. Copy`;
  Select $(1-X)*n$ members of P_k and insert into P_{k+1};
  `2. Crossover`;
  Select $X*n$ members of P_k; pair them up; produce offspring; insert the offspring into P_{k+1};
  `3. Mutate`;
  Select $m*n$ members of P_{k+1}; invert a randomly selected bit in each;
  `4. Evaluate` P_{k+1};
  Compute `Fitness(`i`)` for each $i \in P_k$;
  `Increment`;
  k=k+1;
until *fitness of fittest individual in* P_k *is high enough*;
return *the fittest individual from* P_k

Algorithm 1: Genetic Algorithm

The technical indicators we implemented and were meant to be used for testing are: simple moving (SMA) and exponential moving average (EMA) , bollinger bands , stochastic oscillator, momentum, average directional index (ADX), relative strength index (RSI), moving average divergence convergence (MACD), stochastic RSI, accumulation-distribution index (ADI), Chaikin oscillator.

We used the genetic algorithm to select trading rules which looked like this:

RULE$\|CONDITION\|CONNECTOR\|$...

RULE$\|CONDITION\|CONNECTOR\|...\|ACTION$.

Since it was designed as a genetic algorithm and not as a genetic program, the trading rules structure is fixed and we identify individual rules with (True = 1, or False = 0),

connectors as Boolean operators (in our case AND (11), OR(01) and XOR(10)) and actions such as Buy (1) or Sell (0). Thus a trading rules looked like: 0010011101 which told us to "Sell if AMA False OR RSI True", where a '1' in the appropriate position (fixed for each indicator) shows that the indicator is included in the rule, whereas a '0' shows that the indicator is to be ignored, and each indicator is represented by a number translated to binary [9].

For the sake of simplicity we would have used a string of maximum size for allowing us to include the action, all the technical indicators, the connectors and the fixed bit for whether they are active or not.

In our case we would have performed all the GA operations on bits and we chose a *Gray coding* approach [6]. This is applied to the binary chromosome to help improve the GA search process.

Definition 2. *Gray encoding is a way of transforming binary numbers so that a unit numerical change corresponds to a change of a single bit in the binary representation.*

For example, for the representation of the decimal numbers, 7 and 8 the binary representations differ in all four bits while the Gray coding representations differ in one (7 is in binary 0111 while in Gray 0100, and 8 is in binary 1000 while in Gray 1100). This maintenance of topological closeness in genetic encoding has important implications concerning such issues as the preservation of genetic schema and the perturbation effect of the genetic mutation operator.

We did not proceed with wrapping up this implementation and testing its profitability because we decided we should go further than being bound by previous human developed technical rules and prove if new technical rules developed by our genetic program (which we will present in more detail in the following section) outperform the already found technical indicators (we will attach in appendix all the technical indicators that we've implemented for our solution).

3.2 Genetic Programming

3.2.1 Design

The genetic programming method provides an effective method for searching over space of potential trading rules, both linear and non-linear. This method allows us to evaluate predictability as generally as possible and not impose any effective restriction on the form of the model, predictor or trading rule.

In artificial intelligence, genetic programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task, using a set of instructions and a fitness function. It is a specialisation of genetic algorithms (GA) where each individual is a computer program.

This machine learning technique is used to optimise a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task providing a systematic search process directed by performance rather than gradient.

Starting from an initial set of rules (randomly generated trading rules), the genetic program evaluates on the training data the fitness of various candidate solutions using the given objective function and provides as an output, trading rules that have better in-sample cumulative returns. We then test those rules on a number of out-sample data and decide on the best performing trading rules given the testing circumstances.

In our approach we build a trading rule as a binary logical tree, which produces true (1) or false(0) signals given the set of input variables. If the value of the rule is true, it gives the signal to buy long an asset and if the rule is false the the trader should sell the asset short. The rules are represented in the form of randomly created binary trees with terminals and operations (mathematical and binary) in their nodes.

A genetic program is made of different individual elements such as: functions, terminals, fitness-based selection, genetic operators, variable length programs and population initialisation. There are two different ways of creating a GP program, a *generational* approach where a new generation replaces the old generation and the program cycle continues, or a *steady-state* approach where we have no generations present. In our implementation we will make use of the former approach.

The preliminary steps in a GP run which we will define in more detail below are:

1. Define the terminal set.
2. Define the function set.
3. Define the fitness function
4. Define parameters such as population size, maximum individual size, crossover probability, termination criterion(e.g: maximum number of iterations or trading rule performance), etc.

3.2.1.1 Terminals and Functions

Definition 3. *The* `terminal set` *is comprised of the inputs to the GP program, the constants supplied to the GP program, and the zero-argument functions with side-effects executed by the GP program [13].*

Input, constants and other zero-argument nodes are called terminals or leafs because they terminate a branch in a tree-structured genome. Terminals are inputs to the program, constants or function without argument. In either case, a terminal returns an actual numeric value without, itself, having to take an input. Basically, the terminal set is comprised, in part, of inputs.

In our case, the terminal set contains the variables, which take their values from data and are updated every time new information arrives in the market. Thus, it allows the conditioning information sets to update the trading rule as time passes. The algorithm also explicitly computes log values of the conditioning variables, their moving average values, and maxima and minima over different periods. The terminal set also includes real numbers and specific market price values (e.g: volume of the bid, depth of the ask, etc.) as terminal constants.

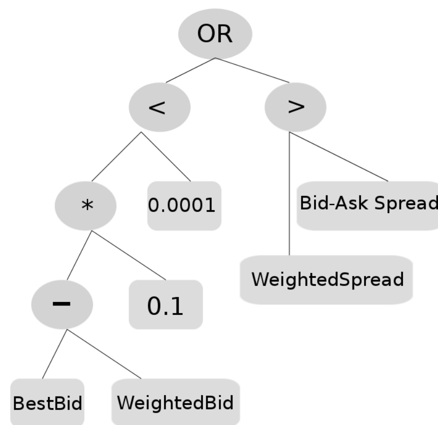

Figure 3.1: Trading Rule

Each of these inputs becomes part of the GP training and test sets as a GP terminal. Genetic programming differs from other machine learning techniques also in how it represents the inputs. Each input in the training set becomes part of the terminal set in a GP system. Thus, the inputs of the learning domain are just one of the primitives GP uses to build program structures, not being represented in any fixed way or in any particular place, thus they can be ignored altogether.

In a typical tree-based GP, a set of real-numbered constants is chosen for the entire population at the beginning of the run and these constants do not change their value during the run, what we do in our design is we choose a set of constants and then when performing the genetic programming operations on each individual from the population, we randomly select from those constants. This is valid for the other variable and real numbers inputs.

Definition 4. *The* `function set` *is composed of the statements, operators, and functions available to the GP system [13].*

The function set may be application-specific and be selected to fit the problem domain. The range of our available functions is broad. This is, after all, genetic programming.

Our function set used to define the technical rules consists of the binary algebraic operations: { +,−,div,mul }; max,min, binary order relations { $<$, $>$, $<=$, $>=$, $=$ }, logical operations { and, or }, and unary functions such as absolute value and change of sign.

One important property of the function set is that each function should be able to handle gracefully all values it might receive as input. This is called the *closure property*. An example of a function that does not fulfil the closure property is the division operator. This cannot accept zero as input. Division by zero will normally crash the system, thereby terminating the GP run. As this is unacceptable, we instead of the standard division operator, defined a new function called protected division which is like normal division except for zero denominator inputs. In that case the function returns purely zero.

The functions and terminals used for a GP run should be powerful enough to be able to represent a solution to the problem. For example a function set consisting only of the addition operator, the binary less sign and the logical and will probably not give interesting and useful results in our case.

3.2.1.2 Initialisation

The first step in actually performing a GP run is to initialise the population. This means creating a variety of program structures for later evolution.

One of the principal parameters of a GP run is the maximum size permitted for a program. For trees in GP, that parameter is expressed at the maximum depth of a tree or the maximum total number of nodes in the tree. In our case, the algorithm maximum depth size is set to 16.

Once we have decided on the terminals and functions set allowable, the initialisation of a tree structure is fairly straightforward. There are two different methods for initialising tree structures in common use and these are called *full*, *grow*.

Figure 3.2 shows a tree that has been initialised using the *grow* method with a maximum depth of five. This method produces irregular shape trees because nodes are selected randomly from the terminal and the function set thought the entire tree (except the root node which uses only a logical or binary operator; in case the root node is a logical operator then its immediate children are binary operators and the rest of the children are part of the functions and terminals set). Once a branch contains a terminal node, that branch has ended, even if the maximum depth has not been reached.

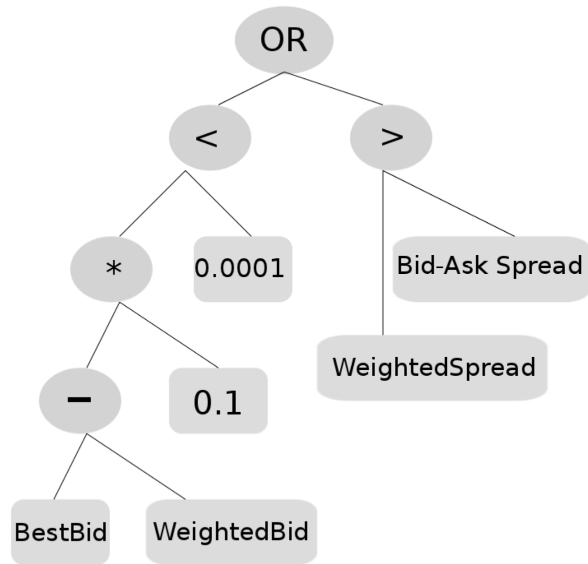

Figure 3.2: Trading rule grow initialisation

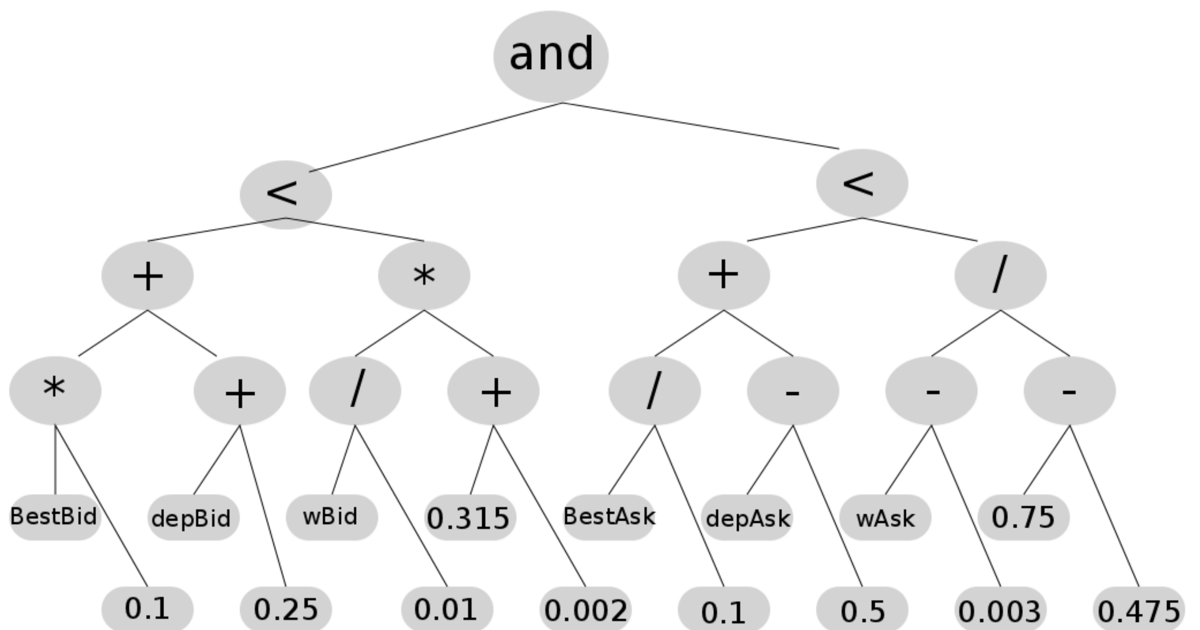

Figure 3.3: Trading rule full initialisation

Instead of selecting nodes randomly from the functions, binary, logical and terminal

set, the *full* method (see figure 3.3) chooses only functions until a node is at the maximum depth. Then it chooses only terminals. The result is that every branch of the tree goes to the full maximum depth (again five in our case).

For both methods, if the number of nodes is used as a size measure, growth stops when the tree has reached the used size parameter.

However, if we use any of the presented above initialisation methods alone, we will not make the most of our design, and we can result in a uniform set of structures in the initial population because the routine is the same for all individuals. Thus according to more recent studies we will use the *ramped half-and-half method*. This method is intended to enhance population diversity of structure from the outset.

The way our algorithm uses this method is: suppose the maximum depth parameter is 8. The population is divided approximately equally among individuals to be initialised with trees having depths 2, 3, 4, 5, 6, 7 and 8. For each depth group, half of the trees are initialised with the full technique and half with the grow technique.

3.2.1.3 Genetic operators

An initialised population has usually low fitness. The evolution proceeds by transforming the initial population by the use of genetic operators which in machine learning terms are called the search operators.

The principal genetic operators that we make use as well are:

1. Crossover

2. Mutation

3. Reproduction.

We use the above GP operators as we will briefly describe below:

The `crossover` operator is used to combine the genetic material of two parents by swapping a part of one parent with a part of the other. The main steps are:

- Choose two individuals as parents, based on mating selection policy;

- Select a random sub-tree in each parent;

- Swap the selected sub-trees between the two parents and the resulting individuals will be their children.

In our implementation one of the offspring resulted by crossover replaces the less fit parent in the population.

The `mutation` operator is used to operate on only one individual and this operation is applied given a probability of mutations which is a parameter of the GP run. When one of the individuals in a particular population is selected for mutation, our algorithm selects a point in the tree randomly and replaces the existing sub-tree at that point with a new randomly generated sub-tree. This new randomly generated sub-tree is created in the same way and subject to the same limitations as programs in the initial random population. The mutated individual is then placed back in the original population.

The `reproduction` operator is the simplest one to use because this operator just selects an individual, it creates a clone of the selected individual and it's copy is placed into the population, together with the original individual. Thus, after this step we will have two versions of the same individual in the population. Reproduction is made using another GP parameter, namely the reproduction probability, and we set this to a lower value than the mutation one.

In our design we don't perform mutation on the first 25% of the population and after that we perform the mutation with a 0.2 rate. We use a crossover probability of 0.2 and that is applied over all individuals in the population. The reproduction probability is set to 0.05 in our case.

3.2.1.4 Fitness Evaluation

As GP must choose which members of the population will be subject to genetic operators such as crossover or mutation, one of the most important parts of its model of evolutionary learning is fitness-based selection which affects both the ordering of the individuals in the population, and the contents of the population as well.

GP's evaluation metric is called a *fitness function* and the manner in which the fitness function affects the selection of individuals for genetic operators may be referred to as the GP selection algorithm.

Definition 5. *`Fitness` is the measure used by GP during simulated evolution of how well a program has learned to predict the output(s) from the input(s) - that is, the features of the learning domain.*

The goal of having a fitness evaluation is to give feedback to the learning algorithm regarding which individuals should have a higher probability of being allowed to multiply and reproduce and which individuals should have a higher probability of being removed from the population.

The fitness function is calculated on the training set allocated by us and is designed such that it gives us continuous feedback about how well a program performs on that particular training set.

The way we measure the trading rules fitness is by making use of the cumulative returns from the following simple trading strategy: according to the signal provided by the in-test trading rule, the trader buys long or sells short 1 million pounds sterling.

With this approach we can control the potential price impact of trade because we can assure we have the necessary liquidity to complete the minimum specified trade size transaction. When new information arrives from the market, the trader can re-evaluate its trading signal and adapt to the new market conditions by updating its position accordingly. Therefore, as soon as any change in the limit order book is present the trader has the choice to keep or change its same position depending on the outcome of the signal(i.e: buy, sell).

For controlling the frequency of trading, we add a trading threshold to the strategy and according to this, the trader is allowed to trade only if the exchange rate exceeds at any point $+k$, $-k$, relative to its transaction price.

Formally, let's assume: z_t = - 1 for a short sterling position and z_t = 1 for a long sterling position and p_t price at time t and p_{t1} price at time t_1 where $t1$ is the time of the trader's last transaction. When

$$|p_t - p_{t1}| >= k \tag{3.1}$$

the trader will be able to re-evaluate its position.

The parameter k is used to filter out weak trading signals and determines an "inertia band" that prompts one to trade only once the exchange rate exceeds the value of a certain characteristic by a value of k. In our case the characteristic followed is represented by the past exchange rate values and is quite related to the filter trading strategy:

Definition 6. *A `filter trading strategy` is a trading strategy where technical analysts set rules for when to buy and sell investments, based on percentage changes in price from previous lows and highs. The filter rule is based on a certainty in price momentum, or the belief that rising prices tend to continue to rise and falling prices tend to continue to fall. It is often considered a subjective screener, due to it being set by an analyst's own interpretation of a stock's historical price history [16].*

It was shown [8] that in the face of Knightian uncertainty incomplete preferences may lead to an absence of trading, thus the importance of such a filter threshold which should be part of the traders strategy. For example, if $k = 0$ then the trades can take place every time the mid-quote of the exchange rate changes, exhibiting the largest number of transactions. Following this, as the value of k increases, the trading frequency drops.

The performance measure which represents in our case the fitness function for our genetic program is based on simple cumulative returns and it mainly evaluates the profitability of trading strategies[20]:

$$R_c = \Pi_t(1 + z_t * r_t) - 1 \tag{3.2}$$

where r_t = (p_t - p_{t-1}) / p_{t-1} is the one-period return of the exchange rate and p_t corresponds for best bid or best ask price.

3.2.1.5 Selection

After we determined how well an individual performs on the training data, by applying a fitness function, we have to decide whether to apply genetic operators to that individual and whether to keep it in the population or allow it to be replaced. This task is called *selection* and is responsible for the speed of evolution and is often declared a culprit when it comes to premature convergence of the evolutionary algorithm.

In our algorithm we use the *ranking selection* which is based on the fitness order, into which the individuals can be sorted. The selection probability is assigned to individuals as a function of their rank in the population. We are using *linear ranking* where the probability of selecting an individual is a function of the rank:

$$p_i = \frac{1}{N}[p^- + (p^+ - p^-)\frac{i-1}{N-1}] \tag{3.3}$$

where p^-N is the probability of the worst individual being selected, and p^+N the probability of the best individual being selected, and

$$p^- + p^+ = 2; \tag{3.4}$$

should hold for the population size to remain constant.

Other methods of selection used in GP are: fitness-proportional selection, truncation selection, tournament selection. We decided to make use of the ranking selection as it's less computationally expensive compared to the others and is a good fit for our trading strategy search selection.

Having defined the individual steps of the GP, we can now present how our generational genetic program works:

1. Initialise randomly the population P_0 of trading rules given the terminal, functions, binary and logical sets and initialise the number of populations;
2. Set $step := step + 1$;
3. Evaluate the in-sample fitness of each individual in the population using the fitness function;
4. Select an individual/individuals in the population using the selection (ranking in our case) algorithm;
5. Perform genetic operations (crossover, mutation and reproduction) on the selected individual or individuals;
6. Insert the result of the genetic operations into the new population;
7. If the termination criterion is fulfilled, then continue, otherwise replace the existing population with the new population and repeat steps 3 - 7.
8. Present the best trading rule (individual) in the population as the output from the algorithm

After each such iteration, rules that have poor performance according to the fitness function are removed from the population and only the more profitable candidates survive and carry their structure on-wards to create new trading rules.

In more details, at the end of each iteration we get the new population of trading rules sorted by their fitness. The way we move to the next generation is by accepting the best half of this new population to move on and rejecting the least fit half of the population. When we will move to the next generation we will inject the best half of the population and we will apply the genetic operators on those individuals, until we will get to the initial wanted trading rules population size. Then, arriving to the next iteration we calculate the new individuals fitness and at the end of this current iteration we get back a new population sorted by the individuals fitness, as so on for N iterations.

At the end of the first test, we save the best strategy or a percentage of the best performing strategies. This best fit strategies will be tested at the end of the X test-run, on the out-sample market data to see their overall cumulative profit performance.

Ultimately, the algorithm converges to the trading rule achieving the best in-sample performance given the conditioning information.

Shortly, one part of our genetic program evaluation is: we run in-sample the algorithm for a number of let's say 100 times, with a convergence criterion of 1000 iterations each and present the best out-sample performing 100 resulting trading rules on each we perform a number of tests for testing its robustness, reliability and overall performance. This tests will be presented later in more detail.

3.2.1.6 Tree structure execution and memory

Definition 7. *`Reverse Polish notation` (RPN) is a mathematical notation in which every operator follows all of its operands, in contrast to Polish notation, which puts the operator in the prefix position. It is also known as post-fix notation and is parenthesis-free as long as operator arities are fixed.*

In RPN the operators follow their operands; for instance, if we want to add 10 and 4, one would write "10 4 +" rather than "10 + 4". If there are multiple operations that we want to compute then the operator is given immediately after its second operand and the expression is written "11 4 + 9" in conventional mathematical notation then would be written "11 4 9 +" in RPN: subtract 4 from 11, then add 9 to that. One of the big advantages of RPN is that it obviates the need for parentheses that are required by infix.

The algorithm for evaluating any post-fix expression is quite straightforward[38]:

```
While there are input tokens left
..........Read the next token from input
..........If the token is a value
........................Push it onto the stack
..........Otherwise, if the token is an operator
........................It is known a priori that the operator takes $n$
    arguments
........................If $<$ $n$ values on the stack
.............................(ERROR) the user has not input sufficient values
    in the expression
........................Else, Pop the top $n$ values from the stack
........................Evaluate the operator with the values as arguments
........................Push the returned results, if any, back onto the stack
...........If there is only one value in the stack
........................That value is the result of the calculation.
...........If there are more values in the stack
........................(ERROR) the user input has too many values.
```

One other advantage of using RPN tree structure is that it uses only *local memory* during execution time.

3.3 Particle swarm optimisation

A basic variant of the PSO algorithm works by having a population (called *swarm*) or candidate solutions (called *particles*). These particles are moved around in the search-space according to a few formulae. The movements of the particles are guided by their own best position in the search-space as well as the entire swarm's best known position. When improved positions are being discovered these will then come to guide the movements of the swarm. The process is repeated and by doing so it is hoped, but not guaranteed, that a satisfactory solution will eventually be discovered. Therefore, is very important to find a good convergence criterion such that we can allow the swarm to find an optimal solution.

Formally, let $f : R^n -> R$ be the cost function which must be minimised. The function takes a candidate solution as argument in the form of a vector of real numbers and produces a real number as output which indicates the objective function value of the given candidate solution. The gradient of f is not known. The goal is to find a

solution a for which $f(a) \leq f(b)$ for all b in the search-space, which would mean a is the global minimum. Maximisation can be performed by considering the function $h = -f$ instead.

Let S be the number of particles in the swarm, each having a position $xi \in R^n$ in the search-space and a velocity $vi \in R^n$. Let pi be the best known position of particle i and let g be the best known position of the entire swarm.

We will describe shortly what each of the main PSO operators do as this algorithm contains different similarities with the GA one [37]:

Parameter Selection Basically, it can be imagined that the function which is to be minimised forms a hyper-surface of dimensionality same as that of the parameters to be optimised (search variables). It is then obvious that the 'ruggedness' of this hyper-surface depends on the particular problem. Now, how good the search is depends on how extensive it is, which is decided by the parameters. Whereas a 'lesser rugged' solution hyper-surface would need fewer particles and lesser iterations, a 'more rugged' one would require a more thorough search- using more individuals and iterations.

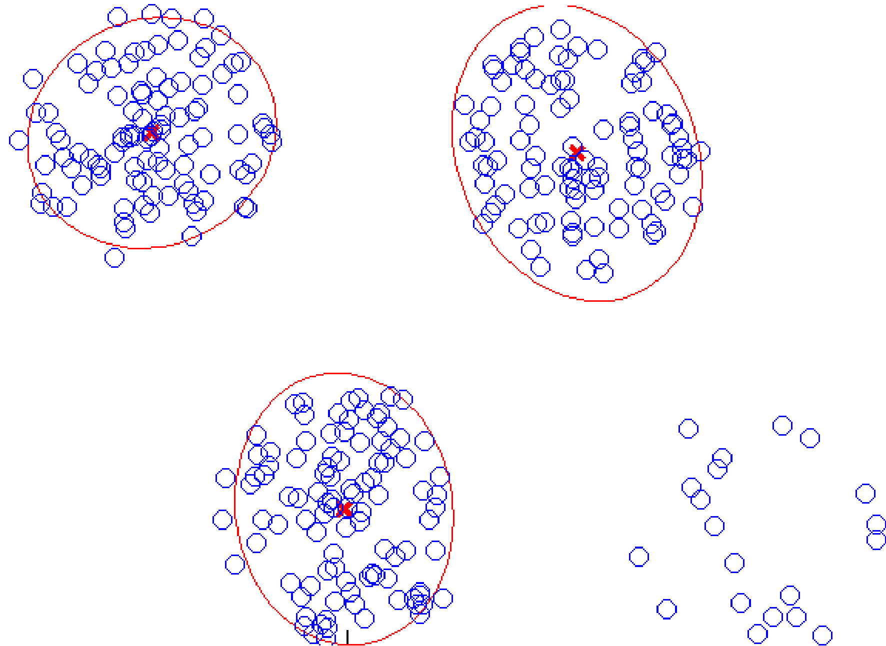

Figure 3.4: PSO approach to split data set into different classifier classes

Neighbourhood and Topologies The basic PSO is easily trapped into a local minimum. This premature convergence can be avoided by not using the entire swarm's best known position g but just the best known position l of a sub-swarm "around" the particle that is moved. Such a sub-swarm can be a set of particles that is not depending on any distance but have the same given characteristics and aim to approach the same goal. In such a case, the PSO variant is said to be local best (vs global best for the basic PSO).

Convergence In relation to PSO the word convergence typically means one of two things, although it is often not clarified which definition is meant and sometimes they are mistakenly thought to be identical: it may refer to the swarm's best known position g approaching (converging to) the optimum of the problem, regardless of how the swarm behaves; or it may refer to a swarm collapse in which all particles have converged to a point in the search-space, which may or may not be the optimum.

A basic pseudo-code version of the PSO algorithm initialisation is described in figure 3.5 and the remaining operators in figure 3.6 [4]:

Algorithm 1 Initialize

```
1: for each particle i in S do
2:   for each dimension d in D do
3:     //intialize all particles' position and velocity
4:     x_{i,d} = Rnd(x_{min}, x_{max})
5:     v_{i,d} = Rnd(-v_{max}/3, v_{max}/3)
6:   end for
7:
8:   //intialize particle's best position
9:   pb_i = x_i
10:  //update the global best position
11:  if f(pb_i) < f(gb) then
12:    gb = pb_i
13:  end if
14: end for
```

Figure 3.5: PSO initialization

Algorithm 2 Particle Swarm Optimization (Global Best)

```
1: //initialize all particles
2: Initialize
3: repeat
4:   for each particle i in S do
5:     //update the particle's best position
6:     if f(x_i) < f(pb_i) then
7:       pb_i = x_i
8:     end if
9:     //update the global best position
10:    if f(pb_i) < f(gb) then
11:      gb = pb_i
12:    end if
13:  end for
14:
15:  //update particle's velocity and position
16:  for each particle i in S do
17:    for each dimension d in D do
18:      v_{i,d} = v_{i,d} + C_1 * Rnd(0,1) * [pb_{i,d} - x_{i,d}] + C_2 * Rnd(0,1) * [gb_d - x_{i,d}]
19:      x_{i,d} = x_{i,d} + v_{i,d}
20:    end for
21:  end for
22:
23:  //advance iteration
24:  it = it + 1
25: until it > MAX_ITERATIONS
```

Figure 3.6: PSO Global Best

3.3.1 Approach

We decided to implement a Hybrid Genetic/PSO algorithm as can be seen in figure 3.7. The figure represents a scanned version of the original optimisation idea we had at some point and we are glad we had time to put it into practice.

The particle swarm optimisation (PSO) was added after we had a stable version of the GP working. So, now, in our hybrid algorithm, we have added one more complexity step, which decreases a bit more the execution speed of the program, but increases its performance.

As we have previously mentioned, we normally perform the genetic operators after the fitness selection step is done. With the new approach, we will actually perform the fitness selection and then we will apply the PSO on the best half of the elites. Afterwards, we will perform the genetic operators on the new enhanced elites only and send the offspring and the enhanced elites into the new population.

Shortly, the PSO search by its nature, will aim to find the global mini-miser or maximiser of a function that we give as input, and it will give back, in our case, groups of trading rules which converge towards our preset optimum target.

The fitness selection of the best fit individuals, after the fitness evaluation is performed, is done using a *fitness proportionate selection* (roulette wheel) approach. In this selection, as in all other types of selections, the fitness function assigns a type to possible solutions or chromosomes[**?**]. This fitness level is used to associate a probability of selection with each individual chromosome. If f_i is the fitness of individual i in the population, its probability of being selected is

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{3.5}$$

where N represents the number of individuals in the population.

After calculating the fitness probability of selection for each individual in the population, we perform the particle swarm optimisation on the best half of population and we receive back the particles. What we mean by particles, is small groups of trading rules which aim to maximise the return, following the 3.2 formula and respect a couple of constrains such as: no two combinations of groups are the same, each group will not contain the same trading rule selected twice.

So, after we get back from the PSO the groups of trading rules which respect the above conditions, we will perform the genetic operators (crossover, mutation, reproduction) within each group. By doing this, we will assure that the children will get parent's characteristics which proved to give good results, so they might have a better chance of success than when we performed crossover between completely random parents (original GP) where there were higher chances of combining two least fit parents and get even a less performing children.

3.4 Implementation Details

3.4.1 C++ implementation

I chose to build my genetic program using C++ because evolutionary algorithms can be very computationally expensive and thus we needed a programming language which can let us easily benefit from compiler level optimizations and specific memory allocation and acces. I have used C++ and not pure C because I wanted to be able to structure my code in classes and objects. One more reason for chosing this language on top of another high-level-rder programming language like Java, for example, is that when running an application for the FPGA using the Maxeler tools, we need to link the hardware design implementation with a C or C++ application.

The program is built as a genetic standalone framework, which allows the future developer to easily add new futures to the current GP structure. We did not use any predefined genetic programming library, instead we have build the whole algorithm from scratch.

The program starts in default mode with a series of default parameters. It is straightforward to change the genetic specific parameters such as GP's operators' probabilities, selection and initialisation type.

Before we start the actual GP, we need to input the training data and/or the testing data. Compulsory for us is to connect our algorithm to the *.csv* file containing the training data. This will allow the GP to find patterns in the data, from which the best in-sample performing ones are saved to new *.csv* files. We can choose to add the testing data from the beginning (as well in *.csv* format) and then our program will automatically recognize our option and after identifying the trading rules, will proceed with testing the best ones and give us back now two *.csv* files: one for the in-sample performance and one for the out-sample performance of the best strategies, per run and for all GP iterations.

From the command line we can choose the type of test we want to run, if we want to run all the tests, or if we just want to find new trading rules on an input data set. We can also choose if we want to run the particle swarm optimisation version or just the simple one.

The majority of the tests performed have been implemented using C++ and integrated with the big application, just more specific statistical tests such as "T-statistics" and "Anatolyev-Gerko" were build using MATLAB. And in those situations we chose to use MATLAB as it contains very well defined fast statistical libraries.

3.4.2 Market Data Parser

I've build my own *.csv* C++ parser which is integrated with the overall program and works for any *.csv* file, but we use it for the limit order book customised market data format.

This C++ *.csv* parser can be used to parse the whole file at a time and store the data into memory and then use it, but this decreases the algorithm speed and can only be used for a limited *.csv* file size (the limitations being related to our computer RAM). Because in both our training and testing we need to use a big number of market prices (order of hundreds of thousand or even millions) and specific values related to them then our *.csv* file size can be of an order of GB in memory size. Thus we can parse one transaction at a time, evaluate its value, store the result for further computations and then parse the next transaction, until we reach the end of file. Or we can use it as we currently do, parse a big chunk of data, store it in memory and evaluate the expression of that, and continuously do this until we evaluate our results on all data needed for training/testing.

3.4.3 Running the statistical and robustness tests

Since we have run a large number of tests, each one of them being represented by a large number of iterations of the genetic program, we needed to be able to split the tests in different jobs over many different machines.

For being able to do this I used *Condor* which is a tool that treats the connect network of machines as a graph in which every machine's available CPU code has been configured as a "slot" on which jobs can be scheduled to run. Thus, when Condor receives a job, it distributes the executable across many machines in order to parallelise not only between machines but also between their CPUs and cores. We submitted the Condor jobs using a command script which describes the executable file to be run and the amount of times (jobs) to perform and specifies also the command line arguments needed for our simulation. We save all output in the usual way to different files with representative names. We thus observed a noticeable decrease in the waiting time of our tests, because of the increased parallelism present when using Condor.

Thus, being able to parallelize 100 tests each with 1000 iterations of the same run of the genetic program was a big benefit when testing for the correctness and robustness of our approach.

3.5 Summary

We started our current chapter by describing the first genetic algorithm approach which was making use of different trading technical indicators (that we have implemented) in order to predict any potential profitable trading rules. As we chose to move further than depending on already discovered mathematical trading rules, we decided to build a genetic program which allows to much more freedom in terms of any potential trading rules as it searches to identify new mathematical formulas for predicting trading signals. We thus described our genetic programming design.

We then described the novel program that we created when adding the particle swarm optimisation on top of the genetic program. As described in the chapter, with the new swarm intelligence addition we will perform the genetic operators only between classes of similar strategies encountered in the PSO search.

We concluded the chapter by briefly detailing the main parts of our program implementation and testing methods.

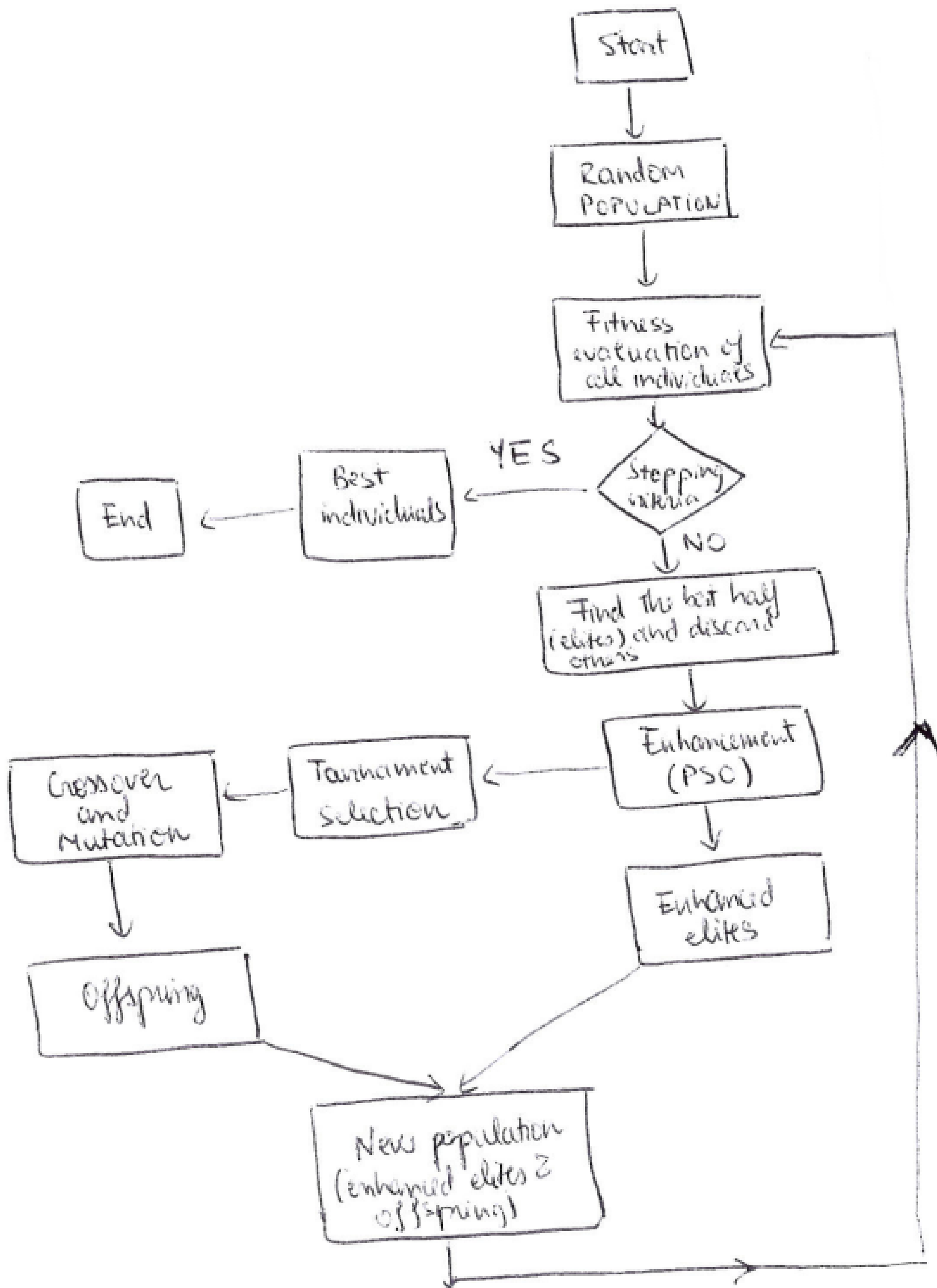

Figure 3.7: Hybrid GA/PSO algorithm original handwritten flow chart

Chapter 4

Trading Algorithm in Hardware

In this chapter we will briefly introduce the most important information, necessary to understand our FPGAs approach and any further improvements that we could add to our application. The topics that will be discussed include:

- `FPGAs Design:` Presents the part from our algorithm that we want to accelerate and our approach to do this.
- `FPGAs Implementation:` Offers a short overview of the main techniques used to implement the solution on the FPGA, together with some MaxJ implementation examples.
- `Performance Evaluation:` shows speedup and compares the runtime results to the ones obtained by the C++ version.
- `Challenges and Improvements:` presents the main challenges and gives an overview of any improvements in general that can be added to our design.

4.1 FPGA Design

4.1.1 Design Specifics

Figure 4.1 presents the part from the Genetic Program CPU program which was accelerated using FPGAs.

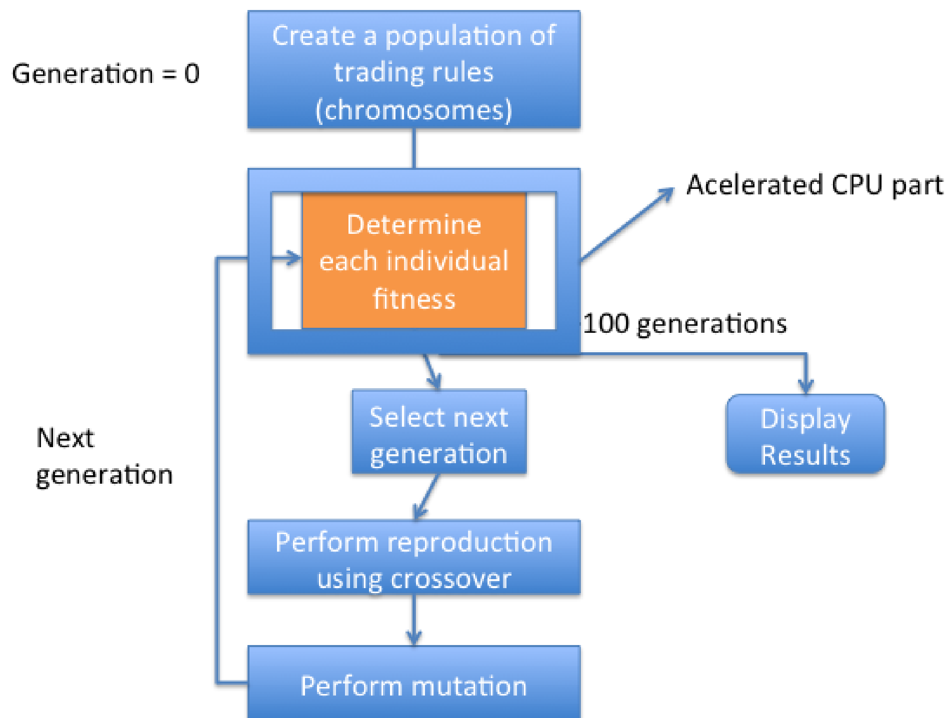

Figure 4.1: Genetic Program with FPGA acceleration highlight

From the genetic programming theory we know that the fitness evaluation part is the most computationally expensive part of the algorithm. Because of this, it takes a significant amount of time, on a CPU, to run the algorithm and get results back. Thus, the need of introducing a hardware acceleration method, where all the algorithmic and compiler level C++ optimisation's (e.g flags such as $-O3$) don't help us too much anymore.

After profiling our C++ application using *Valgrind* (tool for automatically detecting many memory management and threading bugs, and for profiling programs in more detail) we detected that 78% of our CPU application runtime goes into the fitness evaluation part of the algorithm, the remaining 22% being represented by the genetic operators and the *.csv* market data parsing.

In our approach we have used Field Programmable Gate Arrays and managed to obtain a speedup for the execution time of our fitness evaluation.

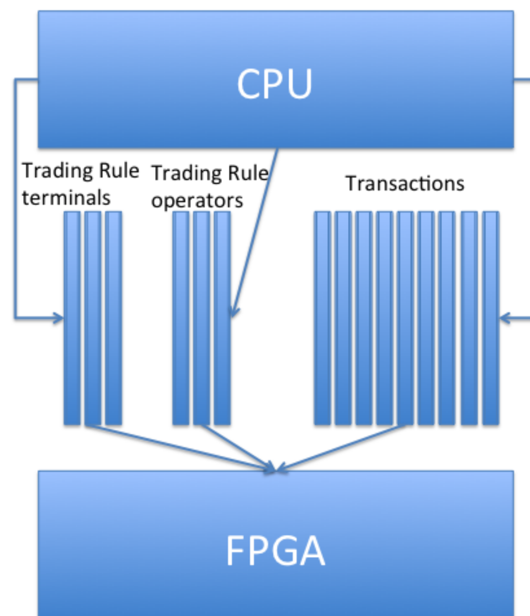

Figure 4.2: CPU - FPGA communication channel

Figure 4.2 shows the way in which our CPU application currently communicates with the FPGA:

Our design involves sending from the CPU the population of individuals (represented as trees) as *two streams*: first stream is represented by each trading rule terminals and the second stream contains each trading rule operators (mathematical, logical, binary functions). We keep the order of the trading rules such that the array block address for the trading value sent in the first stream corresponds to the array block address from the operators value sent in the second stream.

Together with those two streams we currently send a limited number of transactions (as we will further explain why limited) on which we want to compute our individuals' fitness. The transactions (each transaction being represented by an array of market data price values on which we evaluate our trading rules) are all sent together to the FPGA as the *third stream* of data.

After sending the streams from the CPU to the FPGA we now perform the fitness evaluation of the current population on the FPGA: we will evaluate each of the individual trading rules on all of the transactions and get back a result per FPGA cycle. The result corresponding to the last transaction for one of the individuals represents the actual fitness value of the evaluated individual.

Once the computation is done, we will send back to the CPU an array representing the fitness values for each trading rule. From this stage we can further proceed with the remaining operations on the CPU.

The process described above and implemented in the Kernel the FPGA is represented in figure 4.3:

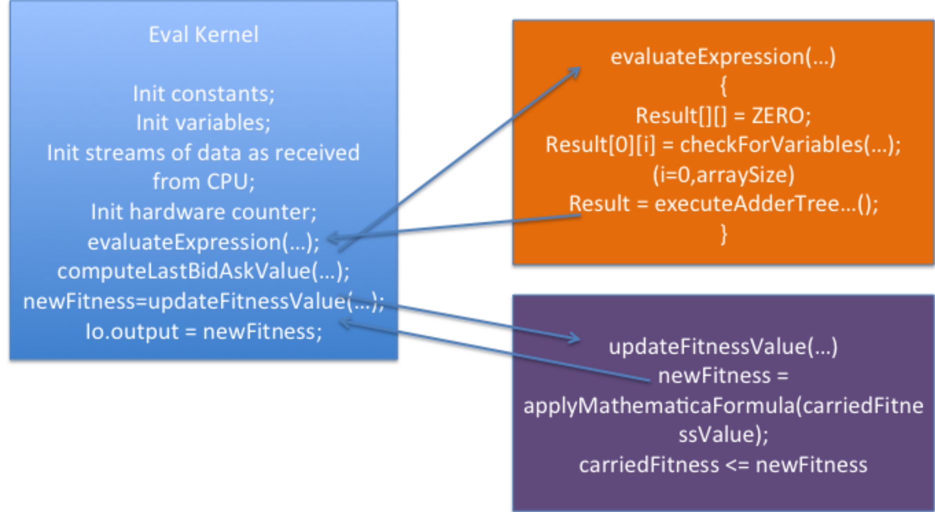

Figure 4.3: Brief overview of implementation design

The hardware acceleration approach we look at is the *streaming model of computation*. In a streaming approach we create a data-flow graph out of simple computational nodes that perform a specific operation on pieces of data pushed in and out of them. Connecting this nodes together creates a pipeline through which one can stream an array of data and get one output per cycle thus achieving high throughput.

4.1.2 Device-Independent Analysis

For computing the acceleration runtime for our application, we use the following formulas:

$$T_{compute} = \frac{CyclesNumber}{ClockFrequency} \tag{4.1}$$

where $T_{compute}$ represents the computation time.

$$T_{transfer} = \frac{DataSize}{Bandwidth} \tag{4.2}$$

where $T_{transfer}$ represents the transfer time.

$$T_{execution} = T_{compute} + T_{transfer} \tag{4.3}$$

where $T_{execution}$ represent the total execution time on the card, including the transfer time.

The formula used for calculating total speedup between two versions of applications is:

$$Speedup = \frac{T_{old}}{T_{new}} \tag{4.4}$$

For each approach implemented on the FPGA we measured the power and energy consumption required for the MaxCard to run our application. The power consumption was measured using a *remote power monitoring* machine which will show how the power value changes when running our application. Thus,

$$DynamicPowerConsumption = RunPower - IdlePower; \tag{4.5}$$

the difference between the measured power value when running our application and the measured power value when the machine is idle, gives us the *dynamic power consumption*.

The *energy power consumption* is calculated with the well known physics formula, taking in account the obtained *DynamicPowerConsumption*.

$$EnergyConsumption = DynamicPowerConsumption * UnitOfTime \tag{4.6}$$

4.2 FPGA Implementation

Usually FPGAs are programmed using a low level hardware description language like VHDL or Verilog, however many tools have been designed that allow a developer to specify high-level designs in a Domain Specific Language (or DSL). We use MaxCompiler, a compiler that lets us specify the computational graph through a high-level Java API, so we focus on the functional aspects of our design and the tool generates a hardware implementation of it.

We are sending three streams of data from the CPU to the FPGA:

The *first stream* represents our trading rule terminals and as presented in the genetic programming implementation, the terminals might contain variables which relate to different market data values within any current evaluated transaction. Because the FPGA does not recognise Strings, we needed to find another method of telling it which variable corresponds to what data in each of the stream array blocks. Thus, we allocated specific constant values for any particular transaction data value (e.g: 100 = bestBid, 101=bestAsk, etc.) such that we can send an array of floating point numbers from the CPU to the FPGA.

The *second stream* represents our trading rule operators and following the same method as above, we will now allocate an integer value for a specific operator (e.g:$0 = -$, $1 = +$, $2 = *$, etc).

The *third stream* contains the transaction specific values and is represented in a floating point number array format.

Our FPGA fitness evaluation consists of three main analytical components:

First, we need to match the existent variables with the respective transaction cost values and we do this in hardware by making use of a multiplexer.

As we can see from the following code snippet what is programmed using nasted if's or a *switch* in C++ will actually be programmed in MaxJ (the Maxeler language used for FPGA implementation) using a multilayer multiplexer represented by a *nasted ternary if* essentially which was used to select between two streams.

```
for (int i = 0; i < arraySize; i++) {
      DFEVar term = inArray[i];
      result[0][i] = (term.eq(100) ? transactions[0] : (term.eq(101) ?
          transactions[1] :
                (term.eq(102) ? transactions[2] : (term.eq(103) ? transactions[3]
                     :
                (term.eq(104) ? transactions[4] : (term.eq(105) ? transactions[5]
                     :
                (term.eq(106) ? transactions[6] : (term.eq(107) ? transactions[7]
                     :
               (term.eq(108) ? transactions[8] : (term.eq(109) ? transactions[9]
                   :
               (term.eq(110) ? transactions[10] : (term.eq(111) ? transactions
                   [11] :
               (term.eq(112) ? transactions[12] : (term.eq(113) ? transactions
                   [13] :
                           term))))))))))))));
   }
```

Second, because one main part of our fitness evaluation implementation is based on a binary expression tree evaluation, we decided to implement an *adder tree* in hardware, which will give back the coresponding binary value of our tree: 1 ("buy" signal) and 0 ("sell" signal). Since the nasted ternary-if operator (? :) can be quite hard to read, we used the MaxJ *control.mux* method to select between different streams of data:

```
for (int i = 1; i <=depth; i++) {
      for (int j = 0; j < width-1; j=j+2)
      {
          result[i][j/2] =
                opsArray[k].neq(-1) ?
                control.mux(
                  opsArray[k].cast(dfeUInt(4)),
                  result[i-1][j]+result[i-1][j+1],
                  result[i-1][j]-result[i-1][j+1],
                  result[i-1][j]*result[i-1][j+1],
                  result[i-1][j]/result[i-1][j+1],
                  result[i-1][j].eq(result[i-1][j+1]) ?  1 : ZERO,
                  result[i-1][j].lt(result[i-1][j+1]) ?  1 : ZERO,
                  result[i-1][j].gt(result[i-1][j+1]) ?  1 : ZERO,
                  result[i-1][j].lte(result[i-1][j+1]) ? 1 : ZERO,
                  result[i-1][j].gte(result[i-1][j+1]) ? 1 : ZERO,
                  result[i-1][j].eq(1) & result[i-1][j+1].eq(1) ?  1 : ZERO,
                  result[i-1][j].eq(0) | result[i-1][j+1].eq(0) ?  1 : ZERO
                  )
                  :
                result[i][j/2];
```

```
            k++;
        }

        width /= 2;
    }
```

Third, after obtaining the solution of the binary tree (trading rule signal) we are now in the position to calculate the fitness on all the transactions and our hardware implementation gives back a result per cycle : the fitness result after evaluating on the first transaction...the fitness result after evaluating on the first and second transactions, until the last cycle result which is the actual fitness of our evaluated individual on all the transactions given as input).

Since in hardware we compute the data as it arrives from the stream (thus pipelining is taking place), and as we already explained, we are received a result per cycle, then if we want to access the result we obtained x cycles ago, we need to connect the result obtained to the stream of results and afterwards we can access it using a *stream.offset* command which works as in the following example which corresponds to a small part of the fitness computation:

```
DFEVar carried_fitness = realType.newInstance(this);

DFEVar fitness = nTransaction.eq(0) ? 1.0 : carried_fitness;
DFEVar newFitness =  bid ? (fitness * (1+opr)) : fitness *(1-opr);

carried_fitness <== stream.offset(newFitness, -1);
```

We used double precision floating point numbers in our FPGA implementation in the beginning, but in the end we tested our application both on single precision and fixed point precision.

4.3 Performance Evaluation

4.3.1 Double/Single Precisions vs Fixed Point Resource Usages

We have performed our initial experiments using three different types of arithmetic precisions and in each case we noticed different resource usages and we could see a speedup increase between the double/single precision implementations and the fixed-point precision. As we will further see, each of these different solutions will require our design to be build at a different clock frequency and using a different number of cost tables (setting up the cost tables number and the clock frequency is a straightforward operation as we can notice from the appendix section .2).

4.3.1.1 Double Precision floating point solution on FPGA

Thus, for the *Double Precision* solution we managed to build and run our design at a clock frequency of 50MHz and using 2 cost tables. The final resource usage as shown in table 4.1 on the FPGA was:

Table 4.1: FPGA total resource usage expressed as a percentage of the total available resource on the chip for double point precision

LUTs	FFs	BRAMs	DSPs	of use
0.29%	0.01%	0.00%	0.00%	stray resources
3.54%	2.40%	5.36%	0.00%	used by manager
44.02%	29.35%	3.34%	8.43%	used by kernels
47.90%	31.76%	8.69%	8.43%	total resources used

We measured the `power consumption` using the formula 4.5, for a total time of 3 seconds and we obtained a value of 17 `Watts`. Thus, according to the 4.6 formula we noticed an `energy consumption` of $51J$.

4.3.2 Single Precision floating point solution on FPGA

We have tried our implementation making use of single precision representation as well and we noticed that the results started to be affected only at the 8 digit, so we had 10^{-7} small errors. This errors were considered unimportant in the scope of the project, since, as noticed from the CPU implementation, usually the trading rule fitness smallest differences are of the order of 10^{-4}, thus our FPGA results are still relevant in comparison with the CPU ones.

For the *Single Precision* solution with the same clock frequency of 50MHz and 2 cost tables, the final resource usage as shown in table 4.2 on the FPGA was:

Table 4.2: FPGA total resource usage expressed as a percentage of the total available resource on the chip for single point precision

LUTs	FFs	BRAMs	DSPs	of use
0.08%	0.01%	0.00%	0.00%	stray resources
3.19%	2.12%	3.48%	0.00%	used by manager
17.25%	10.48%	0.66%	1.69%	used by kernels
20.55%	12.61%	4.14%	1.69%	total resources used

According to the formula 4.5 the power consumption in this case is 13Watts (measured over the same time frame of 3s). So only 4Watts less than in the case of the double-precision solution. Thus, according to the 4.6 formula we now noticed an `energy consumption` of $39J$.

We think this happens because even though the precision is smaller, the FPGA builds a design which deals with floating point numbers, thus it is still limited at computing one result per cycle.

4.3.3 Fixed Point precision solution

We tested our application a different sets of fixed point representations and we obtained good results for the 8 bits representation for the integer part and a 16 bits representation for the fractional part of the real number. This was setup in hardware as a fixed point offset with 8 integer bits, 16 fractional bits and a $TWOSCOMPLEMENT$ characteristic which represents the fact that our fixed-point precision representation for the real number is signed.

Since fixed-point precision increases the pipeline factor, thus being able to compute more computations per cycle, we managed to build and run our application on the FPGA with a clock frequency of 100MHz and 5 cost tables. The final resource usage as shown in table 4.3 on the FPGA was:

Table 4.3: FPGA total resource usage expressed as a percentage of the total available resource on the chip for fixed point representation

LUTs	FFs	BRAMs	DSPs	of use
0.08%	0.01%	0.00%	0.00%	stray resources
2.85%	2.12%	3.48%	0.00%	used by manager
17.29%	10.48%	0.66%	1.69%	used by kernels
20.25%	12.61%	4.14%	1.69%	total resources used

As we can see from the total resource usages results, there is definitely an improvement between the double point precision and the fixed point precision representations.

We can notice a smaller power consumption values as well, as in the fixed-point case this is equal to just 10Watts so with 7Watts less than for the double-precision solution. Thus, according to the 4.6 formula we now noticed an `energy consumption` of $30J$. We think of this as a good result compared to the double-precision one, as the energy consumption decreased by 42% and we explain it as being relevant since with the fixed point precision representation we are able to obtain multiple results per cycle, in comparison to the double/single floating point evaluation.

Also, judging by the resources utilisation, we can observe that for the fixed point precision we will be able to further parallelize our hardware implementation: since we aim for using less than 90% chip resources at any time, we will be able to build our design using 3 pipes (4 in the best case scenario) while using DRAM. So, in this situation we will be able to increase our current speedup by 3X (or 4X).

Formally, building our design on 3 hardware pipes means that we can compute the results for 3 transactions in the same time, compared to the method we have implemented now when we compute the result for one transaction at any time. This will be possible since our design is not memory bound (*Memory bound* refers to a situation in which the time to complete a given computational problem is decided primarily by the amount of memory required to hold data), presumably letting us obtain 3 results per cycle, thus from hardware points of view, our execution time will decrease by 3 times the current time.

4.3.4 Acceleration Measurements

We test our application on the following machines:

`FPGA:Maxeler card MAX3A Vectis (P/N: 13424) (S/N: 1801010006) Mem:24 GB`

`CPU: Intel(R) Core(TM)` $i7-2600$ `CPU @ 3.40GHz, 8GB RAM`

For the hardware build, when running our application with a total of 150 expressions (with their 150 respective operators) and 200,000 transactions, each expression, operators, transactions being represented by an array of 16 single precision floating point elements.

We consider evaluating trees with a maximum depth of 4 (for our hardware tests), thus the $2^4 = 16$ total elements for the expressions and $2^4 - 1 = 15$ operators array. In this case, each of our transactions CPU array contains 14 elements, and each of our CPU array contains 15 elements, but we aligned it to 16 elements, thus introducing to elements equal to -1, which do not represent anything for our evaluation (as we have seen in the code snippet with of the FPGA implementation presented above, we do not take in consideration the array elements equals to -1).

Given that the PCIe theoretical bandwidth on MAX3A cards is $2GB/s$, and first considering the amount of data we need to transfer for one iteration from CPU to FPGA as being: 150 trading rule terminals, 150 trading rule operators and 200,000 transactions, each of 16 single-precision elements each (4 bytes each element), and the

amount of data we need to receive per iteration as being: CyclesNumber*DataSize (we send one result per cycle), then we will have the following estimations:

$$T_{transfer} = (\frac{(TerminalsNumber * sizeof(terminals)}{Bandwidth} + \tag{4.7}$$

$$\frac{OperatorsNumber * sizeof(operators)}{Bandwidth} + \tag{4.8}$$

$$\frac{TransactionsNumber * sizeof(transactions))}{Bandwidth} * \frac{sizeof(eachTransferedElement)}{Bandwidth} \tag{4.9}$$

Thus our expected transfer time is:

$$T_{transfer} = \frac{(200,000 + 150 + 150) * 16 * 4 + (150 * 200,000) * 4}{2 * (1024^3)} = 0.117s \tag{4.10}$$

Our total number of cycles is represented by $TransactionsNumber*ExpressionsNumber$ since we want to obtain one result per cycle for each evaluated expression (one expression contains terminals and its respective operators). When evaluating for the single-precision floating point hardware implementation, we run the design at a 50MHz clock frequency, thus the expected computation time is:

$$T_{compute} = \frac{200,000 * 150}{50 * 10^6} = 0.6s \tag{4.11}$$

Therefore, the total execution time for our hardware application is:

$$T_{execution} = 0.6 + 0.117 = 0.6117s \tag{4.12}$$

When running the application on the `CPU` we notice that the total execution time for the same tested amount of data is equal to 4.87`s`.

If we calculate the total speedup using formula 4.4 then we obtain in this case a total speedup equal to $4.87/0.6117 = 7,96x$.

When computing the same calculus but for the fixed point solution of the hardware implementation, which is running on an 100MHz, we've got a computation time equal to $0.3s$ and thus a total new execution time equal to $0.3117s$. So our new speedup will be $4.87/0.3117 = 15.62x$. Thus, under the given tests our improved version of `hardware implementation` using the fixed-point precision runs 2`x` faster than the single-precision version and 15.62`x` faster than the CPU version.

As stated previously, the fitness evaluation part of our genetic program takes 78% of the execution time for the CPU solution. Thus if we consider being able to optimally accelerate all this part and theoretically reduce the computation time for the fitness evaluation to $0s$ with the FPGA help, then we would have reduced our overall CPU execution time by approximately 4.4 times. Since in our test case scenario with the best implementation version, namely fixed-point precision, the FPGA execution time for the fitness evalution is $0.3117s$, then our overal CPU execution time is reduced by approximately 3.6 times. This means we are 81.81% near the maximum optimal speedup result.

4.4 Challenges and Further Improvements

4.4.1 Challenges

The lack of experience with FPGAs made me spend more time that I should have on the actual implementation. From a programming point of view, I had to understand how a whole new architecture works and how to think in a completely different, highly parallelized and streamed way, in comparison to the conventional software programming. Also, working with new tools gave rise to new errors and more hours of debugging.

One main challenge was to actually get the same exact results on the FPGA as on the CPU. This raised me opportunities to adapt my design such that I minimise the result error apparitions. But this was a challenge, because often on FPGA if I chose the high exact arithmetic precision (double precision floating point) I lost design efficiency.

Another challenge is to get something running on the actual hardware. Sometimes the design is perfectly working in the Maxeler Simulator tools but when one wants to build it for hardware, either it might not be supported because it exceeded the number of resources available, is not parallelized enough or the FPGA clock frequency is too high for the design to be build on. This was hard in the beginning because of the new style of debugging. Except of the actual possibility of having the well-known *printfs* everywhere in the code, there is not tool such as *gdb* provided. The explicit results that we get from debugging on the FPGA can be found after building a graph of memory allocation which also shows the communication flow of our application. An example of such a graph is the figure 4.4

4.4.2 Further Improvements

As we already presented, we currently are in the situation where we could not test our program on all data we were testing on CPU because of the limited PCIe data transfer bandwidth. Thus, one major improvement which can be added on our design, is to make use of DRAMs.

MaxCards have a large external LMem resource available. This allows large amounts of data to be kept local to the board and iterated over by the DFE. The LMem appears as one contiguous piece of memory. Both the Standard Manager and the Manager Compiler allow us to connect multiple streams to the LMem on the MaxCard (see 4.5).

Using LMem we will improve our data transfer bandwidth as well as we can actually transfer only once all of the transactions we want to evaluate their fitness on and store them internally on the FPGA's DRAM (LMem). With this design we could send from CPU to LMem N trading rules such that we send a trading rule, we evaluate it on all the transactions kept on LMem and store its result on LMem. At the end we will send back from LMem one stream containing all the results after the trading rules' fitness evaluation.

Thus we will send the transactions from CPU to LMem only once, at the beginning of our computation avoiding the big transfer time (which is necessary to send them

every time with a new iteration) as being present on the PCIe-only implementation. So with this implementation, we will write *once* the transactions (whom values don't change over one run of the algorithm) on LMem remaining that at every iteration to write on DRAM a new population of individuals which is next to be evaluated. Putting our application on LMem which is quite straightforward to do, but unfortunately we have runned out of time to actually finish the implementation for this improvement. So, in case we have our application running on LMem and because of the low number of resources used, we can further optimise our application as explained before, by making use of 3 pipes. The formula 4.1 becomes:

$$T_{compute} = \frac{CyclesNumber}{PipesNumber * ClockFrequency} \tag{4.13}$$

So, according to the formula 4.13 and knowing that the time of transfer remains the same, if we use a number of 3 pipes, from the obtained 15.62`x` speedup (fixed-point precision implementation) we should expect a new speedup value of 43.59`x`.

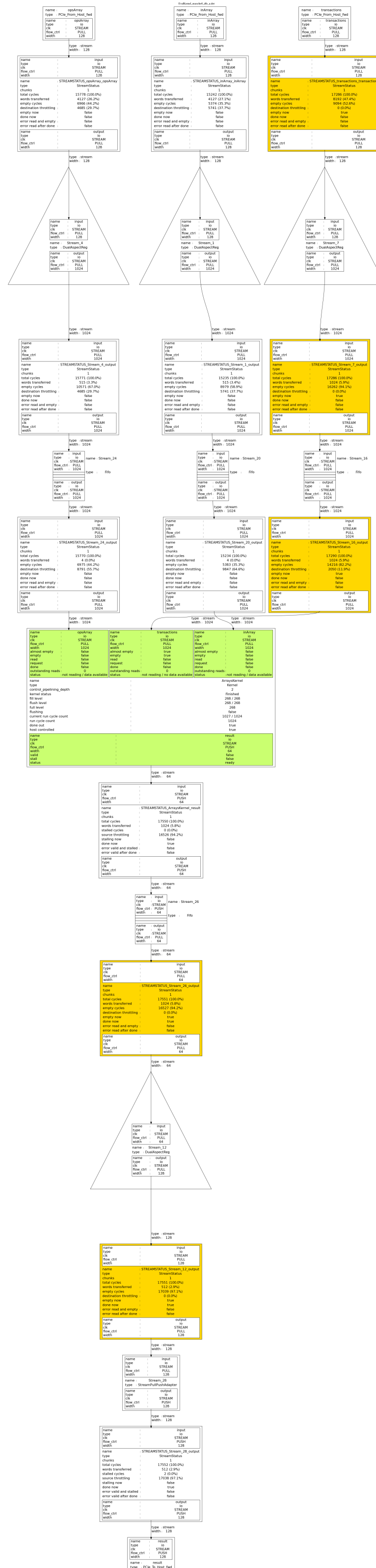

Figure 4.4: Graph resulted when debugging our fitness evaluation application

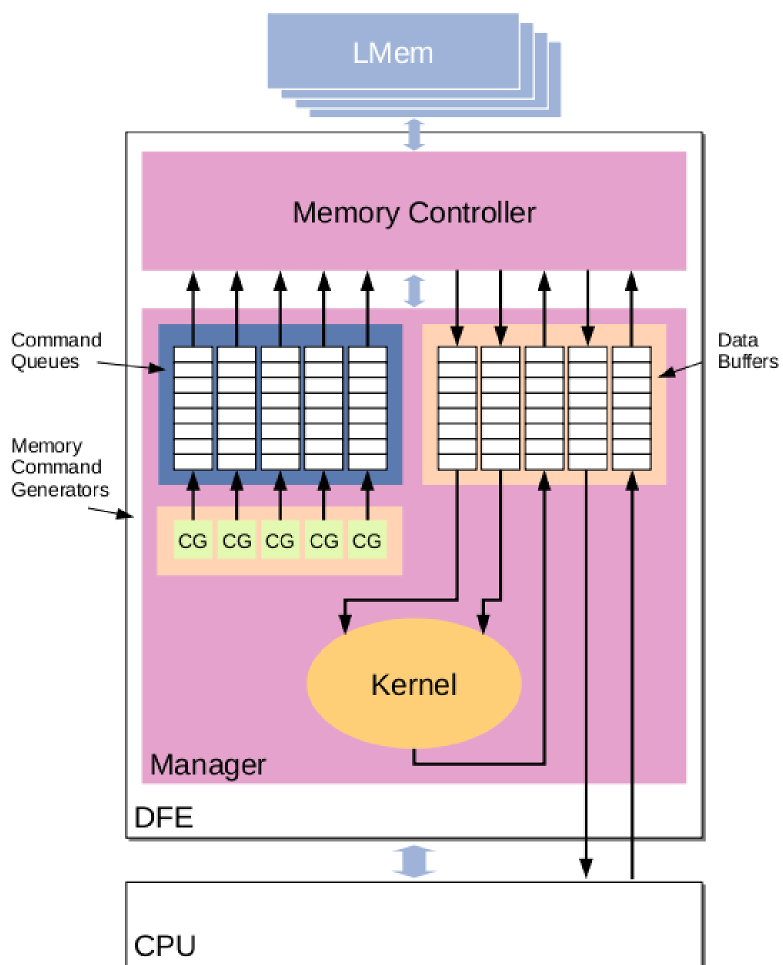

Figure 4.5: LMem controller architecture

4.5 Summary

Throughout the course of this chapter we have presented additional background information about hardware runtime analysis, FPGAs power consumption as well as different Maxeler tool specifics.

We have looked at the design flow and the implementation procedure used to build our design for the FPGA.

Furthermore, attention has been paid on the total number of resources used when implementing solutions with different arithmetic precision format for the real numbers representation (double/single precision and fixed-point precision) and we concluded that the fixed-point precision representation will not only offer correct results, good performance but will consume less resources as well.

We have performed the acceleration measurements for our design and compared the results with our CPU implementation.

Finally, we have also discussed the main challenges of building an application for FPGAs, as well as looking into further improvements that our application can benefit of, such as the memory wrapped DRAM with which we would expect a total speedup of 43.59**x** in comparison to the original CPU version.

Chapter 5

Testing for Robustness

In this chapter we will briefly introduce the most important information necessary to understand the tests we performed to evaluate the robustness of our program. The topics that will be discussed include:

- `Robustness:` Offers a short overview into what robustness theory represents.
- `Individual Testing:` Presents how many times does the best trading rule come out on top of all fit individuals, after a series of tests runned.
- `Aggregate Testing:` Shows how many times the best trading rule from one period appears in the top best set of performing strategies from the remaining two periods of time.
- `Rejection-Acceptance Testing:` Presents some testing results of how many times a rejected strategy will be accepted in the next round of testing if introduced again.
- `Algorithm behaviour in noisy data presence:` Shows how well the algorithm performs when noisy data is introduced into our original training and testing data.

5.1 Robustness

Definition 8. *In computer science, robustness is the ability of a computer system to cope with errors during execution or the ability of an algorithm to continue to operate despite abnormalities in input, calculations, etc.*

The harder it is to create an error of any type or form that the computer cannot handle safely the more robust the software in test is. Robustness is a consideration in failure assessment analysis.

We use evolutionary learning to try and get closer to a solution for one of the most researched problems in the financial sector: finding trading patterns in the market data. From old times to nowadays, traders have tried to identify patterns for the market positions under different regimes in time and with different characteristics, but they did not prove very successful. Thus our chose for a machine learning program. Even the technical indicators that traders and strategists discovered do not prove reliable under all market conditions they developed them for. Thus, one important thing we need to prove for getting our program to a stage where traders can trust it as being a reliable source of trading rules information, is to test for its robustness. And we did this in two different ways: by means of aggregate testing and by inserting noisy market price data in the training as well as testing data. Both of the methods will be presented in more detail later.

A large fraction of studies on genetic algorithms and genetic programs emphasise finding a globally optimal solution. Some other investigations have also been made for detecting multiple solutions. If a global optimal solution is very sensitive to noise or perturbations in the environment then there may be cases where it is not good to use this solution, thus the need for testing for the robustness property is an acute one in our approach.

First step in our testing strategy was to split the 2003 and 2004 testing data-sets, each into three different sets (three months period of time split into each individual month) such that we can identify any possible market price patterns inside of the data, as well as being able to notice if our genetic program gives consistent results. Of course different market conditions will lead to different trading rules, but three month consecutive period of time can be enough for any possible trends ("buy/sell" periods of time) identification.

Because of the nature of our genetic program, we consider a trading rule to be the same if the mathematical expression behind the trading rule tells us the same thing (we will thus ignore different, nonsemnificative constants, by filtering them out from our tests - we care about the best fit rules coming up with the same idea, rather with the exact same numbers).

Some of our tests are written in C++ and are integrated within the big program, such that one can run all the test or just independent ones, every time it trains and tests the genetic program on some input market data. The remaining ones are written in MATLAB.

5.2 Individual testing

This is the first robustness test that we performed and it evaluates in a run of let's say X tests each of N iterations, how many times the best trading rule comes out on top (we define top as being the first 15% of the fittest trading rules), by how much (measure of size) it is better than the second best trading rule (we compare two trading rules by comparing their best average score). We need to point out that at the beginning of each individual test, we start with a new population initialised using the ramped half-and-half method. For each test, after the number of iterations is finished, we end up with a best strategy/best percentage of strategies, which we use then out-of-sample.

Table 5.1: 2003 individual robustness testing

Iterations	Jan (First/Top)	Feb (First/Top)	March (First/Top)	Jan-March (First/Top)
1000	87/127	99/114	86/119	87/121
900	95/132	101/113	98/123	92/114
800	100/122	95/108	92/117	94/123
700	89/109	83/98	96/110	85/119
600	72/112	81/102	67/103	64/107
500	83/118	70/99	66/103	72/95
400	75/109	68/103	72/86	61/84
300	67/96	70/97	58/68	54/76
200	61/86	61/93	54/64	49/78
100	56/88	58/79	54/63	41/66

The results from table 5.1 were obtained after we evaluated 150 times a population size of 150 individuals. What the table tells us is how many times the best strategy after evaluating out-of-sample using our genetic program on different number of iterations is coming up first or in the top 15% best strategies out of 150 individuals (for different time periods).

We notice that the January trading rule maintains its ranking pretty well, being present as the best trading rule/part of the best trading rules over 50% of the time for most of the tests. The only exceptions are for the tests performed on 300 iterations or less. But this is acceptable since, by its nature, a genetic program is expected to perform better when running for a large number of iterations, compared with a low number. We further observe that February and March trading rules give similar appearance results, which further accentuates the idea that our genetic program is systematically selecting a trading rule as the one which performs best and continues to find it most of time,

even when new populations have been generated, being first or at least in the top 15% trading rules.

When analysing the trading rule which comes as a pattern on a bigger data sample, we notice that the results start to deteriorate a bit, and we get less than 50% of the time top appearance, starting with 600 iterations or less. This result was predictable since using a larger amount of dataset, split over a longer time period (now three months compared to one month) will introduce regime specific noise which can cause the program to switch between the best fitting trading rules.

Thus, judging from the results, we can assume that if we were able to run the program for larger number of iterations over the same three months we would be able to get more robust results. We were unable to do this unfortunately, because of the nature of the GP and the CPU computation time limitations.

We present below, in the table 5.2 the difference between the best and second best strategy computed in the same manner as in the above table 5.1.

Table 5.2: 2003 individual robustness testing - first strategy better than the second strategy

Iterations	Jan	Feb	March	Jan-March
1000	0.01230	0.00793	0.05608	0.03809
900	0.09134	0.00063	0.00402	0.10682
800	0.05562	0.00458	0.13705	0.00053
700	0.10442	0.08805	0.08702	0.00561
600	0.02901	0.07803	0.13804	0.05770
500	0.07870	0.12330	0.07902	0.10005
400	0.17664	0.00091	0.03470	0.10097
300	0.00079	0.11034	0.06993	0.13098
200	0.07459	0.15894	0.23780	0.09371
100	0.20870	0.00683	0.17665	0.11082

What we notice is that there usually tends to be little difference between the two best trading rules, thus our results from table 5.1. This is so, since with little difference between best fit strategies, and adding the different time period and market conditions there is a high probability to encounter ranking switches between them.

Thus these ranking switches should not be taken as being errors in the genetic program, but what this suggests is that our trader should take in consideration a list of top (10%, 15%) strategies given by our GP, when adapting its market position.

We performed more tests on the 2004 data, but we noticed that the results decreased considerably and we can only explain this ourselves as being due to different market conditions affecting the market prices (different regime, different genetic programming behaviour).

From table 5.3 we notice that the program's robustness increases with the number of iterations, thus, we think that one solution for making the genetic program more trustworthy would be to run it for a larger number of iterations.

Table 5.3: 2004 individual robustness testing

Iterations	Jan (First/Top)	Feb (First/Top)	March (First/Top)	Jan-March (First/Top)
1000	76/99	81/102	78/96	87/106
900	71/94	76/97	89/91	75/99
800	63/81	61/90	73/94	67/91
700	55/83	53/78	64/96	59/78
600	50/72	55/76	60/88	61/91
500	43/64	48/68	52/80	57/79
400	46/71	43/60	61/86	52/74
300	40/62	38/54	57/73	49/64
200	36/65	31/49	52/69	44/61
100	34/63	28/44	54/72	42/55

5.3 Aggregate testing

In our approach, aggregate testing means that we evaluate how many times the best trading rule from January appears in the top 15% of the best performing strategies from February and March (measure which can be found in the table, for example for the January trading rule, under the column named Feb/March, and similarly for the other strategies (in a testing set of January, February, March individually and all three months together).

After evaluating again the 2003 market data on a population of 150 individuals, with a test size of 150 separate tests, each with a different number of iterations (e.g: 500, 200 iterations, etc.), we obtain the results presented in table 5.4.

Table 5.4: 2003 aggregate testing

Iterations	Feb/March	Jan/March	Jan/Feb
1000	137/118	142/108	136/120
900	140/129	134/104	128/114
800	127/102	130/103	122/119
700	119/108	124/105	123/113
600	122/98	117/92	117/103
500	112/101	125/108	111/94
400	117/92	121/99	115/92
300	110/94	104/98	102/88
200	103/86	97/93	95/84
100	101/92	91/86	87/81

From table's 5.4 results we can observe that in all of the tested cases the best trading rule from one month comes up in the top 15% best trading rules from the other months in over 50% of the cases. So, we can conclude that even though the best trading rule for one time period might not naturally maintain its ranking over the remaining individual time periods (as we noticed in the results from the section 5.2), it will come up in the top ranked strategies most of the time. Thus, our proposed idea for the trader to consult the top set of best fitting trading rules given by our genetic program gains credibility. However, since market data can arise from different regimes, it might leads us to increased/decreased performance/robustness characteristics in our GP. For example, we notice that we find different results after performing the same test on the 2004 market data values.

After analysing the results from table 5.5 in comparison to the ones from the table 5.4 we can identify, without a deep analysis, that the market prices have bigger fluctuations

Table 5.5: 2004 aggregate testing

Iterations	Feb/March	Jan/March	Jan/Feb
1000	107/98	122/92	102/110
900	109/101	106/98	92/101
800	102/92	99/92	104/88
700	106/100	101/105	103/91
600	99/89	94/97	107/91
500	100/91	87/83	94/81
400	86/92	78/86	87/84
300	81/83	72/77	82/73
200	71/79	74/69	67/62
100	66/75	68/60	59/51

in 2004 compared to 2003, thus the weaker appearance of the initially best trading rule, in the out-sample best fitting trading strategies that resulted at the end of our testing.

5.4 Rejection-Acceptance testing

Under this section we will present a more different way of testing our genetic program for robustness.

Naturally, our genetic program runs for a number of iterations (manually set by us) or until the convergence criteria is met (e.g: the profitability measure is above a given threshold). When we want to perform out-sample testing on data then we run our genetic program for a fixed number of times, following the same specific convergence criteria inside of each test:

Thus, for example, we run our algorithm X times, each time for N iterations. At the end of each iteration we get the new population of trading rules sorted by their fitness. Let's assume we are at the end of the first iteration: the way we move to the next generation is by accepting the best half of this new population to move on and rejecting the least fit half of the population. When we will move to the next generation we will inject the best half of the population and we will perform crossover, mutation and reproduction on those individuals, until we will get to the initial wanted trading rules population size. Then, arriving in the next iteration (second in our case) we calculate the new individuals fitness and at the end of this current iteration (second in our case) we get back a new population sorted by the individuals fitness, as so on for N iterations. At the end of the first test, we save the best strategy or a percentage

of the best performing strategies. These best fit strategies will be tested at the end of the final test-run, on the out-sample market data to see their overall cumulative profit performance.

Our present test checks whether or not the least fit strategies come back and are selected again in the best population half, if we put them back in the next iteration after rejecting them at the end of the previous iteration. This is tested in two stages: we test the rejection-acceptance for each iteration measured at the end of the N iterations (thus at the end of one test) and then we measure the whole rejection-acceptance issue at the end of the all the tests that we want to perform.

Keeping the same market data testing split choice, we will evaluate our genetic program on a different number of iterations for different number of tests.

Briefly, let's assume we have a population of X individual trading rules to be tested. At the end of each iteration we will split this population into the best fit half (accepted for performing GP operations next) and the least fit half (rejected to go into the next iteration).

The calculus for our particular test has been made as follows:

Let's say we have X individuals in the population and following the simple rejection-acceptance strategy presented above, we chose to reject $X/2$ individuals from the population in the first iteration, then we introduced those originally rejected individuals in the next iteration and we note how many we now accept m ($m < X/2$) individuals (as being part of the best half of the population) from the ones rejected in the previous iteration.

Thus, for a total of N iterations we will have:

$$TotalAccepted_N = \sum_{i=0}^{N-1} i * itAccepted_i \tag{5.1}$$

where $TotalAccepted_N$ represent the total number of individuals being accepted in the next iteration after originally being rejected in the previous iteration after a total of N iterations; $itAccepted_i$ stands for the number of individuals being accepted after one iteration only, and i represents the iteration number.

Therefore, we will find ourselves with a number of $TotalAccepted_N$ individuals out of the total number of possible individuals to be accepted which is represented by $X/2*N$. Following this, our algorithm is more robust if the percentage of $TotalAccepted_N$ individuals is lower, and less robust if there's a big number of individuals previously rejected being accepted again.

The 2003 and 2004 results presented in the tables 5.6 and 5.7 were obtained using a population of 150 individuals tested once for each of the X (X = 100,...,1000) iterations. So, the numbers are computed using formula 5.1.

We can observe that after testing on the 2003 data we re-accept least fit individuals in a percentage ranging between 16.40% and 50.88%, over all of the different tests with different number of iterations.

Table 5.6: 2003 rejection-acceptance testing

Iterations	Jan	Feb	March	Jan-March	Jan-March%
1000	28,803	22,889	30,055	81,747	36,33 %
900	33,004	27,048	37,059	97,111	47,95%
800	30,559	29,044	31,997	91,600	50,88%
700	27,230	28,220	23,569	79,019	50,17%
600	20,301	19,887	17,750	57,938	42,91%
500	13,230	13,565	15,556	42,324	37,62%
400	12,430	9,443	12,289	34,162	37,95%
300	8,001	6,440	9,547	23,988	35,53%
200	4,305	1,406	3,778	9,489	21,08%
100	1,579	804	1,307	3,690	16,40%

Naturally, having a bigger number of iterations will raise the probability of having least fit individuals selected, which we can notice to be present on average in our case as well. But, even if the probability is higher our program should not permit many of this least fit individuals coming back into the best half of the population all the time. What this all indicates is the nondirectional, stochastic nature of the genetic program search.

Since our worst result comes when we performed the tests over 700 iterations (a high number of iterations) and this is just a little bit over 50% then we take the results as being acceptable and we follow on with further testing on a different set of data because, as presented in the introduction, different market regimes were noticed to affect prices over time, so they might affect our genetic program performance and/or robustness as well.

We notice that for the 2004 data set we re-accept least fit individuals in a percentage ranging between 28.01% and 55.55%, over all of the different tests with different number of iterations. Now our worse result comes as well when we performed the tests over 700 iterations and this is just a bit over 50%. Taking it as a coincidence or not, we can still consider it an acceptable result given the undirected nature of the genetic algorithm search.

One other idea coming up from these results is that, our rates of acceptance for the least fit strategies are higher now and this might suggest that for 2004 the potential market data price patterns might be harder to find, if not in-existent (we cannot be sure at any time that a pattern actually exists, until we are convinced on the presence of the trading rule/set of trading rules giving profit returns).

Table 5.7: 2004 rejection-acceptance testing

Iterations	Jan	Feb	March	Jan-March	Jan-March%
1000	43,324	35,904	41,048	120,246	53,44 %
900	37,896	32,098	41,254	111,239	54,93%
800	31,804	30,659	31,231	93,694	52,05%
700	30,807	29,006	27,689	87,502	55,55%
600	18,258	19,640	18,940	56,838	42,10%
500	15,059	13,282	17,885	46,226	41,08%
400	12,980	10,413	14,689	38,082	42,31%
300	13,783	8,982	14,004	36,769	54,47%
200	5,781	3,716	4,018	13,515	30,03%
100	2,240	1,873	2,190	6,303	28,01%

5.5 Algorithm behaviour in noisy data presence

This test consists of taking the best fit in-sample trading rule and test it on the out-sample data with and without different levels of random noise added in.

The *way in which we add random noise* is:

We take the out-sample market data values and measure their mean and standard deviation. Afterwards, we add inside the original data different amounts of noise market data price values, following a *Normal distribution* defined by the formula:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{5.2}$$

where the parameter μ in this formula is the mean or expectation of the distribution (and also its median and mode). The parameter σ is its standard deviation; its variance is therefore σ^2.

We then vary the standard deviation keeping the prices mean fixed and generate new price values using again the *Gaussian Distribution.*

We continue to vary the standard deviation value and the amount of noise added until we see that our selected trading rule stops predicting and starts giving bad results. This is a different way of testing how reliable our genetic program is and how bad its performance is affected by the market data prices fluctuations.

5.5.1 Results

We performed the test on the best trading rule resulted from an out-sample "January-March" period of time, rule which has a performance of 1.17213 over 150 tests with 1000 iterations each. The buy/sell prices' mean for the tests are 1.61065 and 1.61075 with their original standard deviation (std) being: $6.99e-05/7.41e-05$.

In the table presented below, the last four columns represent the performance results of the same trading rule applied on the original out-sample market data but with a specific random noise percentage addition, randomly distributed within the data (the random noise amount can be found in the first column of the table).

Is is important to note that these results are orientative, as if we run the tests again we will obtain different but quite similar performance measures (because of the random distribution of the new price values inside the market data).

Table 5.8: 2003 noise behaviour - performance measure

Random Noise Amount	Test Std	Test Std	Test Std	Test Std
	0.0001	0.001	0.01	0.1
50%	0.088953	-0.203530	-0.650471	-1.048890
25%	0.454831	0.128904	0.003891	-0.378158
10%	1.119038	1.005998	0.899640	0.639034

The table 5.8 shows what we would have expected more or less to see. If we add little random noise inside the out-sample prices, then our GP resulted best trading rule performance will start to decrease while we increase the standard deviation value inside the Gaussian distribution price generator.

We notice that for a value of 0.0001 standard deviation prices modification we get a very close performance result to the original one (1.119038 and original was 1.17213), while for the 0.1 standard deviation value we obtain a least performant result, 0.639034.

We can also see that while we increase the amount of noise added to the market prices, the best strategy performance decreases significantly : for 50% noise we get a value of 0.088953 compared to the original one, 1.17213.

From a further look from the left to right in the table, we see that we can get from a best fit strategy to one which in other circumstances would have been considered a least fit strategy: for 50% noise and 0.1 standard deviation, we obtain -1.048890, so not only that we do not gain any profit, but we are loosing money at this stage!

Thus, the above table managed to prove that most random prices addition which permits us not to be loosing money is at 25% noise level and 0.01 standard deviation.

The table 5.9 presents the numerical position of the original best strategy (compared to the other top best strategies for a population of 150 individuals), after the random addition of noise:

Table 5.9: 2003 noise behaviour - performance ranking

Random Noise Amount	Test Std Pos	Test Std Pos	Test Std Pos	Test Std Pos
	0.0001	0.001	0.01	0.1
50%	17	34	42	58
25%	9	23	29	36
10%	6	16	21	28

From table 5.9 we can see that if we keep a low level of noise 10% then our original best trading rule ranking remains in the top 20% best fit strategies regardless the increase in the standard deviation.

Again, if we increase the level of noise together with the standard deviation, we observe a noticeable decrease in the ranking of what it was original the best strategy. Now, the ranking ranges between top 11.33% (best ranking position for noise level 50% and standard deviation 0.0001) to 38.66% (best ranking position for noise level 50% and standard deviation 0.1).

Thus, looking now in this table at the corresponding point, which still permits us not to lose money, of the one presented in table 5.8 at 25% noise level and 0.01 standard deviation, we see that the original best trading rule is in the top 20% (19.33) best performing trading rules. This might express that when our best performing trading rule does not start losing money, the trader can still take it in consideration if he looks at the predicted top 20% best performing strategies.

5.6 Summary

Throughout this chapter we introduced the basic theory related to what it means for a program to be robust and then followed up with more specific mathematical tests such as *aggregate testing*, *rejection-acceptance* testing, etc.

The results of this tests are important in determining the algorithm reliability as well. For example, it is important to know how many of the originally rejected trading rules, will come back as being accepted in a next iteration when introducing them again in our genetic program, because in this way we can actually make an approximation of how much we can "trust" our program solutions. Thus, we have shown that in the majority of cases, on the data we tested our algorithm on, we do not accept many of the originally rejected trading rules and thus our program can be considered reliable.

We concluded our robustness testing by introducing white noise inside of the original testing data using a Gaussian Distribution way of sampling random similar prices.

Chapter 6

Further Statistical Results

In this chapter we will briefly introduce the most important information related to the tests we performed for evaluating our program. The main topic that will be discussed shows a comparison between the two different approaches we have implemented, using statistical and economical conclusions:

`Genetic Programming vs Hybrid GP/PSO:` shows a number of statistical tests we performed for evaluating the profitability/predictability and reliability of the algorithm; offers a comparison overview of the actual returns and the predicted returns given by our program and shows the economical significance of our results according to Anatolyev-Gerko statistics test.

6.1 GP vs Hybrid GP/PSO

We employ a series of statistical tests to test for profitability and predictability of each of the trading strategies. We split again each trading period into two parts that serve as in-sample and out-of-sample periods respectively.

With the genetic algorithm-based strategy, we choose the trading rule that produces the best in-sample performance and test its profitability out-of-sample. We want to generate and empirical distribution of the out-of-sample cumulative returns and thus, we run this test 100 times independently. Because of the stochastic nature of the genetic program search, this provides us with potentially 100 different trading rules and their out-of-sample performance.

Table 6.1: 2003-2008 Genetic Program individual returns

Iterations	Jan (20-24) 2003	Feb (17-21) 2003	March (10-14) 2003	March 31, 2008
1000	1.142	1.094	1.003	0.991
900	1.180	0.934	0.992	0.875
800	1.032	1.012	0.850	0.786
700	1.0018	0.996	0.804	0.838
600	0.903	0.855	0.739	0.704
500	0.821	0.747	0.638	0.663
400	0.845	0.684	0.714	0.593
300	0.712	0.784	0.682	0.588
200	0.799	0.684	0.709	0.549
100	0.625	0.633	0.602	0.498

6.1.1 Individual Returns comparison between GP and GP/PSO

Before actually introducing the statistical results obtained after performing the *t-test* we would like to perform a small comparison in terms of the best fit strategies returns for the original genetic programming version and the improved genetic programming with particle swarm optimisation version.

The table 6.1 shows best fit trading strategies returns that appeared after different number of generations for the genetic program version.

From this results we can notice a decrease in return from January to March 2003 and also to March 2008. We can explain the 2003 results as a seasonal trading phenomena, and this might happen because of the different market conditions belonging to different financial regimes that can affect our algorithm in such a way in which it won't be able to find good patterns in the data. Also, we notice that the 2008 market crisis affects our returns as well, thus we obtained less good results that from the 2003 data.

The table 6.2 shows best fit trading strategies returns that appeared after different number of generations for the genetic program including the PSO optimisation version.

We can draw the same conclusion as before from the 6.2, but we now notice an increase in the overall GP/PSO returns and this happens because of the particle swarm optimisation addition which allows our program to perform the genetic operations only on the best fit individuals, thus converging to better trading rules in general.

Table 6.2: 2003-2008 GP/PSO individual returns

Iterations	Jan (20-24) 2003	Feb (17-21) 2003	March (10-14) 2003	March 31, 2008
1000	1.273	1.202	1.185	1.044
900	1.193	1.174	1.103	0.991
800	1.075	1.086	1.032	0.885
700	0.994	0.928	0.965	0.904
600	0.879	0.884	0.923	0.915
500	0.800	0.823	0.849	0.811
400	0.756	0.773	0.724	0.702
300	0.711	0.701	0.694	0.724
200	0.663	0.687	0.688	0.613
100	0.649	0.632	0.641	0.578

6.1.2 T-statistics tests

We can now perform a *t-statistic* to test if the mean of the returns is significantly different from zero, as explained in the background chapter.

We start by testing the genetic program's profitability under different number of iterations until converge. Regarding this, table 6.3 shows using the t-test (at 5% level significance) for zero mean that the genetic program can handle transaction costs well, even when the number of iterations of the algorithm are low. Please note than in all of this cases the "inertia band" k is equal to 0, thus we allow the traders to perform all the possible trades. This table presents the out-of-sample percentage of cumulative returns, generate for the genetic program for the 2003 different sample periods (percentage here means the action of the returns to be adjusted to a daily basis and expressed in percent) and one period of 2008 (due to lack of sample data)

We performed the above same test on the GP/PSO program as well and we obtained different results (as table 6.4 shows) than in the case of the classic genetic programming implementation. Even though the results are different, we don't see results much more statistical significant.

We now perform the same t-statistics test both on 2003 and 2008 data, but we keep the genetic program number of iterations fixed at 1000 and we vary the "inertia band" parameter k within the range 0-20. In our case, we measure the filter parameter k in *basis points* (bp = unit equal to one hundred of a percentage point). For example, a difference of0.10 percentage points is equivalent to a change of 10 basis points (e.g., a 4.67% rate increases by 10 basis points to 4.77%) [35]. We again evaluate out-sample the best-performing strategy in-sample and we obtain the results shown in table 6.5.

Table 6.3: 2003-2008 Genetic Program t-statistics

Iterations	Jan (20-24) 2003	Feb (17-21) 2003	March (10-14) 2003	March 31, 2008
1000	1.630	0.810	0.323	0.592
900	1.628	0.808	0.344	0.584
800	1.629	0.809	0.290	0.593
700	1.582	0.790	0.283	0.589
600	1.591	0.784	0.314	0.583
500	1.603	0.795	0.296	0.592
400	1.587	0.780	0.288	0.572
300	1.593	0.772	0.293	0.580
200	1.580	0.766	0.274	0.572
100	1.573	0.763	0.264	0.567

Table 6.4: 2003-2008 GP/PSO t-statistics

Iterations	Jan (20-24) 2003	Feb (17-21) 2003	March (10-14) 2003	March 31, 2008
1000	1.683	0.832	0.358	0.618
900	1.679	0.838	0.350	0.616
800	1.681	0.829	0.355	0.607
700	1.670	0.825	0.348	0.613
600	1.664	0.819	0.345	0.602
500	1.667	0.827	0.339	0.593
400	1.653	0.812	0.340	0.587
300	1.648	0.806	0.332	0.594
200	1.644	0.786	0.328	0.583
100	1.636	0.792	0.321	0.579

The results obtained in table 6.5 for the 2003 data indicate that the systematic pattern that seems to exist inside of the market data values as noticed in table 6.3 maintains its virtual pattern for different values of k. Small values of the inertia band parameter reflect a high number of transactions, which implies a large cumulative

Table 6.5: 2003-2008 Genetic Program t-statistics with filter

k	Jan (20-24) 2003	Feb (17-21) 2003	March (10-14) 2003	March 31, 2008
0	1.630	0.810	0.323	0.592
2	1.502	0.701	0.213	0.009
4	1.523	0.722	0.268	-0.412
6	1.107	0.603	0.184	-1.432
8	0.991	0.703	0.013	-0.081
10	0.541	0.403	-0.109	0.313
12	0.269	1.083	0.215	-0.106
14	0.120	0.802	-0.009	-0.678
16	0.018	0.590	0.027	0.790
18	-0.240	0.189	0.079	0.301
20	-0.166	-0.608	-0.148	-0.429

transaction cost that exceeds the profits from trading.

For example, we notice that for the January data we obtain profit until $k = 18$ where not only that we do not obtain profit anymore but we start loosing money, whether for the February data the filter parameter does not affect us that much, in terms of we make less money but we don't loose any. In the March data the inertia band will actually oscillate the periods when we make or loose money (e.g: at k: 4, 6, 8, 14, 20 we loose money). It is important to notice though that the genetic program shows high returns across the different k bands for the 2003 data.

There is a difference for the 2008 data as the signs of average returns during 2008 indicate that we cannot find any clear systematic pattern. Positive returns are only generated across the order book data for k equal to 0, 2, 10, 16 and 18 on March 31, 2008. According to the t-test, some of the returns are statistically significant, but we cannot notice any pattern in the 2008 data, as with the 2003 data.

We now perform the latter test of varying the k parameter for our GP/PSO approach and we notice that even though the results obtained look better, there is still no pattern to be observed in the 2008 data set (see table 6.6):

We see that with the GP/PSO version we have a significant positive return for k equals 12 where with our GP we had a negative return. One other noticeable idea is that our positive returns are higher in the GP/PSO case than in the GP case and when we get negative returns, the loses are smaller in comparison to the GP alone.

Also, we can notice a significant difference between our GP and GP/PSO best fit strategies. We want to mention again that this results are for orientation only, because

Table 6.6: 2003-2008 GP/PSO t-statistics with filter

k	Jan (20-24) 2003	Feb (17-21) 2003	March (10-14) 2003	March 31, 2008
0	1.683	0.832	0.358	0.618
2	1.503	0.887	0.210	0.237
4	1.680	1.003	0.617	-0.139
6	1.461	0.841	0.400	-0.581
8	1.035	0.788	0.146	-0.489
10	0.701	0.587	0.069	0.508
12	0.117	0.912	0.018	0.101
14	0.308	1.119	-0.067	-0.203
16	0.242	0.713	0.109	0.664
18	0.010	0.349	0.021	0.257
20	-0.042	0.184	-0.029	-0.254

of the stochastically nature of the GP we will get different results with every new run of the algorithm, but they will eventually tend after a large number of iterations (usually larger than 700) towards a more stable/similar result all the time. This happens because the program starts identifying patterns if any, as of the case with the 2003 data.

If there are no patterns in data, then our program shown us that reacts accordingly giving results less statistically significant and which seem not to follow any trend or other trading specific rule in most of the cases.

6.1.3 Anatolyev-Gerko tests: Economic significance

For this last test to be possible we have taken into account one more way of predicting the trading signal. This corresponds to the idea of the existence of a *"majority"* rule which is calculated following the next approach: 99 independent runs of the genetic algorithm have been performed to select the best in-sample trading rules. Our "majority" rule produces a "buy" (sell) 1 if the majority of the 99 best fit in-sample trading rules produce a "buy" (sell) signal. The resulted rule is then used to trade out-sample.

We adjust returns to a daily basis again and express their amount in percent. We are taking into account the k threshold value as well as the transaction costs which are reflected in the bid-ask spread. All our results presented in the following two tables indicate significance at the 5% level according to the predictability test *Anatolyev-Gerko.*

It is important to note that the test is performed over 99 runs of the GP or GP/PSO, each of the run containing 1000 iterations. Also the real-returns needed to perform the AG test are found out calculating the actual values from the data, while the predicted-returns are calculated according to what our ”majority” rule tells our algorithm to do.

The table 6.7 shows us the results obtained after performing the AG test on the GP for different values of our threshold k which are varying within the range 0 - 4. Unfortunately we pursued quite late with obtaining results for this test and we were able to get some only for a small range of the inertia band k.

Table 6.7: 2003-2008 GP AG test with filter

k	Jan (20-24) 2003	Feb (17-21) 2003	March (10-14) 2003	March 31, 2008
0	1.90	1.46	1.83	0.31
2	1.72	1.26	0.42	0.12
4	0.85	0.79	0.69	-2.13

The table 6.8 shows us the results obtained after performing the AG test on the GP/PSO for different values of our threshold k which are varying within the range 0 - 4. Unfortunately we pursued quite late with obtaining results for this test and we were able to get some only for a small range of the inertia band k.

Table 6.8: 2003-2008 GP/PSO AG test with filter

k	Jan (20-24) 2003	Feb (17-21) 2003	March (10-14) 2003	March 31, 2008
0	1.73	2.11	0.97	-0.84
2	1.07	1.10	0.28	0.24
4	0.76	0.69	0.54	-1.92

We can observe from the results obtained that returns across the 2008 sample are not systematically positive and they change sign from one information set to another. For the tested range we don't obtain negative results, but this is not excluded to happen for bigger values of our filter band k. Unfortunately, because we have run out of time we were not able to test our algorithm performance to see what will happen. This is due to further work. From the results we have we can just note that overall, the limit order book data shows statistically significant performance according to the Anatolyev-Gerko test.

6.2 Summary

Throughout this chapter we have shown different results obtained after performing the $t - statistics$ test as well as how the individual returns offered by our trading rule strategies change according between our two different approaches: the genetic program and the genetic program with particle swarm optimisation.

The t-statistics test showed us that both our evolutionary approaches can handle transaction costs surprisingly well, giving us what looks like a systematic pattern for the 2003 data, as well as not a pattern in 2008 data. Thus, our algorithm performed as we expected such that it did not show signs of finding patterns where big economical factors come into place, such as the 2008 economical crisis.

To conclude we performed a predictability Anatolyev-Gerko test which showed us again that in 2008 we alternate with our returns, by obtaining positive and negative ones without actually respecting any systematic pattern. This last test is of very much importance in the economical field as it shows predictability level in our forecasted daily returns.

If we now compare our results to the original paper that we based our research on, which is the *Salmon and Kozhan* [20] paper, then we can actually see that our obtained results are not far from the ones they obtained. Even the GP/PSO approach will let us draw the same conclusions, that the gains we got from using evolutionary algorithms to predict market behaviour dropped from 2003 to 2008. The only difference in the results, was that with our GP/PSO approach we obtained better returns that without the swarm intelligence optimization.

We think that evolutionary methods are most of the times quite powerful in identifying market behaviour, and with the help of fast computation tools, such as hardware acceleration, we can obtain even faster and more reliable feedback about the market status at any tested point in time.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Building a specific genetic programming framework for finding trading patterns from scratch has proven to be a challenging and rewarding task. While developing it I had the opportunity to learn many new things and improve my C++ programming skills, while dealing with bad memory access and memory leaks. This project also gave me the chance to build a limit order book market data parser from scratch (without using additional libraries for specific operations) which not only works well, but is also very fast.

Moreover the project also gave me the opportunity to study more about the financial services sector and to become up-to-date with the latest theory in the evolutionary algorithms domain. It gave me the opportunity to think out-of-the-box and design a new program which involves making use of advanced artificial intelligence techniques such as particle swarm optimisation.

I have also applied my statistical skills and learnt about economic issues which might affect financial markets while performing specific econometric tests to check the predictability of our program.

Perhaps the most important aspect of my project is the hardware acceleration aspect which allowed us to show that evolutionary algorithms will become a very powerful prediction tool in the near future, since their execution time speed limitations can be substantially reduced with the help of a high-performance computing tool, such as the FPGAs.

Thus, in this project we managed to design and build a new machine learning technique based on genetic programming and swarm optimisation. We have also managed to show a speedup in the execution time of the program, by accelerating the fitness evaluation part of the genetic program, making use of the FPGAs. Furthermore, we managed to find some interesting profitability and predictability results which show that our program identifies patterns during a range of market conditions and seems

not to find a systematic pattern when unpredictable events such as an economic crisis, occurs. Also, we managed to show that our algorithm is highly robust conforming to our concluded remarks and tests performed.

The application of the combination of genetic programming and swarm intelligence within a hybrid CPU/FPGA environment is as far as we know novel and has not appeared in the computational or financial literatures and as already presented it provides different advantages both from a financial services institution point of view where they can use this project to try to maximise the profits that they might obtain from trading services, as well as for a regulator's point of view where this project can be used a research tool in analysing how trading rules might potentially affect the markets behaviour and seek feedback to what is happening at any point in financial markets.

7.2 Future Work

We think that there are quite a few areas we can improve this research in the future:

- `Real time market conditions adaptation`
  As we have already presented, our program struggles to deal with unexpected economical events that might appear in the financial markets at any point in time. Thus, a significant improvement will be to incorporate *anytime learning* in our program. Anytime learning is a general approach to continuous learning in a changing environment. In this approach the agent's learning module continuously tests new strategies against a simulation model of the task environment, and dynamically updates the knowledge base used by the agent in the basis of the results. Also, the execution module includes a monitor that can dynamically modify the simulation model based on its observations of the external environment; an update to the simulation model causing the learning system to restart learning.
- `Case-Based Initialisation of the Genetic Program`
  Studies have shown that case-based initialisation (the process of solving new problems based on the solutions of similar past problems) of the population results in a significantly improved performance in case of evolutionary algorithms [28]. Thus we think this will be a valuable addition to have which will add not only performance value but will move our approach towards two very researched subjects at the moment: cognitive science and behavioural finance.
- `Support Vector Machine (SVM) classifier on top of GP/PSO`
  Once our evolutionary algorithm identifies any patterns in data, one powerful sort of classifier, such as an SVM one will add contribution to our results because we would then be able to use it to identify any potential relations between the patterns found. The classifier will be able to split our different type of patterns into classes and so we could give more concluding results in terms of any potential financial regimes the market state can be found in.

- `DRAM solution for the FGPA implementation`
  At the moment we are limited on the amount of data we can test our FPGA implementation of the fitness evaluation on, because of the PCIe solution. Once we will wrap up the DRAM implementation we will be able not only to evaluate a large number of individuals (> 1000) on a large number of transactions (order of millions), but we will decrease our data transfer time by approximately 4 times and we will be able to set up multiple pipelines for our application thus decreasing the total computation time by the number of pipelines we've set up on DRAM.
- `Genetic Operators on the FPGA`
  We think that it will be a great addition if we could manage to add all the genetic operators computation (crossover, mutation, etc.) onto the FPGA, thus we will be able to accelerate the whole application. Recent studies shown that it is possible to actually obtain a total 60x percent speedup for a whole genetic algorithm implementation [21]. This approach might get us towards the ideal scenario where we could get feedback about trading rules characteristics behaviour in real time.

Additionally, there are a number of predicted validity tests, such as *Giancomini-White* which we can additionally perform for further testing our approach correctness.

Appendix A

.1 Technical Indicators

1. `Simple Moving Average (SMA)`: it is the simple mean of the previous n price data points of a security.

$$SMA(M,n) = \frac{1}{n}\sum_{i=0}^{n-1} P_{M-i} = SMA(M-1,n) + \frac{P_M - P_{M-n}}{n} \quad (1)$$

   where P_i is the ith data point and M is the current data point.

2. `Exponential Moving Average (EMA)`: it is a weighted mean of the previous n data points where the weighting factor for each older data point decreases exponentially, never reaching zero.

$$EMA(M,n) = \lambda \sum_{i=0}^{n-1} (1-\lambda)^i P_{M-i} = \lambda P_M + (1-\lambda) EMA(M-1,n) \quad (2)$$

   where $\lambda = 2/(n+1)$.

3. `Average Directional Index(ADX)`: measures the strength of the upward or downward movement by comparing the current price with the previous price range and displays the result as a positive (upward) movement line (pDI), a negative (downward) movement line (nDI), and a trend strength line (ADX) between 0 and 100. The ADX is calculated in terms of upward (U) and downward (D) price movements, and the true range (TR). Average True Range (ATR) measures the volatility.

$$ADX = 100 * EMA_n \frac{|pDI - nDI|}{|pDI + nDI|} \quad (3)$$

$$pDI = 100 * \frac{EMA_n(pDM)}{ATR} \quad (4)$$

$$nDI = 100 * \frac{EMA_n(nDM)}{ATR} \tag{5}$$

$$U = High_{n+1} - High_n D = Low_n - Low_{n+1]} \tag{6}$$

$$ATR = EMA_n(TR) \tag{7}$$

$$TR = MAX(High_{n+1}, Close_n) - MIN(Low_{n+1}, Close_n) \tag{8}$$

4. `Moving Average Divergence Convergence (MACD)`: it turns two trend-following indicators, moving averages, into a momentum oscillator by substracting the longer moving average from the shorter moving average. As a result, MACD offers the best of both worlds: trend following and momentum.

$$MACD(f, s) = EMA_f(Close) - EMA_s(Close) \tag{9}$$

$$Signal(S) = EMA_S(MACD) \tag{10}$$

$$Hist = MACD(f, s) - Signal(S); \tag{11}$$

5. `Relative Strength Index (RSI)`: RSI is a momentum oscillator that measures the speed and change of price movements. It is a trading indicator which intended to indicate the current and historical strength or weakness of a market based on the closing prices of completed trading periods. It assumes that prices close higher in strong market periods, and lower in weaker periods and computes this is a ratio of the number of incrementally higher closes to the incrementally lower closes.

$$RSI = 100 - 100 * \frac{1}{1 + RS} \tag{12}$$

$$RS = \frac{EMA_n U}{EMA_n D} \tag{13}$$

6. `Stochastic Oscillator`: Stochastic Oscillator (%K) is a momentum indicator that shows the location of the close relative to the high-low range over a set number of periods. Does not follow the price but the momentum of the price. As a rule, the momentum changes direction before price.

$$\%K = 100 * \frac{C_M - L_{min}}{H_{max} - L_{min}} \tag{14}$$

$$\%D = 3 - daySMAof\%K \tag{15}$$

where C_M represents current closing price, L_{min} and H_{max} represents lowest and highest trading price over the horizon of n trading period.

7. `StochRSI`: StochRSI is an oscialltor that measures the level of RSI relative to its high-low range over a set time period. StochRSI applies the stochastic formula to RSI values, instead of price values.

$$StochRSI = \frac{RSI_M - RSI_{min}}{RSI_{max} - RSI_{min}} \tag{16}$$

where RSI_M represents RSI value calculated using trading price over the horizon of n trading period.

8. `Bollinger Bands`: these are volatility bands placed above and below a moving average. Volatility is based on the standard deviation, which changes a volatility increase and decreases. An indicator derived from Bollinger Bands called %b tells us where we are within the bands. Bandwidth, another indicator derived from Bollinger Bands, may also interest traders. It is the width of the bands expressed as a percent of the moving average.

$$MiddleBand = n - periodSMA \tag{17}$$

$$UpperBand = middleban + n - periodstd.deviationofprice * m \tag{18}$$

$$LowerBand = middleband - n - periodstd.deviationofprice * m \tag{19}$$

$$\%b = \frac{Close - LowrBand}{UpperBand - LowerBand} \tag{20}$$

$$Bandwith = \frac{UpperBand - LowerBand}{MiddleBand} \tag{21}$$

where typical values for m is 2 and n is 20.

9. `Accumulation/Distrbution Index (ADI)` ADI assess the cumulative flow of money into and out of a security. The degree of buying or selling can be determined by the location of the Close, relative to the High and Low for the corresponding period. There is buying pressure when a stock closes in the upper half of a period's range and there is selling pressure when a stock closes in the lower half of the period's trading range.

$$ADI_M = ADI_{M-1} + volume * \frac{(C - L) - (H - C)}{H - L} \tag{22}$$

where H,L and C represent the highest, the lowest and the closing price of current trading period respectively.

10. `Chaikin Oscillator`: is simply the MACD indicator applied to the ADI indicator.

.2 FPGA specifics

The following code presents the way in which we tell the Manager to set the clock frequency for our FPGA design:

```
...
Manager manager = new Manager(params);
manager.setClockFrequency(80);
...
```

Also, we can tell the Manager the number of cost tables to try, the parallelism value as well as the effort of trying to find an optimised low resurces version of our design:

```
...
Manager manager = new Manager(params);
BuildConfig bc = manager.getBuildConfig();
bc.setMPPRCostTableSearchRange(1, 5);
bc.setMPPRParallelism(5);
bc.setBuildEffort(Effort.HIGH);
...
```

List of Figures

2.1 FPGA Basic Architecture Organization 13
2.2 Maxeler toolchain diagram 14
2.3 Diagram of the hardware parts of the MAXCard showing the relationship between the PCIe, DRAM, Host and FPGA 15
2.4 32-bit Floating Point Representation conforming to the IEEE-754 standards 15

3.1 Trading Rule 25
3.2 Trading rule grow initialisation 27
3.3 Trading rule full initialisation 27
3.4 PSO approach to split data set into different classifier classes 34
3.5 PSO initialization 36
3.6 PSO Global Best 36
3.7 Hybrid GA/PSO algorithm original handwritten flow chart 41

4.1 Genetic Program with FPGA acceleration highlight 44
4.2 CPU - FPGA communication channel 45
4.3 Brief overview of implementation design 46
4.4 Graph resulted when debugging our fitness evaluation application 56
4.5 LMem controller architecture 57

References

[1] Fpga architecture for challenge. 12

[2] Tarek Aboueldahab and Mahumod Fakhreldin. Prediction of stock market indices using hybrid genetic algorithm/particle swarm optimization with perturbation term. *International Conference on swarm intelligence*, 2011. 16

[3] D.B. Thomas A.H.T. Tse and W. Luk. Accelerating quadrature methods for option valuation. *Field Programmable Custom Computing Machines, 17th IEEE Symposium on, pages 29-36*, April 2009. 12

[4] Artificial Life by Example... Particle swarm optimisation. 36

[5] Rohit Choudhry and Kumkum Garg. A hybrid machine learning system for stock market forecasting. *International Journal of Social and Human Sciences*, 2008. 18

[6] Swee Leong Christian Dunis, Andrew Harris. Optimising intraday trading models with genetic algorithms. *Neural Network World*, 1999. 18, 23

[7] Markets Commitee. High-frequency trading in the foreign exchange market. September 2011. 16

[8] M. O'Hara D. Easley. Liquidity and valuation in an uncertain world. *Journal of Financial Economics 97, 1-11*, 2010. 30

[9] M.A.H. Dempster and C.M. Jones. A real-time adaptive trading system using genetic programming. *Topics in Quantitive Finance*, 2000. 18, 23

[10] Stephen Satchell Emmanual Acar. Advanced trading rules. June 2002. 16

[11] Oliver Linton et al. Economic impact assessments on mifid ii policy measures related to computer trading in financial markets. *Foresight*, August 2012. 16

[12] Peter Nordin et. al. Genetic programming: An introduction. 2013. 8

[13] Wolfgang Banzhaf et co. Genetic programming, an introduction. 1998. 25, 26

[14] FX for Begineers. The foreign exchange markets for beginners. 6

[15] Ruoen Ren Hua Zhang. High frequency foreign exchange trading strategies based on genetic algorithms. *Networks Security Wireless Communications and Trusted Computing (NSWCTC), 2010 Second Internation Conference, Vol.2*, 24-25 April, 2010. 17

[16] Investopedia. Filter trading strategy. 30

[17] Investopedia. Trade definition. 7

[18] A. Kablan and W.L. Ng. High frequency trading using fuzzy momentum analysis. *World Congress of Engineering, Vol.I*, July 2010. 16

[19] Morgan Kauffman. Reconfigurable computing: The theory and practice of fpga-based computing. 2008. 12

[20] Roman Kozhan and Mark Salmon. The information content of a limit order book: The case of an fx market. 4th August 2011. 21, 30, 78

[21] David B. Thomas Liucheng Guo and Wayne Luk. Customisable architectures for the set covering problem. *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, Edinburgh*, 13-14 June, 2013. 19, 81

[22] Melanie Mitchell. An introduction to genetic algorithms. 1996. 8

[23] M.V.SUBHA and S. Thirupparkadal Nambi. Classification of stock index movement using k-nearest neighbours (k-nn) algorithm. *WSEAS*, 2012. 17

[24] Cristopher J. Neely and Paul A. Weller. Technical analysis in the foreign exchange market. *Federal Reserve Bank of St. Louis - Research Division (working paper series)*, January 2011. 1

[25] D.B. Thomas Qiwei Jin and W. Luk. Exploring reconfigurable architectures for explicit finite difference option pricing models. *Field Programmable Custom Computing Machines, Internation Conference on, pages 73-78*, September 2009. 12

[26] M.S. Vijaya R. Abirami. An incremental learning approach for stock price prediction using support vector regression. 2011. 18

[27] Ramin Rajabioun and Ashkan Rahimi-Kian. A genetic programming based stock price predictor together with mean-variance based sell/buy actions. *World Congress of Engineering, Vol.II*, July 2-4, 2008. 17

[28] Connie Loggia Ramsey and John J. Grefenstette. Case-based initialization of genetic algorithms. 1993. 80

[29] Alina F. Serban. Combining mean reversion and momentum trading strategies. November 2009. 6

[30] Pieter Pietzuch Stephen Wray, Wayne Luk. Exploring algorithmic trading in reconfigurable hardware. *IEEE International Conference*, July 2010. 18

[31] Wall Street Technology. The high-speed arms race on wall street is leading firms to tap high-performance computing. March, 2008. 12

[32] Kyrylo Tkachov. Accelerating unstructered mesh computations using custom streaming architectures. 2012. 13

[33] S.Dey V.Kapoor and A.P.Khurana. Genetic algorithm: An application to technical trading system design. *International Journal of Computer Applications*, December 2011. 16

[34] Mehul N. Vora. Genetic algorithm for trading signal generation. *International Conference onbusiness and Economics Research*, 2011. 17

[35] Wikipedia. Basis point — wikipedia, the free encyclopedia. 2013. 73

[36] Wikipedia. Genetic programming — wikipedia, the free encyclopedia. 2013. 9

[37] Wikipedia. Particle swarm optimization — wikipedia, the free encyclopedia. 2013. 34

[38] Wikipedia. Reverse polish notation — wikipedia, the free encyclopedia. 2013. 33

[39] Stephen Wray. Exploring algorithmic trading in reconfigurable hardware. *Master Thesis*, June 2009. 18

[40] Jean-Pierre Zigrand. Feedback effects and changes in the diversity of trading strategies. *Foresight*, 2011. 16