Imperial College London

MEng Final Year Project

Investigating a Heterogeneous System for Detection of Triangular Arbitrage

Author:
Jan Saganowski

Supervisor:
Prof. Wayne Luk

Second Marker:
Dr. David Thomas

June 21, 2011

Abstract

The aim of the project is to investigate the behaviour of a proposed heterogeneous system, which combines a financial modelling and an on-line component. The application scenario is the detection of triangular arbitrage opportunities in the Foreign Exchange spot market. The approach taken is to separate the problem into arbitrage detection, market data prediction and prediction checking. The constituent parts are implemented on different hardware architectures, showing interaction between the CPU and Field Programmable Gate Arrays (FPGAs).

A clear relationship was found between the amount of predictions and the number of clock cycles required to execute prediction checking in hardware. This characteristic allows for straightforward adjustments to the proposed model. Furthermore, the heterogeneous system provides a considerable reduction in latency ranging from 83% to 90% for numbers of currencies most representative of the current market size.

Acknowledgements

I would like to thank my supervisor, Professor Wayne Luk, for providing the inspiration for this project and his openness about which areas to investigate. Also, for his time and belief in what could be achieved.

I extend my gratitude to Dr. David Thomas not only for his generous involvement, valuable advice and support throughout the course of my work but also for his suggestion to concentrate on the Foreign Exchange market.

My thanks go out to Dr. Tsoi for providing help and much needed guidance regarding the AutoPilot tool. I must also acknowledge CSG, specifically Duncan White, for patience when faced with all my computing resource requests during the project.

Last, but by no means least, I would like to thank my family and friends for their unwavering support and encouragement during my years at Imperial. These last few months have been challenging and I could not have made it without you. So once again, an enormous Thank You!

Contents

Chapter 1

Introduction

When discussing Financial Institutions one cannot understate the importance of Electronic Trading Systems. Indeed, these have considerably grown in importance over the past decade, with key themes such as decreased costs, low-latency and high-frequency. Although with a predominant presence in the equities markets, electronic trading has played an important role in the Foreign Exchange space over the last 15 years. In 1995, 20 to 30 per cent of interbank trading in major currencies was executed electronically, with the figure rising to over 90 per cent in 2001 [18]. As a matter of fact, some of today's trading systems are nearly completely automated, eliminating the need for employing multiple traders for the same task.

There are multiple contributing factors which have led to the growth of Electronic Trading. In equities for instance, decimalization, imposed by the Securities and Exchange Commission in 2001 had a significant impact on pushing market participants to this technology [18]. Moreover, these developments would not have been possible without advances made in the field of computer science.

Companies are increasingly looking towards new technology to give them an edge over their competition [31]. Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) and Cell processors are some key technologies mentioned in this context. Unfortunately, due to the secretive nature of the industry, it is virtually impossible to tell the current status quo.

One might argue that replacing traders with computers might result in a less error prone environment. Unfortunately, as witnessed by trading events on Wall Street on the 6^{th} of May 2010, algorithmic trading can go wrong and with very severe consequences. What was likely a result of a computing glitch caused the S&P index to fall 8.6 per cent within five minutes and the value of Accenture's stock to be traded at the price of \$0.01 [27]. Therefore, as Electronic Trading systems grow in significance, these events emphasise how critical a role the correct and accurate implementation plays in realising the systems potential and in minimising risk.

1.1 Foreign Exchange Market and Arbitrage

So why is the foreign exchange market an appropriate choice for this project? Crucially, as I do not have a substantial background in the field of finance, it is important to choose an area which is relatively easy to comprehend, hence not diverting too much energy from

the main focus of the project. In essence, Foreign Exchange is not difficult to understand. Most readers who have travelled abroad and exchanged money will already be familiar with the basic principles such as exchange rates, transaction costs and the bid-ask spread (further details are presented in Chapter 2).

The events of the financial crisis have shown what can happen when financial institutions take on too much risk. Therefore it is important to investigate possibilities for less risky profit, with arbitrage transactions offering just that. Although there are some who dispute the existence of arbitrage opportunities, I will discuss papers which negate this claim (see Chapter 2). Moreover, a recent article by a partner at the TABB Group provides further proof, estimating that the annual profits in low-latency arbitrage exceed $21 billion (market aggregate) [23].

It is also interesting to note the resilience of the foreign exchange market in the times of crisis, showing a 20 per cent increase in turnover since April 2007, to an astonishing $3.98 trillion (one million million) average daily turnover (as of December 2010) [5].

1.2 Objective

The objective of this Individual Project is to investigate the collaboration between a financial modelling engine and an on-line trading component. Presenting a simplified view, these two elements have very different characteristics and are usually kept as separate systems. Financial modelling is computationally expensive, whereas the issue of low-latency is key when it comes to electronic trading.

In this thesis, I propose a heterogeneous system for the detection of triangular arbitrage, which combines a modelling and on-line component. Due partly to the differences in approach necessary when dealing with these modules, I have decided to develop the heterogeneous model by implementing the constituent parts on different hardware architectures. I primarily investigate the combination of a CPU and Field Programmable Gate Arrays (FPGAs), although I also attempt to integrate Graphics Processing Units (GPUs). A further motivation behind implementing the system components in this manner, is to identify how the heterogeneous nature of the proposed model can map to a collaborative approach between different hardware architectures.

As already mentioned, the application scenario I am concerned with focuses on Foreign Exchange, specifically on detecting triangular arbitrage opportunities in the spot market. However, there is no reason that the proposed system could not be adapted to search for arbitrage in other markets.

Furthermore, as already hinted, the aim of the project is to investigate the behaviour of the heterogeneous system and show how the problem of detecting triangular arbitrage can be split into components across different hardware architectures. Most focus will be given to achieving a reduction in latency for the proposed model, but the scalability and profitability of the solution must also be evaluated.

The project is both challenging and interesting since, to the best of my knowledge, it represents a novel idea in the field (or one that has gone unpublished). In an industry where every millisecond of latency reduction can have a considerable impact on profit, it is worthwhile to examine alternative systems that may be used to accomplish this objective.

The reader may already recognise, that the project sets out to deal with numerous domains. Herein lies part of the complexity as multiple programming languages and development procedures will need to be employed. The further challenge is in striking the right balance between time spent on the in-depth development of the individual components and considerations of the heterogeneous system as a whole.

As the scope of this thesis is quite broad, there will undoubtedly be areas requiring further attention, that I will not be able to investigate. In these circumstances it must be remembered that the main goal is to present the complete heterogeneous system.

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 thesis and how they relate to the report structure:

- **Background:** Firstly, I explain the basic details of the Foreign Exchange market, introducing the concept of triangular arbitrage that will be used throughout the project. Attention is given to hardware methods for accelerating computation with major focus on using Field Programmable Gate Arrays. The chapter concludes with an overview of the most relevant related work, discussing how it is associated with the thesis. (Chapter 2)

- **Heterogeneous Model:** I introduce the particulars of the proposed heterogeneous model, outlining a novel solution for the detection of arbitrage, by means of combining modelling and on-line components using different hardware architectures. After discussing issues surrounding the usage of market data, I present details on the collaboration, outlining **arbitrage detection**, **data prediction** and **prediction checking**. Along with the evaluation, these constituent parts form the further contributions of my thesis and are discussed in separate chapters. Information regarding the arbitrage calculation is presented along with examples, focusing on different approaches to incorporating transaction cost into the model. (Chapter 3)

- **Arbitrage Detection:** I give details as to the first major system component responsible for detecting arbitrage. After developing an initial recursive formulation for the algorithm, I attempt different optimisations in order to improve the performance of the CPU implementation. I make modifications to the core arbitrage detection algorithm and try multi-threading. Moreover, I show how the algorithms could be used in the heterogeneous model and conclude with details of an acceleration attempt implemented on the GPU. (Chapter 4)

- **Market Data Prediction:** In this chapter I develop an algorithm for the prediction of market data, based on the idea of weighting historical data points according to their variability. An evaluation of the algorithm accuracy follows, by means of a square difference method and by comparison with mean, perfect and random predictors. Furthermore, I discuss reasons for not allocating extra time to developing more advanced prediction techniques. (Chapter 5)

- **Prediction Checking:** In this chapter I document the implementation of the prediction checking component on an FPGA. Further background material covering the architecture and tools is provided. I describe an initial algorithm and subsequently consider various performance optimisations, such as loop unrolling and different RAM types. Special attention is given to the memory utilisation on the FPGA device and modifications presented in order to reduce it. Furthermore, minor modifications are made to the design in order to facilitate a straightforward evaluation process. I conclude with remarks concerning my experiences throughout the design process. (Chapter 6)

- **Evaluation:** Continuing from the detailed explanation of the individual system components in the previous chapters, I evaluate these under the condition of varying the number of currencies in the market (i.e. market size). The **prediction checking** component is additionally evaluated by adjusting the number of predictions and also, as the memory utilisation on the FPGA device nears 100%. Having completed these steps, I evaluate the heterogeneous model as a whole, placing emphasis on both synchronous and asynchronous approaches. Furthermore, the proposed system is evaluated from the perspective of profitability, scalability and crucially, latency. (Chapter 7)

- **Conclusions and Future Work:** In the last chapter I conclude with a brief summary of my thesis. I revisit the project objectives and discuss what has been achieved. Finally, I present future work and explain how the project could be expanded and which areas are good candidates for further development. (Chapter 8)

Chapter 2

Background

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

- **Foreign Exchange Market:** The basics of the Foreign Exchange market, covering the concept of trading a currency pair and the relevance of the bid-ask spread.
- **Arbitrage:** A short description of arbitrage and triangular arbitrage, distinguishing these from statistical arbitrage.
- **Hardware Acceleration:** A brief discussion of hardware acceleration techniques including both FPGAs and GPUs, with more emphasis being put on the former.
- **State-of-the-art:** A summary of the most relevant related work.

2.1 Foreign Exchange Market

Firstly, we will consider the foreign exchange market. Most readers, especially those who have had the misfortune of experiencing fluctuating exchange rates, will already be familiar with the key points such as the bid-ask spread and transaction fees.

The foreign exchange market is a worldwide financial market for the trading of currencies. The sheer size of the foreign exchange market is staggering with an average daily turnover of close to $3.98 trillion (as of April 2010) [5], making this the largest financial market of all. To mention some of the unique characteristics, the majority of transactions are executed from London and New York, but the market operates 24 hours a day moving from one financial centre to another (closed on weekends). The other major hubs are Tokyo, Hong Kong and Singapore [3, 20, 35].

The foreign exchange market serves one primary purpose, that is to allow for investment in global markets, as businesses are able to convert their domestic currencies into foreign ones. The major participants in the market are Commercial banks, Central Banks, Foreign exchange brokers, Investment Funds and Corporations, with high street customers requiring foreign exchange services for the purpose of travel or money transfer forming an insignificant part of the market [3].

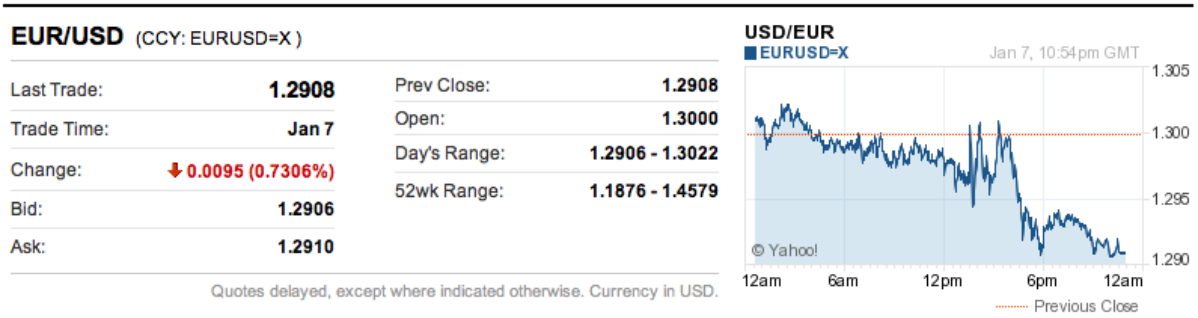

Figure 2.1: The EUR/USD trading pair, showing fluctuations in the exchange rate over one day, the bid and ask prices are also quoted [45].

By its nature, the market is over-the-counter. This means that for the majority of trades, there is no individual, physical market place, rather a collection of participants and market makers.

The most important financial instruments traded are spots, forwards, futures, options and swaps [35, 20]. For the purpose of this individual project we will be focusing on spots, the simplest transactions in the market, an exchange of two currencies at the prevailing market rate (also known as spot rate).

2.1.1 Currency Pair

In the market, currencies are traded against one another, with the value of one currency being defined in terms of the second. All foreign exchange trades involve the simultaneous buying of one currency and selling of another. However, the instrument which is actually traded is called a currency pair. These are defined by concatenating two ISO currency codes, three letters each. One of the most frequently traded currency pairs is EUR/USD, which defines the exchange rate between the Euro and the US Dollar. The first currency in the pair, the Euro in our example, is the base currency while the second is referred to as the quote currency (also known as the counter currency). Currency pairs can be thought of as a single unit that can be bought or sold. When purchasing a currency pair one buys the base currency and sells the quote currency. Figure 2.1 shows fluctuation in the value of the EUR/USD currency pair over a single day; also visible is the bid-ask spread, which we will now cover [15, 35].

2.1.2 Bid-Ask Spread

When dealing with currency pairs, we must also consider the bid-ask spread. A clear illustration of this concept is exchanging currencies on the high street. By definition, the bid-ask spread is the difference between the value at which the broker, bank or market maker is willing to sell the product and the value at which they are willing to buy the product [14], thus ensuring they make a small profit. Revisiting our high street example, when we want to exchange Pounds for Swiss Franc, the broker will charge a higher price when we buy Francs; this is the ask price. Conversely, when we want to sell our Francs to get Pounds, we will be quoted a lower price, referred to as the bid price.

Revisiting the EUR/USD currency pair (see Figure 2.1), the latest exchange rate value is 1.2908, with a bid price of 1.2906 and an ask of 1.2910. The ask price is the price at which we can buy the currency pair from the broker. Therefore, if we buy the EUR/USD currency pair, we will be paying 1.2910 (ask price) US Dollars for each Euro we buy. Conversely, if we sell the pair, we will get 1.2906 (bid price) US Dollars for each Euro we sell [7].

There are rules that need to be adhered to when formulating currency pairs and they are based on the relative priorities of currencies (EUR, GBP, AUD, NZD, USD, CAD, CHF, JPY). This is why the EUR/USD currency pair should never be listed in the reverse order [33].

The bid-ask spread can be seen as a measure of transaction cost and is usually tighter for currencies with high liquidity. The unit for the spread is percentage in point (pip) and represents the smallest unit of a currency, with most currencies being priced to four decimal places. In our EUR/USD example, the bid-ask spread is 4 pips.

The foreign exchange market is also unique in the number of factors that can affect the FX rates. Chief amongst these are economic factors such as government policy, inflation, unemployment, interest rates etc. Further influences include political conditions and market psychology. However, these are not directly related to the project and therefore we will not go into further details [35].

As already noted in the Introduction, the foreign exchange market has been growing during the financial crisis [5], which reinforces its resilience.

2.2 Arbitrage

Having introduced the foreign exchange market, let us concentrate on the subject of arbitrage, also highlighting the differences to statistical arbitrage.

The concept behind arbitrage is relatively simple, it is the practice of making money by exploiting differences in price between different markets. In general terms, if you are able to buy a product and then sell it at a higher price, you have made a profit. However, a vital condition for arbitrage is that these transactions occur simultaneously, otherwise the trader would be exposed to market risk. Therefore, at least in principle, arbitrage offers the possibility of risk free profit [13].

With the development of electronic and algorithmic trading it is becoming increasingly difficult to capitalise on arbitrage opportunities, with these occurring for short periods of time [13], leading issues of latency to become pivotal in the process.

2.2.1 Triangular Arbitrage

A specific type of arbitrage, and one that will be considered in this project is triangular arbitrage. It is the process of converting currencies in order to exploit a state of disequilibrium.

The process involves three exchange rates and takes advantage of differences in cross rates between the currencies. Triangular arbitrage can be thought of as a means of interaction between currencies [32].

Perhaps the easiest way of explaining triangular arbitrage is through the means of a simple diagram (see Figure 2.2). In the example we hold Euro (EUR) and convert this to

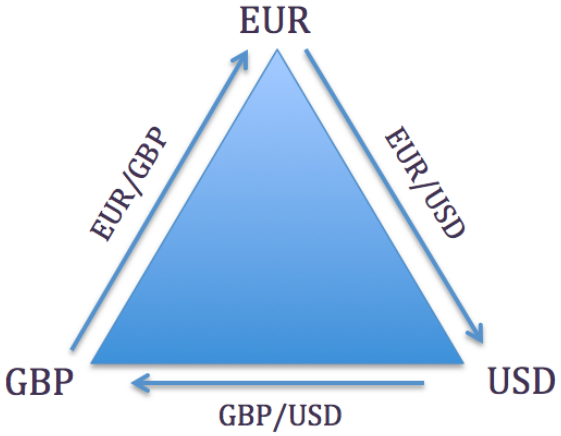

Figure 2.2: An example triangular arbitrage transaction between EUR, USD and GBP.

US Dollars (USD). The Dollars are then exchanged for Pounds (GBP) and finally Pounds are sold for Euros, thus completing the transaction.

If there exists an imbalance between the cross exchange rates, the above mentioned scenario may be profitable and present an **arbitrage opportunity**. A worked example with real values is presented in the following chapter (see Section 3.3).

2.2.2 Statistical Arbitrage

While on the subject of arbitrage, it is worthwhile to note what differentiates statistical arbitrage, which relies heavily on mathematical modelling techniques. Here a trader might decide to execute a transaction based on the forecast from the model, which detects an opportunity for profit. The degree of risk, however, is higher than in the arbitrage scenario discussed previously, as the market is not guaranteed to behave in accordance with the prediction [16].

2.3 Hardware Acceleration

Two hardware acceleration methods of greatest importance to the project are Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). Both represent solutions for accelerating computation.

2.3.1 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 offer a considerable increase in performance, but at the cost of greater development

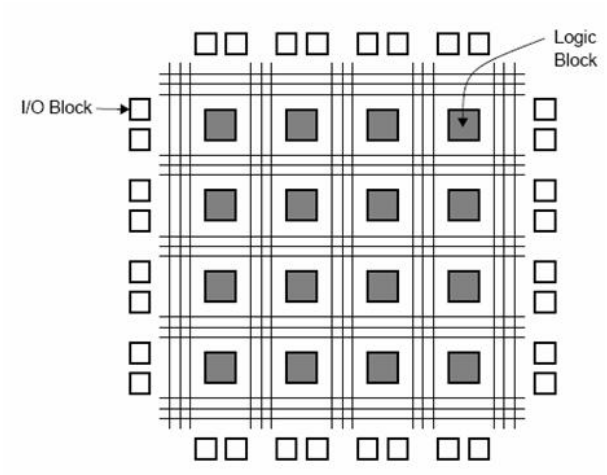

Figure 2.3: General architecture of an FPGA [6].

expense and time. Once a chip has been fabricated you cannot issue a bug fix like with software, the hardware has been pre-configured and will do what it was designed for [10].

Field Programmable Gate Arrays offer the speed advantages of integrated circuits, while also providing the ability to be re-programmed by the customer. Thus they attempt to merge the boundaries between software and hardware.

The advantage to FPGAs 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 (programs) [10]. Techniques exist to simplify programming, including C-to-hardware compilers. AutoPilot is one example and will be discussed in greater detail in Section 6.1.1.

Fundamentally, there are two types of resources on FPGA devices; logic (i.e. arithmetic, functions) and interconnections. Figure 2.3 presents a simplified view of an FPGA, incidentally it is this array like structure where the name is derived from. As a quick side note, the "area dedicated to interconnect greatly dominates the area dedicated to logic" with the ratio being quoted as 90% to 10% respectively [10].

In most general terms, hardware components can be created using truth tables, which in turn can be implemented by look up tables (LUTs) and form the basic building blocks of FPGAs. Logic blocks on an FPGA "contain processing elements for performing simple combinational logic as well as flip flops for implementing sequential logic." [10]

In order to improve the performance of FPGAs we frequently find other components on chip, such as adders, multipliers and memory, thus allowing the devices to implement complex circuits. Hardware Description Languages (HDLs) such as Verilog or VHDL are used during the implementation process.

The two main manufacturers of FPGAs are Altera and Xilinx [34]. The device which has been considered in the implementation work carried out during the project belongs to Xilinx's Virtex-5 offering and is the XC5VLX330T-FF1738. This was chosen even though more recent products exist, like the Virtex-6 and Virtex-7 families [43], as there was a possibility of executing the optimised designs on real hardware at the University.

As running algorithms on the FPGA eliminates the overhead of an operating system, these configurations can offer vast performance increases. Furthermore, it has been

speculated that financial institutions have achieved notable speedup utilising these technologies [31]. FPGAs have also been the topic of research in the field of computational finance [40, 17, 28].

More details as to the implementation process on FPGA devices is provided in Chapter 6, where we consider additional background information (Section 6.1) and the hardware implementation of the **prediction checking** algorithm.

2.3.2 Graphics Processing Units

As many readers up-to-date with recent developments in the computer industry will know, Graphics Processing Units are no longer used exclusively for gaming.

In short, GPU computing allows programmers to take advantage of massively parallel architectures with the aims of increasing program performance. The approach is not without drawbacks, as issues such as data transfer to the GPU, memory alignment and algorithm optimisation need to be considered to take full advantage.

NVidia's CUDA architecture (Compute Unified Device Architecture) allows programmers to execute C and C++ kernels on the company's graphics cards. AMD has a similar offering called FireStream, whereas OpenCL has been introduced with the aim of unifying the two platforms.

2.4 Related Work

Unfortunately, I have not been able to find research relating directly to the topic of combining financial modelling with on-line trading. To the best of my knowledge, the project represents a novel approach in this regard, or at least one that has gone unpublished. Therefore, this section will focus on the topics of hardware acceleration, triangular arbitrage and modelling the foreign exchange market.

An interesting paper regarding triangular arbitrage titled "Triangular Arbitrage in Foreign Exchange Rate Forecasting" [32] deals with the issue of forecasting future exchange rates. The paper presents a model based on Neural Networks. A discussion on the existence of triangular arbitrage in the "real-world" is also presented, as this is sometimes believed to be non-existent. Nonetheless, the paper substantiates the existence of triangular arbitrage opportunities in the foreign exchange market, even when taking into account the transaction costs. Finally, the paper discusses the vast range of transaction cost estimates which have been provided by other researchers. A limitation of this approach is that bid-ask spread has not been considered when examining the data, as it could have provided a more accurate representation of the transaction costs.

A further contribution to the field of market data prediction is made in the paper "On Uncertainty, Market Timing and the Predictability of Tick by Tick Exchange Rates" [24]. The work mentions disappointing results for predicting the foreign exchange market, but shows that this can be done more effectively in the short-term. The heterogeneous system presented in this project relies on the predictability of tick-by-tick market data, but this is also only required for a few ticks into the future (i.e. short term). One limitation of the paper is the market data used, which covers transactions executed in October 1998. It is questionable, whether results obtained from this historical data are applicable to

the current foreign exchange market. However, as the paper was published in 2008 this underscores the problems with access to live (or even more current) market data in an academic scenario.

The study "The Mirage Of Triangular Arbitrage In The Spot Foreign Exchange Market" [4] is very relevant for providing grounding in the model of triangular arbitrage. Critically, the authors investigate triangular arbitrage opportunities in the foreign exchange spot market. The conclusion is that these do exist, although more than half with durations of less than a second, emphasising the importance of co-location and latency reduction. This is an important conclusion, as my thesis relies on the existence of triangular arbitrage opportunities in the market. Moreover, the paper substantiates this claim with the most recent market data (from October 2005) seen in any of the publications mentioned, giving more credibility to the conclusions. Interesting is the magnitude of the opportunities, which is frequently as low as one basis point.

The paper "Triangular arbitrage as an interaction among foreign exchange rates" [2] also provides evidence for the existence of triangular arbitrage opportunities, making the assumption, that orders can always be filled at the bid-ask prices (The previous study [4] puts to question the validity of this assumption). The authors present a model which takes into account the interaction among three exchange rates, as previous approaches do not consider the interaction between multiple prices, with the model successfully describing the fluctuations in exchange rates. This is significant, as having a model of the underlying market is a first step to being able to predict the market data.

The work is followed up by "A microscopic model of triangular arbitrage" [1], building on the previously introduced macroscopic model [2]. This approach is combined with a microscopic model (i.e. two interacting ST models), which replicates the behaviour of dealers in the market and can describe the underlying data better. Furthermore, the relation between the microscopic and macroscopic models is explored through a spring constant.

While on the topic of arbitrage and the discussion on whether it is profitable or if it even exists, it must be noted that the "TABB Group estimates that annual aggregate profits of low-latency arbitrage strategies exceed $21 billion, spread out among the few hundred firms that deploy them" [23], the author being a partner at the TABB Group. This is a vital piece of information, as this Individual Project fundamentally deals with introducing a heterogeneous system, one that searches for arbitrage opportunities using a low-latency approach.

The paper titled "Axel: A Heterogeneous Cluster with FPGAs and GPUs" [30] describes a heterogeneous computer cluster developed at the University. The core problem discussed, N-body simulation, is not the subject of my work. However, the paper is very much related, not only by providing a potential setting to deploy the code developed in the course of my project, but also by showing a successful collaboration between FPGA, GPU and CPU architectures. The best performance acceleration is observed on the FPGA architecture, but this comes at a price of the longest development time, over a month as opposed to approximately one day for the GPU. In order to reduce the FPGA development time for the Individual Project, it seems appropriate to look at C-to-hardware compilers in order to simplify the process.

Although not directly related to the topic of the individual project, there has been a considerable amount of research in the field of accelerating option pricing algorithms,

which gives an idea of the performance that can be achieved on the FPGA architecture. The paper "Exploring Reconfigurable Architectures for Explicit Finite Difference Option Pricing Models" [17] investigates using the GPU and FPGA as a means of accelerating the finite difference method and provides implementation details for the two platforms showing considerable speedup on the FPGA (x12) and GPU (x44). The paper also briefly highlights the trade-off between development time and the performance of the FPGA code. Interestingly, the collaboration between FPGA and GPU is mentioned as possible future work, representing a direction that my project can explore.

The study titled "Debunking the 100x GPU vs. CPU Myth" [19] provides much needed perspective on performance improvements through the utilisation of GPUs, showing that claims of speedups between 10x and 1000x are rarely justified. After performing appropriate optimisations on the CPU architecture, the gap narrows to 2.5x on average. The paper covers multiple algorithms (in the fields of Computational Finance, Linear Algebra and Image Analysis to name a few) and describes in detail techniques used to achieve speedup. These results lead me to consider more carefully the optimisation process carried out on the CPU (the subject of Chapter 4) before declaring immense performance increases on other architectures.

Stephen Wray's Master's Thesis on "Exploring Algorithmic Trading in Reconfigurable Hardware" [39], along with a conference publication of the same title [40] 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. This is particularly important, as the reduction in latency is a key consideration for my project. The issue of run-time re-configuration is also investigated; however, this will not be the subject of my work.

2.5 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 market and the key financial concepts, focusing on currency pairs (i.e. how currencies are traded against one another) and the bid-ask spread. Furthermore, we introduced the subject of arbitrage taking a look at statistical and **triangular arbitrage**, the latter being the main focus of my thesis.

Additionally, we provided an overview of hardware acceleration methods, concentrating mostly on Field Programmable Gate Arrays, noting that additional background information will be discussed before details of the FPGA implementation are given in Chapter 6. We also presented a brief overview of how Graphics Processing Units can be used in the context of increasing the speed of computation.

We concluded with an exploration of the state-of-the-art, i.e. the most relevant related work. As it was challenging to find research directly related to the subject of my thesis, the issues of triangular arbitrage, foreign exchange market modelling and hardware acceleration have been presented.

Chapter 3

Heterogeneous Model

In this chapter we present the proposed heterogeneous arbitrage detection model. The collaboration between system components is outlined as well as key areas for further investigation.

- **Model Overview:** We begin with a general overview of the model, taking a look at the external components, specifically market data and the placement of orders in the market.
- **Model Details:** An in-depth discussion of the model follows, describing the interaction between the financial modelling and on-line trading components. We further subdivide the former, thus defining areas for investigation in subsequent chapters.
- **Arbitrage Calculation:** Here we focus on the calculations necessary for arbitrage detection, presenting alternative ways of incorporating transaction cost into the model.
- **Selected Calculation:** Finally, we present the arbitrage calculation which has been chosen and provide the motivation behind this decision.

3.1 Model Overview

We first consider the general overview of the model being proposed. In simplest terms, as can be seen in Figure 3.1, the model needs to cater for market data, which will be fed into the proposed system. The result of the processing will be a set of orders which could be sent out onto the market (possibly via an Electronic Trading component). The key point to consider is the reduction in latency, the faster the system can respond to market data, the greater the possibility of profit.

3.1.1 Market Data

Delivering market data is an important consideration for the project and we will be dealing with the foreign exchange market, that is FX rates. Market data can be represented by a square array with sizes equal to the number of currencies we are considering ($\mathbf{n}$).

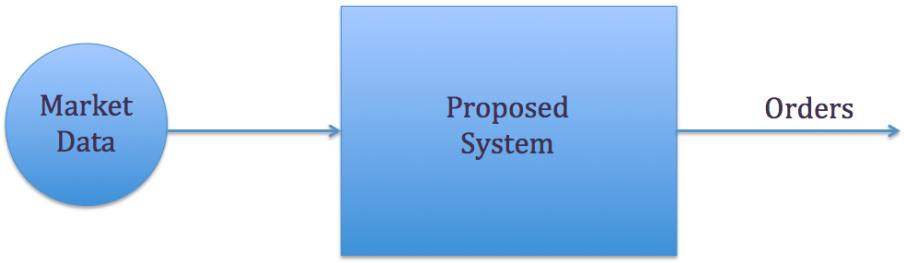

Figure 3.1: Overview of the proposed model looking at the external components.

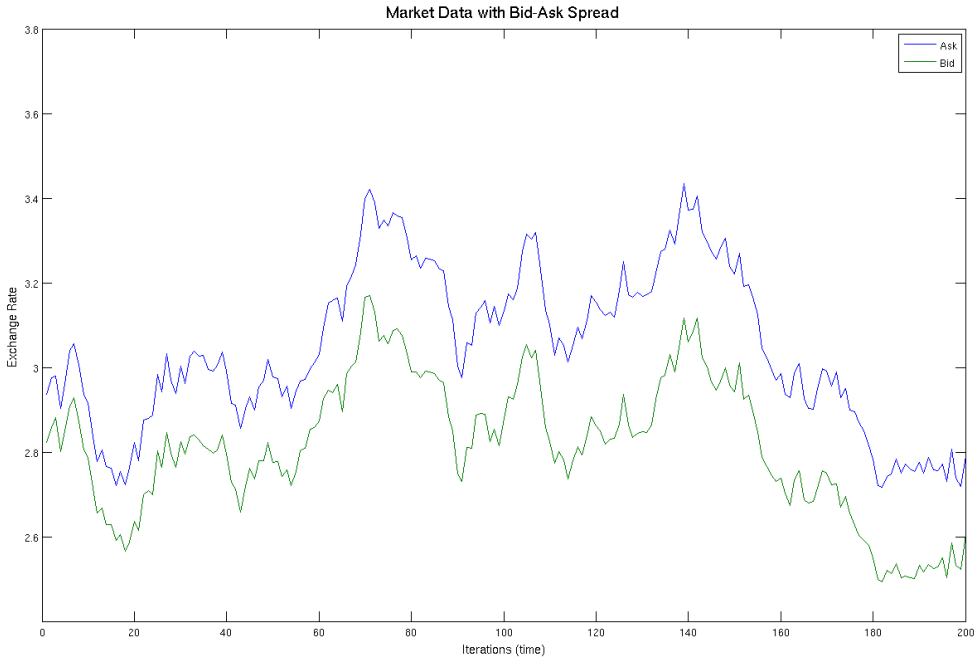

Figure 3.2: Generated Market Data with Bid-Ask spread.

Furthermore, we need one such array for each iteration (or time step) in the market that we wish to simulate (**market_it**, also market tick).

The most realistic solution would be to gather and feed real market data. However, this approach presents a few drawbacks. Namely, getting access to live data would be challenging and would increase the complexity of the proposed model. It is important to keep in mind, that the purpose of this project is not to create a system using the most state-of-the-art components, but rather to investigate the behaviour of the proposed model.

A simpler approach, and incidentally one offering far greater flexibility, is to generate random market data. This component has been provided by Dr. Thomas in the form of Matlab/Octave code. Some modifications were necessary to get the code working reliably with larger numbers of currencies. Namely, the initial matrix for the currencies was not guaranteed to be positive definite which would cause errors. This was fixed by ensuring

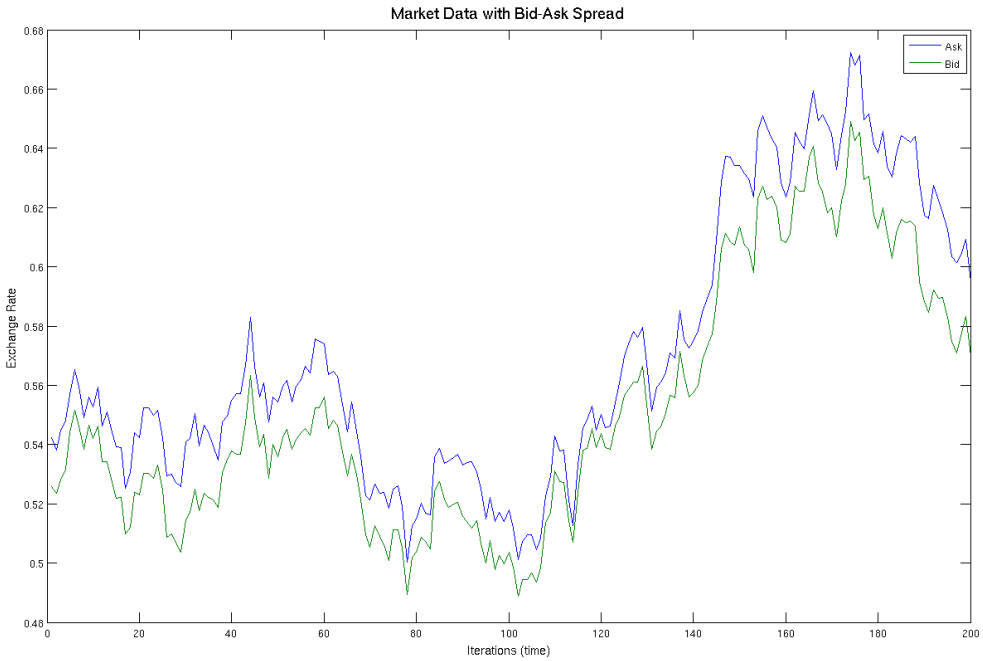

Figure 3.3: Generated Market Data with Bid-Ask spread.

that it is a covariance matrix (using built in Matlab functions).

The results from two different executions of the data generation algorithm are presented in Figure 3.2 and Figure 3.3. Both follow the evolution of a single exchange rate taken from the generated market data. The values are plotted over time, taking into account the bid-ask spread.

Moreover, I created a script to easily generate market data with varying numbers of currencies and iterations. The files generated store one value of the market data per line, with all the iterations being stored consecutively in this file. Although this does not allow for great clarity when reading the files, it was found to be faster than generating tab delimited arrays. Finally, it should be noted that when generating this data, Matlab far outperforms Octave. I understand this to be related to the treatment of for loops by both programs. As a result, Matlab has been chosen to generate the market data files which will be used during the implementation phase.

3.1.2 Placing Orders

Continuing the discussion on the external components, the placement of orders will be outside of the scope of this project. The most important consideration is the reduction of latency for the detection of arbitrage opportunities, and I will not be considering optimisations necessary for the order placing.

To rephrase, the proposed system will be responsible for generating market orders, by providing the currencies (hence necessary information) that would form the trade. However, sending and executing this on the market will not be the subject of further investigation.

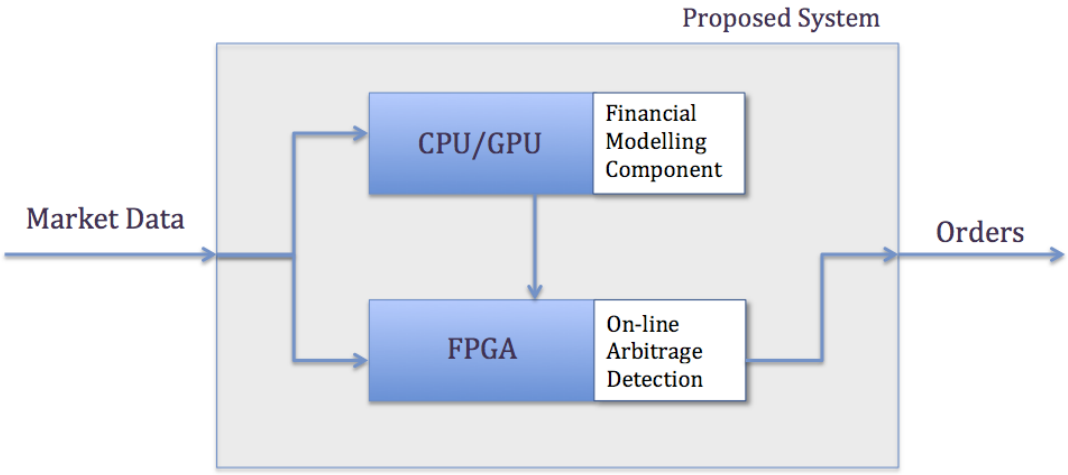

Figure 3.4: A more detailed view of the proposed model showing the collaboration between different hardware architectures.

3.2 Model Details

Having presented the overview for the proposed system and having taken a look at the external components, we can now focus on further details.

As mentioned in the Introduction (Chapter 1), the primary aim of the project is to investigate the cooperation between financial modelling and on-line components. The motivation is to allow the execution of higher complexity tasks (i.e. the modelling component) on a CPU architecture, while allowing for some investigation of the GPU. The comparatively simpler task of on-line arbitrage detection can be accomplished on the FPGA architecture, to take advantage of its low latency characteristics. This scenario is presented in Figure 3.4.

The system will attempt to identify arbitrage opportunities, with the FX rates (i.e. market data) being fed into both system components. The modelling side will be responsible for making predictions on future market data, then checking to see if these would present arbitrage opportunities before sending the values as recommendations to the on-line arbitrage detection.

The on-line arbitrage component (implemented on an FPGA) can then iterate through the predictions and evaluate these against the latest market data it has received. Upon finding a profitable arbitrage opportunity, the system will output the order information.

Following this introduction into the model details, let us now consider the two core components in greater depth.

3.2.1 Modelling Component

As already briefly outlined, the first step to be carried out by the modelling component is to predict future market data. This will be referred to as **data prediction** and is covered in Chapter 5.

The second stage is to search these predicted values for possible arbitrage opportunities. In order to achieve this, we will run an **arbitrage detection** algorithm (covered in

Chapter 4). Finally, the arbitrage opportunities detected by this algorithm can be sent to the on-line component executing on the FPGA.

Both data prediction and arbitrage detection will be implemented on the CPU architecture, although some investigation is also offered into accelerating this computation by using GPUs. It must be remembered in the case of the GPU, that while potentially offering considerable speedup, we also need to take into account the time necessary to transfer the data to and from the device.

Previous work in the field of modelling, has suggested using Neural networks for the prediction of exchange rates [32]. However this was a long term prediction of the exchange rates. In our case, we are more concerned with a few market ticks (iterations) into the future. It must be noted at this stage, that the aim of the project is not to provide the most advanced data prediction component, but to illustrate how one could be used in the proposed model. This will be the subject of further debate in Chapter 5.

3.2.2 On-line Component

The on-line trading component will receive inputs from the financial modelling component and has been implemented on the FPGA. Details of the algorithm, implementation and optimisation process are provided in Chapter 6.

Core to this component is the **prediction checking** algorithm. In essence, this is quite a straightforward algorithm that takes predictions generated by the modelling component and evaluates them against the on-line (latest) market data available. The predictions being transferred between the components are a list of currency triples.

3.3 Arbitrage Calculation

In order to detect arbitrage, we must first be able to calculate it. This section will examine the approach to be taken. I will first present a simplified example with no bid-ask spread, but refine this approach in the subsequent section.

Data provided in Table 3.1 has been gathered using Yahoo Finance [45] and will be used for the purpose of the examples. Values were accessed on 11th January 2011, but are not guaranteed to be quotes from precisely the same point in time. The table contains the exchange rate along with both the bid and ask prices.

Currency Pair	Average	Bid	Ask
EUR/USD	1.2973	1.2971	1.2975
EUR/GBP	0.8303	0.8303	0.8304
EUR/JPY	108.1750	108.1700	108.1800
USD/JPY	83.3750	83.3500	83.4000
GBP/USD	1.5619	1.5618	1.5621
GBP/JPY	130.2604	130.2239	130.2969

Table 3.1: Currency pairs listed with exchange rates, bid and ask prices.

3.3.1 Simple Approach

The simple approach is to use just the quoted exchange rates, without considering the bid-ask spread. We will see later, how this approach can be adapted.

The foreign exchange rates will be represented as follows:

$$F_{EUR/USD} = 1.2973 \tag{3.1}$$

As we are not taking into consideration the difference between the bid and ask prices, we may also write the following (For information why the values of USD/EUR are not usually quoted see Section 2.1.2):

$$F_{USD/EUR} = \frac{1}{F_{EUR/USD}} = \frac{1}{1.2973} \approx 0.7708 \tag{3.2}$$

These values have been calculated for the example data supplied and are presented in Table 3.2. Please note that the row represents the **base currency** and the column should be read as the **quote currency**. The values for currency pairs that were not quoted in Table 3.1 have been filled in by taking the inverse, according to Equation 3.2.

	USD	**EUR**	**GBP**	**JPY**
USD	1.0000	0.7708	0.6402	83.3750
EUR	1.2973	1.0000	0.8303	108.1750
GBP	1.5619	1.2044	1.0000	130.2604
JPY	0.0120	0.0092	0.0077	1.0000

Table 3.2: Currency table with exchange rates calculated from currency pairs, not taking into account bid-ask spread.

I will now show a sample calculation of triangular arbitrage based on converting USD to GBP to EUR and back to USD. Remember that $F_{GBP/USD}$ represents the price of Pounds in terms of US Dollars, which is why the inverse value needs to be taken when calculating the final balance B, assuming an initial input of one.

$$B = \frac{1}{F_{GBP/USD}} \times \frac{1}{F_{EUR/GBP}} \times \frac{1}{F_{USD/EUR}} \approx 1.00035 \tag{3.3}$$

Incidentally, the same calculation could be carried out using the inverse exchange rates:

$$B = F_{USD/GBP} \times F_{GBP/EUR} \times F_{EUR/USD} \approx 1.00035 \tag{3.4}$$

As we can see, the above transactions if placed and filled correctly would result in profit (however small). Of course, this calculation does not take into account transaction costs and as such is unrealistic. In order to improve the approach, a transaction cost T could be added to the equation, thus modelling the cost component. It is important to note, that even when taking the transaction cost into account, it is possible to achieve profit in this scenario.

3.3.2 Bid-Ask Approach

We will now look at a more realistic representation of the market data and take into consideration the bid-ask spread. The $F^a_{EUR/USD}$ and $F^b_{EUR/USD}$ notation will be used to represent ask and bid prices accordingly. The currency pair ask values from Table 3.1 have been used and the ask prices filled in where appropriate. Where the inverse rates were necessary, these were calculated taking the inverse of the bid prices for the corresponding currency pairs (see Equation 3.5). These final calculated values are presented in Table 3.3.

	USD	**EUR**	**GBP**	**JPY**
USD	1.0000	0.7710	0.6403	83.4000
EUR	1.2975	1.0000	0.8304	108.1800
GBP	1.5621	1.2044	1.0000	130.2969
JPY	0.0120	0.0092	0.0077	1.0000

Table 3.3: Currency table with exchange rates, having taken into account bid-ask spread.

We can now re-evaluate the arbitrage possibility as calculated in the previous section. The results show, that having taken into account the bid-ask spread, this would not have presented an arbitrage opportunity:

$$F^a_{USD/EUR} = \frac{1}{F^b_{EUR/USD}} \tag{3.5}$$

$$B = \frac{1}{F^a_{GBP/USD}} \times \frac{1}{F^a_{EUR/GBP}} \times \frac{1}{F^a_{USD/EUR}} \approx 0.99995 \tag{3.6}$$

Just to be clear, this does not mean that other permutations of currencies will not offer arbitrage opportunities under the bid-ask scenario. This can be assessed by checking all the possibilities.

As a side note regarding implementation, storing the exchange rates in an inverse manner to the one just presented would make the calculation slightly less computationally intensive.

3.3.3 Arbitrage Condition

Taking three arbitrary currencies i, j and k (assuming that the cross exchange rates between these exist and can be traded) we can formulate a condition that will evaluate to true whenever there exists an arbitrage opportunity.

$$A_{cond} = \frac{1}{F^a_{j/i}} \times \frac{1}{F^a_{k/j}} \times \frac{1}{F^a_{i/k}} - 1 > 0 \tag{3.7}$$

Which can be re-arranged to:

$$F^a_{j/i} \times F^a_{k/j} \times F^a_{i/k} < 1 \tag{3.8}$$

3.4 Selected Arbitrage Calculation

Ideally, I would have liked to use the Bid-Ask model for computing arbitrage opportunities as it closer represents how trades are carried out in the market. However, having implemented the arbitrage detection algorithm, there were no arbitrage opportunities whatsoever in the data.

As a solution I have reverted to the Simple Model (Section 3.3.1). Therefore the data generation scripts discussed earlier have been adapted to take this into account. The modifications are not discussed in detail but are accomplished by ensuring the generated market data is symmetric around the diagonal. To be precise, the values on opposite sides of the diagonal are multiplicative inverses of each other.

In order to evaluate an arbitrage opportunity, we must now select an **arbitrage threshold** (Equation 3.9). If a value is below this threshold, then we deem it profitable and hence an arbitrage opportunity. This is a direct extension to the Simple Model, as the threshold represents the transaction cost. Moreover, by modifying the threshold value, we can adjust the transaction cost in our market.

$$F^a_{j/i} \times F^a_{k/j} \times F^a_{i/k} < threshold \tag{3.9}$$

It is important to note, that while representing a simplified model, the selected approach should not be seen as a disadvantage. This is because it offers similar, if not greater, flexibility analytically. We are concerned with finding arbitrage opportunities in the generated market data. Therefore all we require is that they represent a small portion of all the possible currency trades. This can easily be achieved by manipulating the threshold value. Furthermore, we do not need to generate new market data (with bid-ask spread) to investigate a situation with a higher number of arbitrage opportunities. Instead, we can simply adjust the threshold value.

As a result, the threshold value has been incorporated into the arbitrage detection algorithms.

3.5 Summary

In this chapter we have given an overview of the proposed heterogeneous model. We began by looking at the external components, covering the topic of random market data generation.

We then presented the model in greater detail focusing on the two main components, financial modelling and on-line trading. In the context of modelling, we introduced **arbitrage detection** and **data prediction** which will be the topic of further investigation. We also noted the on-line component will be implemented on the FPGA and deals with **prediction checking**.

Furthermore, we covered the various approaches to arbitrage calculation, looking at both the Simple and Bid-Ask models. We note that a modification to the Simple model has been selected for implementation, taking into account transaction cost in the form of the **arbitrage threshold**. This was necessary as the generated market data did not present arbitrage opportunities under the Bid-Ask approach.

Chapter 4

Arbitrage Detection

In this chapter we turn our attention towards the implementation of the **arbitrage detection** algorithm. We are primarily concerned with the CPU architecture, although some consideration will also be given towards GPU acceleration.

- **Preliminaries:** We cover initial details such as the hardware and compiler used. Furthermore, we discuss the issue of timing and the run configurations.
- **Arbitrage Detection Algorithm:** We introduce an initial recursive algorithm for the detection of arbitrage. An alternative, optimised formulation is also explained, followed by further performance considerations.
- **Multi-Threading:** We attempt to achieve further speedup by introducing multi-threading. Two and eight thread variants are explored.
- **Heterogeneous Model:** The handling of predictions in the system is considered and three modifications to the arbitrage detection algorithm are proposed.
- **GPU Acceleration:** We provide a concise explanation as to the implementation process on the GPU.

4.1 Preliminaries

Before considering details of the algorithms developed and the implementation and optimisation processes involved, let us first discuss a few preliminaries, most notably the hardware and compiler selected.

4.1.1 Hardware

Very briefly, I present details of the hardware used during the development and benchmarking. A machine running Intel's latest available processor micro-architecture has been selected.

The specifications are outlined in Table 4.1. At this point one should also note that although this is a quad-core processor, "Hyper-threading" technology is enabled. This means that a total of eight cores will be reported to the Operating System.

Processor	Intel Core i7-2600K
Clock Rate	3.40GHz
Cores	4
RAM Size	4096MB

Table 4.1: Hardware used during development.

4.1.2 Compiler

The code developed for the purpose of Arbitrage Detection has been written in C. This is so, that it may be easily ported to other architectures considered throughout the course of the project (NVidia's CUDA uses an extension to the C/C++ language, whereas there are C-to-hardware compilers available for the FPGA architecture).

What remains, is to select a suitable compiler. Based on advice from the "Advanced Computer Architecture (332)" course, I was keen to use the Intel C Compiler [12]. As this tool is offered by the same company that delivers the processor, they clearly possess in-depth knowledge about the architecture and can take advantage of this when optimising code.

Unfortunately, the results when compiling code from this chapter were quite disappointing. The performance was actually worse than that of the GNU C Compiler (GCC) [22], even after experimenting with the compilation parameters.

As a result, the GCC compiler has been used throughout the project. Furthermore, the "O3" optimisation flag was used.

4.1.3 Environment Setup

In order to assess the performance of the different algorithms and optimised code versions, I have executed these against generated market data (see Section 3.1.1). The algorithms were wrapped in code responsible for reading market data from files.

Market data file sizes were kept in the region of approximately 500 to 1024MB, as this allowed them to be stored in the cache, so that once in memory, the slower disk IO would not become a bottleneck when benchmarking.

4.1.4 Run Configurations

The algorithms were executed using multiple iterations of the market data as to better gauge their relative performance by increasing the run time. The parameters used throughout the course of this chapter remain largely the same. We consider the number of currencies (**n**), number of market iterations (**market-it**) and the arbitrage limit (**arb-limit**).

The number of **market iterations** is simply how many market data snapshots the data has. We can also think of this as the number of times the arbitrage detection algorithm will execute throughout the course of a run.

The **arbitrage limit** (also referred to as **arbitrage threshold**) has been selected to limit the number of currency permutations that should be classed as profitable arbitrage opportunities and is kept constant.

Finally, we consider the average of three runs, each using different market data, although randomly generated using the same parameters.

4.1.5 Timing

Although timing the execution of these runs might seem trivial at first, the situation is actually a little more complex. Namely, there are different notions of time on Linux/Unix systems. We distinguish between **real**, **user** and **system** time.

Real time, is defined as "the elapsed time between invocation and termination" [11] and is sometimes referred to as wall-clock time.

User time is "the total amount of time spent executing in user mode" [21]. Occasionally, the operating system will need to be called (requests such as IO, malloc, fork). **System time** is "the total amount of time spent executing in kernel mode" [21], required to execute the requests to the operating system just mentioned.

The resource usage statistics utilised to retrieve timing information from the Operating System are accessible via "getrusage" [21]. These provide details as to the user and system time spent while executing the code.

Unless otherwise noted, the timing measure used through the course of this project is the sum of user and system time, as this should be most representative of the total execution time of the algorithms. Linux is an inherently multi-tasking operating system which employs a scheduler. Therefore, we cannot be sure whether other process have been scheduled during the execution of our code, rendering any wall-clock timing information inaccurate. Hence real time is not a suitable measure.

4.2 Arbitrage Detection Algorithm

The papers I researched, dealing with the topic of triangular arbitrage, did not discuss the specific algorithms used (see Section 2.4). Therefore, for the purpose of this project, I have developed two possible approaches. Firstly, we look at a recursive algorithm. This is not covered in much depth as the alternative (optimised) approach is presented later on, offering considerable advantages.

4.2.1 Recursive Algorithm

A recursive algorithm seemed like the most logical first step towards solving the problem and is also the first approach I tried. However, as we will see it has some disadvantages.

The aim is to find **triangular arbitrage opportunities**, that is to say instances of profitable three-currency trades. If we consider a market with **n** currencies, then we should iterate over the entire possible space (n^3). This is not entirely the case, as this space also contains permutations that are not valid under the triangular arbitrage scenario. Namely, we have the requirement that all three currencies be distinct, therefore our possible sample space is actually $n(n-1)(n-2)$.

The recursive algorithm implemented will first loop through all **n** currencies calling a recursive function to deal with the subsequent iterations. We keep track of the currencies visited along this path to ensure that we only consider valid scenarios. Finally, the algorithm terminates once all possibilities have been exhausted.

During execution, the product of the currencies visited is accumulated (for each permutation) and the values found to be below the arbitrage limit are classed as arbitrage opportunities.

Parameters		**Performance**	
n	100	run 1 [s]	148.96
market-it	5000	run 2 [s]	148.74
arb-limit	0.9	run 3 [s]	150.9
		avg [s]	**149.53**

Table 4.2: Performance of initial recursive implementation.

The performance of this initial design is presented in Table 4.2. The parameters of these runs have already been discussed and the performance figures over three different runs give us an indication of speed.

It can immediately be observed that the performance of this approach is not entirely satisfactory. This will become clearer once we discuss the performance of the revised algorithm.

However, the disadvantage of recursive functions is not limited to their impact on performance. Considering the heterogeneous aspect of this project, they are also harder to map onto the FPGA and GPU architectures. As such it is advisable to investigate an alternative approach, which would enable similar implementations on the other platforms. This leads us to consider the alternative algorithm.

4.2.2 Alternative Algorithm

While dealing with the optimised algorithm let us first clarify the approach based on a slightly simplified problem. We take the initial (first) currency as fixed and look for the possible permutations of the second and third currencies.

Such an approach is presented in Figure 4.1. As mentioned, the initial currency is fixed and we will use the index 0 (zero) to represent it. Following this, we can reasonably look at values 1, 2, 3 etc. for the second currency. The arrows in the figure correspond to paths which represent valid arbitrage scenarios (i.e. the three currencies are unique). The reasoning for the third column is exactly the same. Finally, we need to convert back to the original currency, hence the final column is once again "First". As seen in the figure the transactions involve three different currency exchanges.

One of the requirements of this proposed algorithm is to allocate enough memory to store the entire contents of the third column (referred to as the **results array**). This represents a requirement which is order n^2 with respect to the original number of currencies. Immediately visible in Figure 4.1 is the number of cells that have no incoming or outgoing arrows associated with them. These can be thought of as representing allocated, but unused memory. While this may look substantial for the example provided, where only 4 currencies are visible, the effect becomes less pronounced when we look at larger number of currencies. In this example we access 37.5% of the entries, however, for a case with 100 currencies, this would be 97.02%.

In the interest of avoiding random memory access, we also notice that the way in which exchange rates from the market data need to be referenced is very structured.

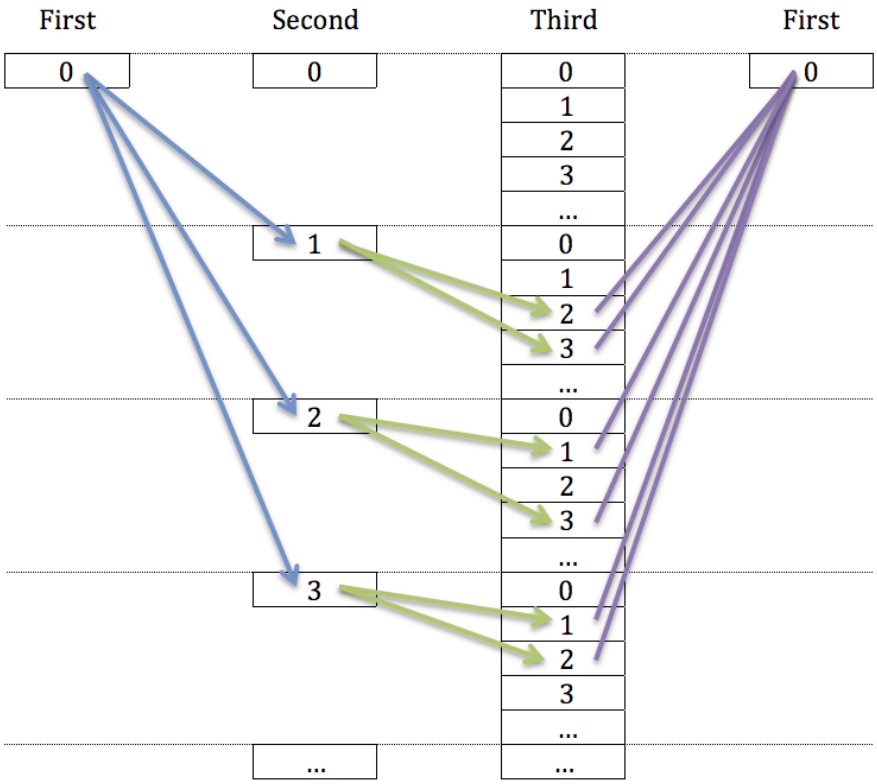

Figure 4.1: Outline of the approach used for the alternative algorithm.

This is mirrored in the storage of exchange rates in the market data, as one row and column represent all the exchange rates for a given currency.

The proposed algorithm works in four steps. The outline of these is presented in Listing 4.1. Please note, exact implementation details of the individual steps have been omitted for brevity.

```
for ( n = 0; n < num_currencies; n++ ) {
    populateFirstExchangeRate();
    populateSecondExchangeRate();
    populateThirdExchangeRate();
    determineArbitrage();
}
```

Listing 4.1: Alternative algorithm - overview

Initially, the algorithm populates the **results array** with exchange rates formed from the first and second currencies. Notice, that we will store the same value for n consecutive entries (this is evident from the "Second" column in Figure 4.1).

The next step involves the "Third" Column from Figure 4.1. This is the least structured memory access as we need to retrieve all possible combinations of two currencies from the memory. Please note that the values retrieved are multiplied by those already stored in the results array.

Now we can once again multiply all the entries in the results array, only this time by the respective exchange rates that will convert back to the initial currency. This is a structured memory access as one of the currencies in the exchange rate is fixed.

Finally, having the products of all permutations stored in memory, the algorithm iterates over this list and determines which ones represent an arbitrage opportunity by comparing with the arbitrage limit (threshold). Only entries with a unique combination of first, second and third currencies are considered in this process.

By splitting the computation up into stages and using a structured approach towards memory access, this algorithm aims to improve on the performance of the initial implementation.

However, one last modification is necessary to allow this algorithm to carry out arbitrage detection. We must iterate over all possible initial currencies (here we assumed the initial currency was known). This is a simple modification facilitated by running the alternative algorithm in a for loop, which iterates over all currencies.

Parameters		**Performance**	
n	100	run 1 [s]	27.50
market-it	5000	run 2 [s]	27.35
arb-limit	0.9	run 3 [s]	27.46
		avg [s]	**27.44**

Table 4.3: Performance of alternative algorithm for Arbitrage Detection.

The performance figures for the algorithm are presented in Table 4.3 and the average value over all the runs presents an approximately 82% decrease in computation time when compared with the recursive algorithm.

4.2.3 Further Optimisation

Following the alternative formulation of the algorithm just described, with the computation split up into stages, I also investigated the performance when combining all these steps into a single computation. That is to say, all the required exchange rates for one entry in the **results array** were accessed simultaneously.

The performance figures for this optimisation are presented in Table 4.4 and represent a further 28% reduction in computation time. This suggests, that the advantages to a more structured memory access pattern are outweighed by reducing the total number of iterations over, and hence memory accesses to, the results array (as implemented by combining the stages).

Parameters		**Performance**	
n	100	run 1 [s]	19.58
market-it	5000	run 2 [s]	19.62
arb-limit	0.9	run 3 [s]	19.65
		avg [s]	**19.62**

Table 4.4: Alternative prediction algorithm - Stages combined.

As a result of the performance figures, we will consider this version of the arbitrage detection algorithm for further optimisation in the remainder of this chapter.

4.3 Multi-threading

Another way of improving the performance of the arbitrage detection algorithm is by introducing multi-threading. Considering the recent predominance of multi-core machines this seems like a logical step.

Furthermore, when referring back to the alternative algorithm (see Listing 4.1), we see that it is not too complicated to introduce multi-threading. As the computation is already split by iterating over possible values for the initial currency, we can have multiple threads concurrently processing different initial currencies.

We take a closer look at two approaches, firstly considering two threads and then moving on to eight.

4.3.1 Two Threads

The dual threaded algorithm has been implemented by incorporating the modifications outlined above. Two threads are launched simultaneously, one for odd, the other for even initial currencies.

Parameters		**Performance**	
n	100	run 1 [s]	34.35
market-it	5000	run 2 [s]	33.37
arb-limit	0.9	run 3 [s]	33.82
		avg [s]	**33.85**

Table 4.5: Performance of Dual-threaded arbitrage detection algorithm.

The results for this implementation are displayed in Table 4.5. This version of the algorithm takes approximately 72% longer to complete than the previously optimised code.

In contrast to previous time measurements, the values taken for this experiment are "real time" (i.e. wall-clock). Although the "user time" was similar to the values seen before, the algorithm was taking noticeably longer to finish (even over multiple executions and regardless of system load). Part of the problem can be attributed to the overhead in creating and joining threads, exhibited in an observed increase in the "system time".

4.3.2 Eight Threads

In order to better understand the effects of multi-threading on this algorithm, I increased the number of threads to eight. This was a maximum reasonable value, as the host machine was running a quad-code (hyper-threaded) processor, reported as eight processors to the Operating System.

Table 4.6 shows the results when executing this algorithm. This is still approximately 30% slower than the single-threaded version. Once again, the "real time" value had to be used for timing, because the sum of "system" and "user" time was actually higher. This is to be expected on a multi-core machine, however the discrepancy observed (approximately 20%) was quite low, suggesting that all cores were not being fully utilised. Incidentally, this is similar behaviour to that observed in the dual-threaded approach presented earlier.

Parameters		Performance	
n	100	run 1 [s]	23.34
market-it	5000	run 2 [s]	28.42
arb-limit	0.9	run 3 [s]	24.68
		avg [s]	**25.48**

Table 4.6: Performance of Optimised Algorithm - Eight threads.

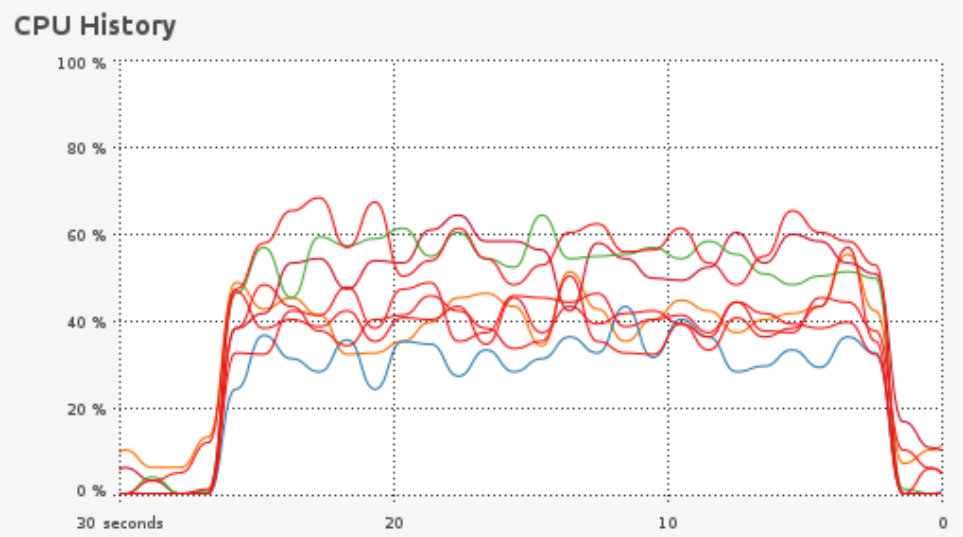

Figure 4.2: Processor utilisation for the multi-threaded optimisation (8 threads).

As a matter of fact, this can be confirmed by looking at the CPU utilisation when executing the code. A cut-down version representing the time during which the algorithm was executing is shown in Figure 4.2. The different coloured lines represent the different cores.

Although the performance has improved when increasing the number of threads, the low CPU utilisation points to a bottleneck, which is likely the memory access to the market data array, where all threads are competing. Given this problem, the results of the multi-threaded optimisations are perhaps not so surprising. Indeed, based on past experience from optimisation exercises during the "Advanced Topics in Software Engineering (475)" course, the best optimisations for an algorithm that was heavily reliant on multiple memory accesses were single-threaded.

As multi-threading the previously optimised algorithm has not provided any performance benefits, we will consider the initial single threaded approach in further discussions.

4.4 Heterogeneous Model Considerations

Before continuing with the implementation details, let us take a moment to reconsider the Heterogeneous Model proposed and the direction that we should take.

The **arbitrage detection** algorithm is required for two main reasons. Firstly, we need a version for the purpose of the evaluation, where we compare the performance of the heterogeneous model with a CPU only implementation. Secondly, after predicting future market data (we will look at details in Chapter 5), we need to run the algorithm to check for arbitrage opportunities. Once these are found, they can be transferred to the **prediction checking** algorithm (the subject of Chapter 6) for processing.

We now take a detailed look at modifications necessary for both these scenarios.

4.4.1 Single-Best Arbitrage Opportunity

Here we present details necessary for the later evaluation of the project. We require a version of the arbitrage detection algorithm that will be used as a reference point and will be executed on the CPU only (also referred to as the "reference CPU implementation")

The approach we take is to find a single best (most profitable) arbitrage opportunity, so that this might be sent to the market. This particular method has been selected, as it is equivalent to the one used in the FPGA implementation of the prediction checking algorithm (see Chapter 6 - further reasons for the choice will be discussed here).

The necessary modifications to the code are very minor. We simply need to keep track of the most profitable arbitrage opportunity and compare any new values against it. As noted before, the single-threaded optimisation of the algorithm has been used.

The performance results for this modification are shown in Table 4.7. We observe a roughly 4% increase in the computation time, which is to be expected given that we have increased the complexity of the code slightly.

4.4.2 Trade Size Modification

We now turn our attention to modifications necessary when processing predicted market data and sending the results to the prediction checking algorithm.

Parameters		Performance	
n	100	run 1 [s]	20.39
market-it	5000	run 2 [s]	20.12
arb-limit	0.9	run 3 [s]	20.96
		avg [s]	**20.49**

Table 4.7: Performance of Single Best Arbitrage Opportunity modification.

For the purpose of this Individual Project, the number of predictions to be sent for processing is limited. Therefore, apart from checking for arbitrage opportunities we need also to prepare the list of predictions. Furthermore, the three currencies involved in the arbitrage opportunity are sufficient to represent the prediction.

One approach is to store a sorted list of length **trade size** representing the best arbitrage opportunities encountered. Once we have iterated over all possible currency permutations, this list can form our predictions.

Code modifications are required to keep track of this sorted list. Upon finding a profitable arbitrage opportunity the entry is inserted into the appropriate place in the sorted predictions array. The precise implementation will traverse the predictions array searching for this position and moving the remaining entries down once it is found. I appreciate, that when it comes to performance, this may not be the most optimal solution. Optimisations could be found by using algorithms such as quick-sort or storing the predictions in a different data structure. Unfortunately, I did not have the time to explore these options.

TRADE_SIZE	16	32	64	128	256	512
t_all [s]	20.39	20.42	20.6	20.7	21.2	22.82
t [ms]	**4.078**	**4.084**	**4.12**	**4.14**	**4.24**	**4.564**

TRADE_SIZE	1024	2048	4096	8192	16384	32768
t_all [s]	27.57	40.23	68.58	124.66	241.41	546.07
t [ms]	**5.514**	**8.046**	**13.716**	**24.932**	**48.282**	**109.214**

Table 4.8: Evaluating the impact of different values of Trade Size.

What remains to be investigated is the implication of changing the **trade size** on the performance of the algorithm. Data in Table 4.8 has been compiled by running the algorithm with the same market data (run 1 configuration to be exact) but adjusting the trade size. This is also shown in Figure 4.3. Please note, that both the x and y-axis use a logarithmic scale. This scale was chosen, as alternatively, the lower trade size points would be displayed on top of each other. Furthermore, execution time is plotted for a single iteration of the algorithm (**t** in Table 4.8), rather than a sum of all the iterations (**t_all** in Table 4.8) as has been used for the performance tables.

What we observe, is that the algorithm is well suited to low trade size values (in the region of 16 - 2048). Once we surpass these values, we pay a considerable penalty for increasing the number of predictions further. The time to complete the computation increases quadratically, although by considering the data, we could also look at fitting a

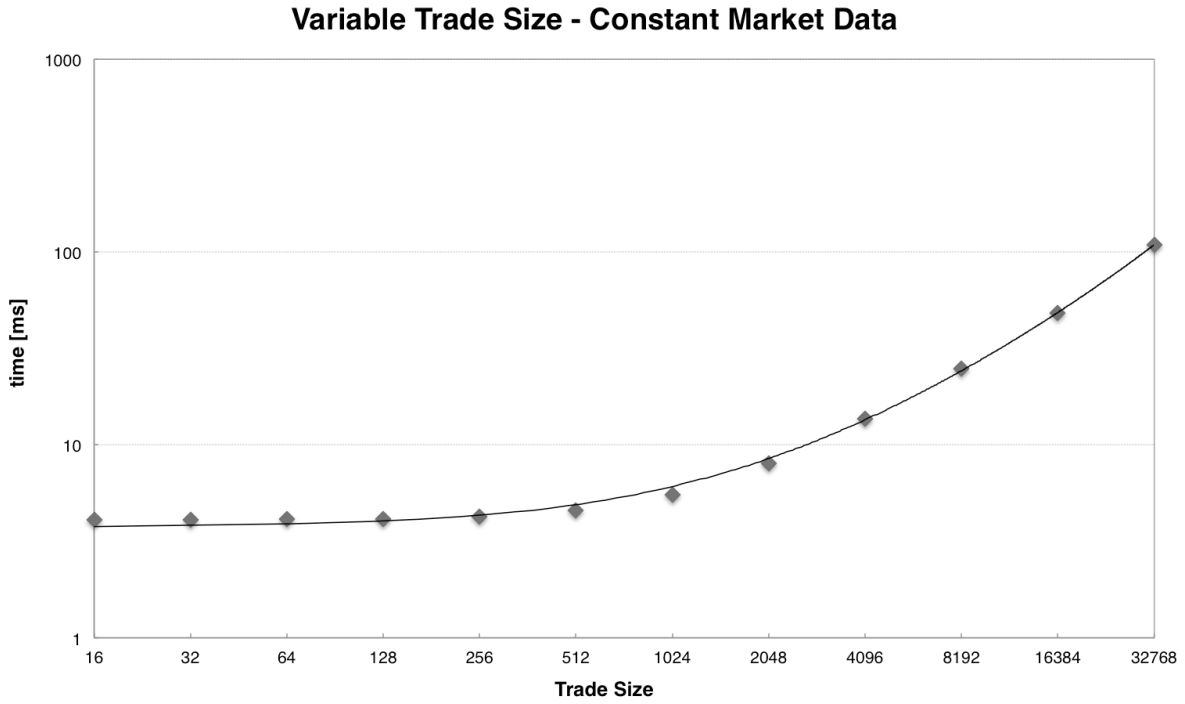

Figure 4.3: Evaluating the impact of different values of Trade Size.

linear trend if we were only to consider data points above a trade size of 1024.

Parameters		**Performance**	
n	100	run 1 [s]	28.46
market-it	5000	run 2 [s]	28.26
arb-limit	0.9	run 3 [s]	26.61
trade-size	1024	**avg [s]**	**27.78**

Table 4.9: Algorithm Performance using a Trade Size of 1024.

Finally, the performance results for a trade size value fixed at 1024 (and in the convention used for this chapter) are presented in Table 4.9. The time necessary for execution is over 40% longer than that of the most optimised algorithm presented before. Although highly dependent on the quality of the prediction algorithm used, 1024 is a relatively small number of predictions. As has been shown in Figure 4.3, the penalties for choosing larger numbers are severe. This is why, we look towards an alternative.

4.4.3 Threshold Modification

The grounding for this alternative approach lies in the realisation that we do not necessary need to keep an ordered list of predictions. Instead, we might simply store all values that present a profitable arbitrage opportunity. In our model, we can do this by comparing against a threshold arbitrage limit.

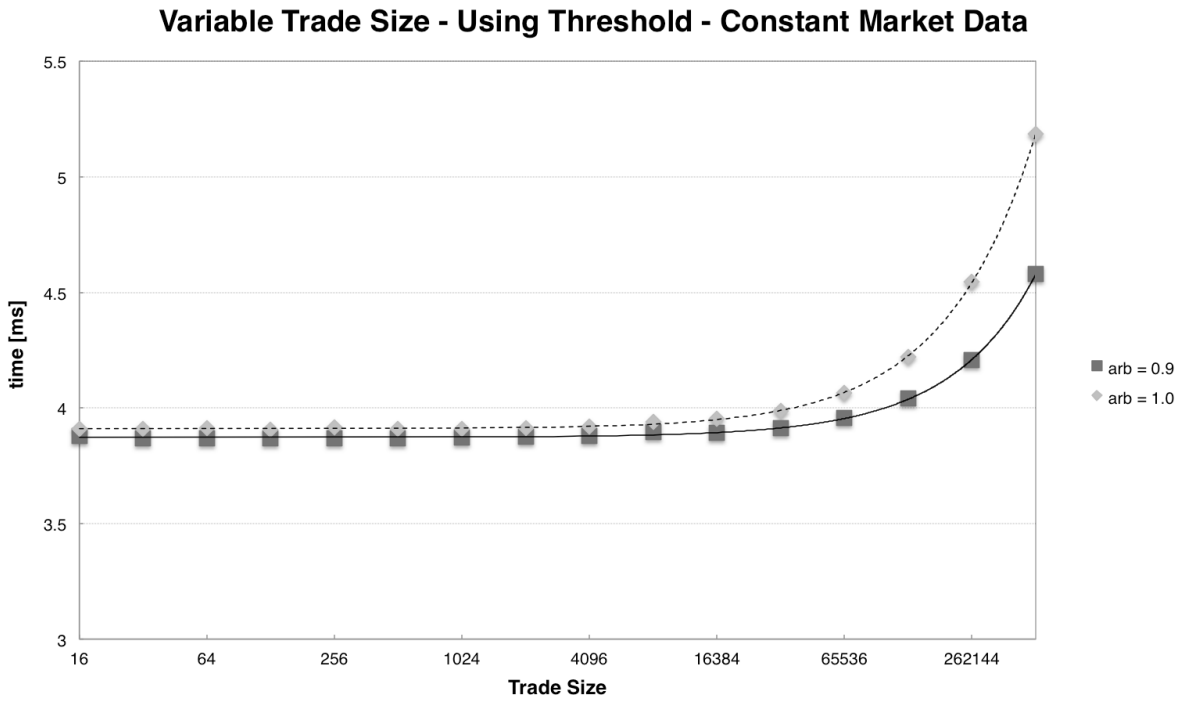

Figure 4.4: Evaluating the impact of different values of Trade Size when using a threshold.

The disadvantage of this approach is that we run the risk of ignoring potentially better opportunities by only storing the first **trade size** values found. This can be offset by increasing the trade size number (i.e. maximum number of predictions stored).

The performance of this approach is examined in Figure 4.4. Please note that results for a single iteration have been plotted, hence the time is presented in milliseconds. Two different values for the arbitrage limit (threshold) have been selected, to show the effect of filling up the prediction array (of length **trade size**). By setting the arbitrage limit to one, we ensure that half of all possible currency permutations are classed as arbitrage opportunities. Although this is a completely unrealistic scenario for the market, it ensures that we fill our predictions array. This will increase the amount of writes to the predictions array, where arbitrage opportunities are stored.

The conclusions that we draw from this data, are that we can use a substantially larger number of predictions than in the sorted case described previously. When increasing the trade size up to a value of 16 384 there are little performance implications. After this, the performance does begin to degrade and is further deteriorated if the market presents a number of arbitrage opportunities capable of filling the entire allocated array. However, as a logarithmic scale has been used for the x-axis, this degradation is by no means severe. If we increase the number of predictions (i.e. trade size) from 16 to 524 288 the time necessary to execute the algorithm increases by little over 30%.

Finally, the performance figures for this modification are presented in Table 4.10. A trade size value of 16 384 has been used, but the required execution time is only slightly (approximately 1%) longer than the previous optimised algorithm (i.e. faster than the sorted case).

Parameters		Performance	
n	100	run 1 [s]	20.10
market-it	5000	run 2 [s]	19.58
arb-limit	0.9	run 3 [s]	19.86
trade-size	16384	**avg [s]**	**19.85**

Table 4.10: Algorithm Performance when using a threshold and Trade Size of 16384.

4.5 GPU Acceleration

An attempt was made at speeding up the **arbitrage detection** computation by executing on Graphics Processing Units. NVidia's CUDA parallel computing architecture was selected.

Unfortunately, after initial experimentation, the results were quite disappointing and no speedup could be achieved. At best, even a simplified arbitrage detection algorithm could only achieve performance similar to that of the CPU implementation.

One of the issues is the necessity of transferring data from the host (i.e. PC, CPU) to the GPU device before computation and afterwards copying the results back. If an algorithm shows considerable speedup when executing on the GPU, this disadvantage can be overcome.

Core to the parallel computing in CUDA is the creation of multiple threads which are executed concurrently on the device. Herein lies the greatest challenge when attempting to implement the arbitrage detection algorithm. Although it is possible to devise a scheme whereby we iterate over all possible currency permutations, the trouble arises when we wish to compile a list of predictions.

A possibility is to store the entire sample space on the GPU and then sort it to find the best predictions. Not only is this computationally expensive, but the memory resources on the GPU are usually more restricted that those on the CPU.

A second option, similar to the approach we have taken on the CPU, is to limit the number of predictions and select only those above a certain threshold (i.e. no sorting). This approach also presents challenges during the implementation. The main problem I faced was trying to keep track of a global (in respect to all the running threads) list of predictions. Although I utilised atomic addition for a counter (as a means of referencing the position in the prediction list), this was not sufficient to allow atomic access to a shared memory and resulted in corrupted results.

This can be solved by using a single thread to accumulate the predictions once all permutations have been computed. Naturally, this process is slow as we are effectively scheduling the execution of a single thread, which is not where the GPU's performance advantages lie.

These issues arise as we cannot know beforehand, how many arbitrage opportunities a certain thread will detect. A workable solution is to give each thread a limit for this. We would effectively be defining a **trade size**, but for each thread individually. This would solve the problem of atomic access to a global predictions array, as every thread would have designated entries associated with it. The disadvantage of this method lies in the definition of the individual trade sizes and would either limit the number of arbitrage opportunities that can be detected, or increase the size of the necessary prediction array

to be transferred back to the host (CPU).

A thorough investigation of a GPU acceleration process would be necessary to provide a detailed view of the performance. As initial results were disappointing and time was limited, more attention has been given towards optimisation on the CPU and FPGA architectures instead.

However, based on the work completed, the conclusions are that given the choice of CPU or GPU, the **arbitrage detection** algorithm should be executed on the former for best **performance**. Moreover, arbitrage detection is just a single component of the heterogeneous model proposed and this does not mean that a GPU could not be deployed effectively when used for **data prediction**.

4.6 Summary

Throughout the course of this chapter we have discussed the implementation of the **arbitrage detection** algorithm, primarily focusing on the CPU architecture.

After giving a brief introduction into the setup, run configurations and timing we discussed algorithms for arbitrage detection. An alternative algorithm was proposed, as the original recursive formulation did not offer the performance required and would not be easily portable to other architectures.

We explored making the algorithms multi-threaded using both two and eight threads, but this did not provide any performance advantages. Therefore the single-threaded code has been used instead.

Moreover, we discussed how predictions would be handled by the arbitrage detection algorithm and provided three different modifications. The **Single-Best** version will later be used for evaluation, whereas the **Trade Size** and **Threshold** variants present two alternative ways of accumulating predictions. The main difference was, that the former represents a sorted approach.

Finally we discussed performance and implementation issues, when attempting to accelerate the code by running it on the GPU. We concluded that the optimised **arbitrage detection** algorithm is best suited to executing on the CPU.

Chapter 5

Market Data Prediction

In this chapter we focus on the issue of **data prediction**. This is the first step necessary when preparing a list of predictions for processing on the FPGA and has been implemented on the CPU.

- **Initial Thoughts:** We present the key assumption for the data prediction component along with a mention of the related work.
- **Prediction Algorithm:** Two alternative predictors are developed based on the idea of weighting. Both a mean and exponential case are explored.
- **Performance Evaluation:** The performance of the proposed algorithms is evaluated against perfect and random predictors.

5.1 Initial Thoughts

To begin with, let us state a key assumption. There will be someone highly specialised in the field of quantitative finance and the foreign exchange market, capable of predicting this data. Indeed, as already discussed in the Background (specifically Section 2.4), approaches concerning the use of Neural Networks [32] have been proposed. Other possibilities include Genetic Algorithms, which have been successfully applied to foreign exchange data to accomplish short-term predictability [24] (i.e. only a few data ticks into the future).

A state-of-the-art modelling component can easily be substituted into the proposed model. The only requirement is that we are able to transfer and store the market data in a consistent fashion. Therefore, it is important to stress that the issue of market data prediction is not central to the Individual Project.

What we must also take into consideration is the market data used throughout the course of the project. As this is generated randomly, algorithms that behave well in this context are not guaranteed to perform well on real foreign exchange values.

Nonetheless, it is important to provide a working prediction algorithm so that we may demonstrate and evaluate the operation of the heterogeneous model as a whole. Every effort has been made to ensure that the performance of the defined predictions is reasonable and will be the subject of further discussion in this chapter.

5.2 Prediction Algorithm

Before looking at details of the prediction algorithms investigated, let us reconsider the entire model and where the prediction component fits in.

The **data prediction** algorithm is essentially a first step to providing arbitrage predictions, which will later be evaluated using the **prediction checking** algorithm. We must take the most current market data available and predict how this is going to change in the next iteration. After this step, the **arbitrage detection** algorithm can be executed (already discussed in Chapter 4) in order to compile the list of arbitrage predictions.

The **data prediction** component has been implemented on the CPU with two alternative approaches being proposed. The same programming language has been used as for the **arbitrage detection** code, that is C. This allows for simpler integration between the components. In principle, however, any implementation (be it software or hardware) could be used as long as it offers sufficient performance and can feed predicted market data into the detection component.

It should come as no surprise that the primary technique we consider when trying to predict future market data is turning to historical values. Indeed a **prediction sample** has been defined which determines the number of past market data snapshots that are considered when making a prediction. We now look at two alternative ways of using this data.

5.2.1 Equally Weighted

The first approach is to equally weight the historical data points. This is essentially the process of finding the mean value of the data points in the prediction sample. It must be noted, that this operation is carried out separately for each entry in the market data array. This is to say, for each point in the market data array, we look at its previous values with the goal of predicting the next. The formula used is shown in Equation 5.1, where N is the **prediction sample**.

$$Mean = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{5.1}$$

The weighted approach is not the most robust, as in principle, we could look for much more complicated patterns and correlations in the data. Sadly, I did not have the time to develop such predictors.

5.2.2 Exponentially Weighted

A more interesting strategy than looking at the mean is to experiment with different weights for the historical data.

Here I investigate an approach which uses the variability of the data in order to determine the weighting factors for the individual data points. The standard deviation has been used as the measure (formula shown in Equation 5.2). Please note, that N represents the **prediction sample**, whereas $\bar{x}$ stands for the mean.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \quad [38] \tag{5.2}$$

The weights of the individual data points will be adjusted based on this variability measure (i.e. standard deviation). The method applied is known as "Exponential smoothing", but regardless of what the name suggests can also be applied to forecasting [37]. This approach represents an exponential weighting of the data points and can be defined recursively:

$$s_1 = x_0, \quad t = 1 \tag{5.3}$$

$$s_t = \alpha x_{t-1} + (1 - \alpha)s_{t-1}, \quad t > 1 \quad [8] \tag{5.4}$$

The first equation (Equation 5.3) represents the base case, i.e. at the first point in time we use the initial value x_0 as our smoothing output. The second equation (Equation 5.4) shows the definition of the recursive step with the most current data point (x_{t-1}) being multiplied by the smoothing factor α.

We can expand this recursive formulation to present this smoothing output using four data points. This is presented in Equation 5.5.

$$s_4 = \alpha x_3 + \alpha(1 - \alpha)x_2 + \alpha(1 - \alpha)^2 x_1 + (1 - \alpha)^3 x_0 \tag{5.5}$$

Furthermore, if we look only at the weights of the points (Equation 5.6), we notice that these sum to one. This means that we need not normalise the smoothing output.

$$\alpha + \alpha(1 - \alpha) + \alpha(1 - \alpha)^2 + (1 - \alpha)^3 = 1 \tag{5.6}$$

Therefore, we need only to consider one input to the exponential smoothing (other than the number of points), the smoothing factor. The effects of taking different values for the smoothing factor are presented in Figure 5.1. Please note that the graph assumes we have five data points (numbered from zero, i.e. $x_0, x_1...$). As we could expect from the recursive formulation, when increasing the smoothing factor, the most recent points are weighted considerably stronger than the previous value. This effect diminishes as the value is decreased.

However, for points with smoothing factor less than 0.5, this formulation starts to favour the initial data point most considerably. The effect is more pronounced for values smaller than 0.3 and also has the unfortunate effect of decreasing and then increasing the weights for the individual points (as can be seen for smoothing factor 0.3 in Figure 5.1). We would like to avoid such behaviour in our predictor.

One final issue we have to consider is how to connect the exponential smoothing with the variability measure. The assumption is made that most recent data points should be favoured and the greater the variability of the data, the more pronounced this effect should be in the computation of the prediction.

Therefore, the implementation takes a normalised number for the standard deviation so that it ranges between zero and one (i.e. valid smoothing factor values). Normalisation is accomplished by keeping track of the maximum standard deviation seen so far. This

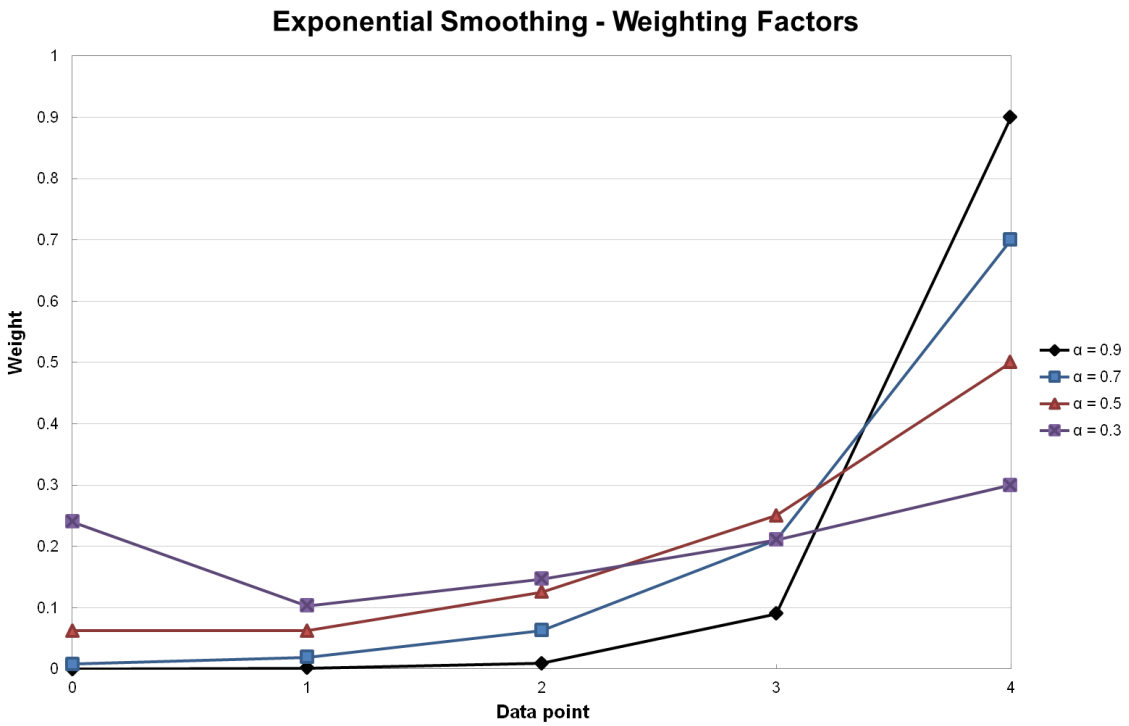

Figure 5.1: Exponential smoothing - Weight factors for individual data points when using different smoothing factors (i.e. values of α).

value is then fed directly into the exponential smoothing in order to calculate a prediction for the future market data point. In order to prevent the weights from decreasing and then increasing (as seen for values smaller than 0.5), but also to avoid emphasising older data points, the minimum smoothing factor has been set at 0.5. Furthermore, I have experimented with different cut-off values in order to prevent extreme weighting of the most recent point and found 0.9 to be a sensible value.

5.2.3 Initial Performance Measure

Having proposed two alternative solutions, we can now proceed to evaluating their relative performance. This is best done by comparing the "real" market data (i.e. most recent iteration of the randomly generated data) with the predicted values. Computationally, this means iterating over all points in the market data array and comparing real with predicted values.

A suitable measure is the square difference and is defined in Equation 5.7, where n represents the number of currencies.

$$D = \sum^{n}\sum^{n}(x_{real} - x_{pred})^2 \tag{5.7}$$

As we want the prediction to be as accurate as possible, we look towards minimising the square difference, D.

In order to test the performance, six different market data files have been selected with different numbers of currencies and varying percentages of arbitrage opportunities

in the data. This run configuration data is given in Table 5.1.

Run	run_1	run_2	run_3	run_4	run_5	run_6
n	64	100	150	256	300	400
market-it	10000	5000	2000	750	500	300
Arb. Opport. [%]	0.408	0.378	0.292	0.270	0.279	0.258

Table 5.1: Run configurations used when comparing the performance of predictors.

Having explained the procedure, we can now run the comparison and results are shown in Table 5.2. Please note, that as these values were quite large (executed over multiple iterations), the results have been normalised so that the mean predictor equals 100%. The results show that the weighted predictor performs much better when evaluated using the Square Difference metric and reduces the difference between the real and predicted data.

Run	run_1	run_2	run_3	run_4	run_5	run_6
Mean [%]	100.00	100.00	100.00	100.00	100.00	100.00
Weighted [%]	46.63	46.70	45.77	46.76	47.12	46.78

Table 5.2: Comparing the two predictors using the Square Difference measure.

5.3 Performance Evaluation

However, the initial performance measure is not the only evaluation method available. A different approach is to compare the performance of both predictors with the behaviour of a **perfect** as well as **random** predictor. The former represents a situation, where all the arbitrage opportunities are predicted correctly. The latter is a case of sending random predictions to the prediction checking algorithm and observing the performance. What we would expect is for the predictors developed to lie within the bounds of these two extreme cases.

First we discuss in greater detail the perfect and random predictors as well as an outline of the implementation necessary. This then leads onto an evaluation of the obtained results.

5.3.1 Perfect Predictor

As already outlined, the perfect predictor represents a rather unrealistic scenario whereby all the arbitrage opportunities in the data are predicted correctly. However, it enables us to establish an upper bound on the performance of any predictor.

Fortunately, the implementation for this kind of predictor is quite straightforward. We require to use the **arbitrage detection** algorithm, albeit with slight modifications. These are necessary, so we do not search for arbitrage opportunities before the predictor has filled its **prediction sample** of historical data and begun processing.

5.3.2 Weighted Predictor

The situation is more complicated when it comes to checking the performance of the weighted (mean and exponential) prediction algorithms proposed.

Once **prediction sample** historical data points have been loaded into the predictor, we can execute the algorithm. After every iteration, the predictions must be checked against the most current (i.e. next) market data and the number of arbitrage opportunities recorded. This process continues until we have consumed all the market data available and enables us to establish how many of the predictions made were actually correct.

This forms the outline of the implementation that was wrapped around the mean and exponentially weighted predictors in order to evaluate them.

5.3.3 Random Predictor

Finally, the random predictor is similar in implementation to the weighted predictors we have just discussed. The crucial difference is that the list of predictions sent for processing is created randomly.

5.3.4 Results

We can now concentrate on the evaluation results obtained when executing all four predictors. At this point it must be noted that the values presented have been normalised, so as to indicate 100% for the perfect predictor. This makes comparison easier and was necessary due to large values obtained from the execution of the algorithms.

Firstly, we consider the number of predictions made by the algorithms (shown in Table 5.3). We must remember that this number will vary as the algorithms will only make a prediction when there is an arbitrage opportunity in the predicted data. This is not exactly true for the random predictor, as it is essentially making guesses and hence does not use the predicted data. The number of predictions for the random case has been increased to allow the algorithm a greater chance of success.

The last column in Table 5.3 shows the average value obtained over the six runs executed. What we observe from the data is that the mean algorithm makes less predictions that the perfect case. In essence, at this point it is already destined to perform worse than "perfect". This is not entirely surprising as calculating the mean makes any peaks in the market data less pronounced, leading to a reduction in the number of arbitrage opportunities (hence predictions).

With the exponentially weighted approach, on the other hand, we are boosting the most recent data points, which leads to there being more arbitrage opportunities in the predicted data (hence more predictions). The number of predictions made is increased above both mean and perfect, immediately suggesting that some of these predictions must be incorrect.

We now turn our attention to the correctness of the predictions. Table 5.4 shows the percentage of predictions made that were correct with the first row once again underscoring the behaviour of the perfect predictor.

The value that is perhaps most noticeable is the performance of the random predictor, with only 0.46% (on average) of the predictions made being correct. This is not surprising

Run	run_1	run_2	run_3	run_4	run_5	run_6	**Average**
Perfect [%]	100.00	100.00	100.00	100.00	100.00	100.00	**100.00**
Mean [%]	95.77	95.78	95.40	95.65	95.59	93.39	**95.26**
Weighted [%]	106.48	105.83	106.45	107.29	104.19	104.85	**105.85**
Random [%]	175.48	164.21	198.59	194.16	181.23	178.78	**182.07**

Table 5.3: The number of predictions made by the tested algorithms. Results in percentages.

considering the predictions were generated randomly.

More interestingly, we observe a high degree of accuracy in the mean predictor, with approximately 86% being correct. However, before making any meaningful comparisons with the exponentially weighted solution, we must remember that we are dealing with different numbers of predictions.

Run	run_1	run_2	run_3	run_4	run_5	run_6	**Average**
Perfect [%]	100.00	100.00	100.00	100.00	100.00	100.00	**100.00**
Mean [%]	85.39	84.99	84.07	87.58	88.82	87.19	**86.34**
Weighted [%]	76.77	77.00	75.70	78.59	81.36	78.31	**77.96**
Random [%]	0.67	0.56	0.37	0.36	0.42	0.40	**0.46**

Table 5.4: Percentage of predictions that were correct for each algorithm.

Taking this into account, we look at the results as a percentage of the performance of the perfect predictor, that is to say we present the percentage of arbitrage opportunities that were correctly detected (given in Table 5.5).

Not much has changed with regards to the random predictor. Having taken into account the number of predictions made, the performance is still very poor. We also note that the performance of both the mean and weighted predictors lies between that for the random and perfect cases, although closer to perfect.

Run	run_1	run_2	run_3	run_4	run_5	run_6	**Average**
Perfect [%]	100.00	100.00	100.00	100.00	100.00	100.00	**100.00**
Mean [%]	81.78	81.40	80.20	83.77	84.91	81.43	**82.25**
Weighted [%]	81.74	81.49	80.59	84.32	84.77	82.10	**82.50**
Random [%]	1.18	0.91	0.74	0.70	0.77	0.71	**0.83**

Table 5.5: Arbitrage opportunities detected as a percentage of perfect predictor.

However, the main comparison is between the mean and weighted predictors. Averaged over six runs, the performance is very similar, with the weighted approach having a slight edge. The first point to make is the difference in how these algorithms behave. The mean approach makes less predictions, but these are more accurate. The exponentially weighted calculation on the other hand increases the number of predictions, but they are less accurate. In the end both solutions lead to similar results.

It would be interesting to experiment with different **prediction sample** sizes and establish whether this has any impact on the relative performance of the algorithms.

Sadly I did not have the time to complete this step.

When we look back at the initial performance measure (square difference) discussed in Section 5.2.3, we recall that the weighted approach appeared to give predictions which better represented the true data. This is interesting, given that both predictors behave very similarly when taking into account the arbitrage opportunities detected.

In other words, even though we are making better predictions about the data, this does not allow us to capture more arbitrage opportunities. This suggests, that in order to improve the prediction algorithms we would need to place more emphasis on detecting scenarios that can lead to arbitrage.

We could consider developing more complicated predictors that search for patterns which may lead to arbitrage opportunities rather than concentrating solely on the most accurate prediction of future market data. As mentioned earlier though, the development of more complicated modelling techniques was not the main focus for this Individual Project.

In conclusion, both the mean and weighted predictors offer good performance when evaluated against a "perfect" implementation, capturing above 80% of the arbitrage opportunities present in the data.

5.4 Summary

In this chapter, we have discussed the prediction of market data as part of the heterogeneous model. We also noted that the development of a state-of-the-art predictor is not the main focus for the project. Nonetheless, with the aim of providing representative results, two predictors were developed, **mean** and **exponentially weighted**. The latter predictor bases its output on the variability of the underlying data which is fed into an **exponential smoothing** calculation.

Furthermore, the suggested predictors were evaluated against the **perfect** and **random** cases and demonstrated over 80% of the perfect predictor accuracy. The values were also comfortably above the performance figures for the random case, which at best managed to perform at 1.18%.

Although the initial performance measure (square difference) favoured the exponentially weighted predictor, after evaluating the percentage of arbitrage opportunities detected, both algorithms exhibited similar results. This suggests that alternative algorithms could be looked at, which not only aim to minimise the square difference, but also search for arbitrage patterns in the market data.

Chapter 6

Prediction Checking

This chapter covers details of the implementation of the **prediction checking** algorithm on the FPGA architecture. We are primarily concerned with the optimisations which have been carried out, but also cover the following:

- **Additional Background:** We present additional details as to the tools used; AutoPilot and Xilinx tools, as well as the implementation process.
- **Prediction Checking Algorithm:** An overview is given of the prediction checking algorithm.
- **Optimisation Process:** Starting with the original algorithm, a memory wrapper is introduced to produce more reliable results. Furthermore, loop unrolling and different RAM types are considered.
- **Compacting Data:** The representation used for the predictions array is compacted in order to reduce the memory utilisation on the device.
- **Result Space Reduction:** The number of arbitrage calculations stored after computation is reduced, with two alternative approaches being compared.
- **Evaluation Preparation:** The design is adapted and prepared for evaluation.
- **Design Remarks:** Concluding remarks are presented, concerning experiences from the design process for the FPGA architecture.

6.1 Additional Background

Before we go on to explain the implementation details on the FPGA and the optimisations which have been made, let us first consider some additional background material for this chapter. We first take a look at the tools which have been used during the optimisation process. These are AutoPilot and the Xilinx tools.

6.1.1 AutoPilot

AutoPilot is a tool that allows the compilation of C and C++ code into hardware and supports execution platforms such as FPGA and ASIC chips [29].

Such a toolset forms a good starting point for users that are new to FPGAs (as was my case), as at least in theory one can provide input in the form of C code and expect to be able to implement the results on an FPGA. As physically running the design on an FPGA circuit was not the main priority of this Individual Project, it seemed that a lot of the work could be done at the higher level.

Unfortunately, matters are slightly more complicated. To allow the user to have an input on the final design, the tool utilises a TCL script where the user can provide details of the FPGA device, the clock period that should be achieved as well as further information regarding the optimisation of the generated output and interfaces between the function and its environment.

It is important to note, that this is a new tool and as such there was only a single floating license available at the University for the vast majority of the project duration, causing some problems. Understandably, there was not much expertise with regards to this tool and the help from Dr. Tsoi and his guide to AutoPilot [29] was invaluable.

The tools provide output in various formats, most notably Verilog, which can then be used by the Xilinx tools for the implementation. AutoPilot will also generate files for Functional Simulation, which I have used to ensure the correctness of the designs during the optimisation process.

The AutoPilot environment is not limited to providing output for Xilinx FPGAs and supports other vendors. Incidentally, the company that developed AutoPilot (AutoESL Design Technologies Inc.) has been acquired by Xilinx during the course of my Individual Project [26]. In my opinion, this move holds great potential for the tools, as a good integration with the Xilinx design environment could substantially speed up the implementation process, but more on this later.

Originally I had planned to rely heavily on the AutoPilot tool for the purpose of obtaining reliable performance results on the FPGA device. Initially, after examining the output reports it seemed that the tool was generating all the necessary information. AutoPilot reports the latency in terms of clock cycles as well as estimates for the clock period, area and resource usage.

As I had no previous hands-on experience with FPGAs, I asked for advice regarding the use of the AutoPilot estimates. Unfortunately, after discussions with Dr. Thomas and Dr. Tsoi it became apparent, that these results could not be relied upon and only a complete FPGA Implementation with the Xilinx tools would be able to provide the accuracy required. This is described in detail in the following section.

6.1.2 Xilinx Design Flow

Since there was a possibility of running the design on Xilinx Virtex-5 devices, I used the Xilinx tools for the process. This section will only give a very brief overview of the stages involved, to illustrate this process to the reader. The two main aspects are synthesis followed by implementation.

A diagram of the Xilinx Design Flow is presented in Figure 6.1 [42]. Starting with the initial design (this would be the output of the AutoPilot tool), we need to run "Design

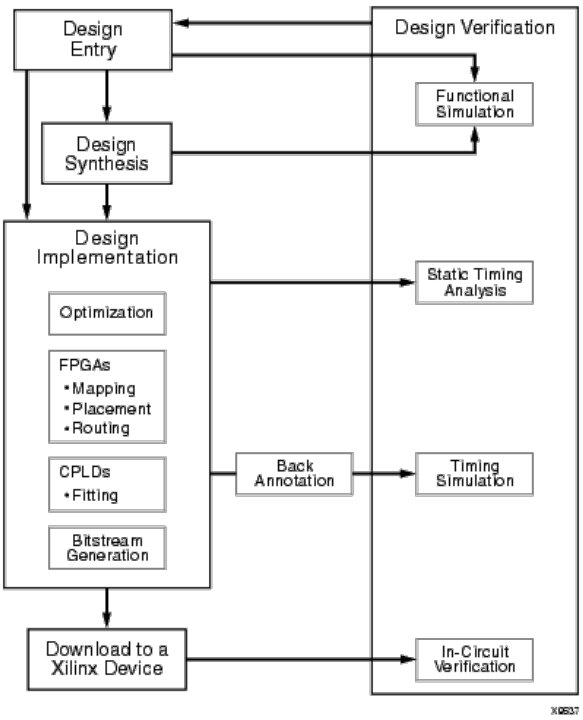

Figure 6.1: Xilinx Design Flow [42]

Synthesis", a process which will synthesise the HDL (Hardware Description Language), Verilog in our case.

During the "Design Implementation" stage, as we are looking at FPGA devices, there are three aspects: Mapping, Placement and Routing. In this process (which is split up amongst different programs in the Xilinx tool-chain) the design is mapped to the specific hardware available on the device, connected and optimised. Only after all these steps are complete, can the "Timing Simulation" be executed, which can be used to derive reliable timing estimates for the implemented design.

Once complete, the design can be represented as a Bitstream and uploaded to the FPGA device.

As mentioned earlier, to achieve reliable results I had to follow the design flow all the way to the Timing Simulation step. Due to the vast amount of configuration options available for the process, the initial (sample) Makefiles provided to me were sometime lengthier than the original C code implementation, adding to the challenge of the FPGA Implementation.

When optimising any hardware design it is important to note, that the software will cut out parts of the design, which it deems as not being used. Therefore it is crucial, that all the parts of the design are connected correctly and are being used to achieve the result of the computation. Otherwise, these will be removed and the synthesis will produce inaccurate results.

6.1.3 Complete FPGA Implementation Run

Unfortunately, I have not been able to carry out as many complete FPGA Implementation runs (up until "Timing Simulation") as I would have ideally liked.

I will not present here all the details of the process, but there were multiple steps involved to begin synthesis of the AutoPilot output. Amongst others, Floating Point modules needed to be generated, Makefiles adapted, output files copied over, project files generated and finally, providing an additional overhead, the memory wrapper files (as discussed in the subsequent sections) modified.

6.2 Prediction Checking Algorithm

We now present an overview of the **prediction checking** algorithm. Having already generated and sent the predictions to the FPGA, the outline of this algorithm is purposefully simple.

The precise code has been omitted for clarity. Suffice to say, that we need to iterate over all the predictions which have been delivered. These are stored as currency triples and contain all the information necessary for evaluating arbitrage. We use this information to look up exchange rates between the first and second, second and third, and finally third and first currencies (recall Figure 4.1).

We can then calculate the product of the three exchange rates to determine whether the triangular arbitrage scenario is profitable. Similarly to the **arbitrage detection** algorithm, this is done by comparing with an **arbitrage threshold** value.

6.3 Optimisation Process

This section builds on the explanation of the prediction checking algorithm. I now present a detailed, step-by-step view of the optimisation process carried out for the FPGA implementation.

To give an overview for readers wishing to skip the precise implementation details, the optimisation stages are as follows:

- **Original algorithm:** The first step is to implement the original algorithm in hardware.
- **Memory wrapper:** Building on this, a memory wrapper is introduced, to give a more accurate representation of the performance of the device.
- **Loop unrolling:** As the device utilization is low and the number of loop iterations high, loop unrolling is used to increase performance.
- **Compacting data:** The prediction data array is compacted to reduce the BRAM utilization on the device.
- **Result space reduction:** The number of prediction results is reduced to save BRAM resources and eliminate unnecessary overheads for selecting trades.

- **Evaluation preparation:** The optimised algorithm is adapted slightly to prepare for the evaluation.

6.3.1 Original Algorithm

The first step to an FPGA implementation is to start with a basic version of the original algorithm.

We will refer to the algorithms implementation as the **core** function. As can be seen in Listing 6.1, this function takes as input market data along with prediction data and produces a set of results. All this data is stored as arrays.

```
void core (
  float market_data[M_SIZE * M_SIZE],    /* market data input */
  int   pred_data[P_SIZE * P_DEGREE],    /* prediction data input */
  float results[P_SIZE]                  /* results output */
)
```

Listing 6.1: Original Algorithm - function parameters.

The prediction array has dimensions which are `P_SIZE` (number of predictions) by `P_DEGREE` (degree of arbitrage, in our case three). This algorithm will simply iterate through the entire array using indexes for each currency as stored in the prediction data and save the result of the computation in the output array. Unsurprisingly, the code representing this algorithm is presented in Listing 6.2. Please note that this is an overly simplified representation.

```
/* cycle through the prediction data */
for ( i = 0; i < P_SIZE; i++ ) {
  /* calculate arbitrage and store in results */
  results[i] = calculate_arbitrage(i);
}
```

Listing 6.2: Original Algorithm - main loop.

The code used during the AutoPilot compilation is more complex than the excerpt presented and has been left out for the sake of clarity. Specifically, the way of indexing currencies in the market data array as well as the prediction data array have been adapted to use single-dimensional arrays. This was necessary, due to limitations with the AutoPilot tool, which would not automatically convert representations from two-dimensional arrays.

Before discussing the results of the AutoPilot compilation and FPGA implementation, we must first present a set of parameters that will be kept constant throughout most of the optimisation process. These are **n**, the number of currencies in the market data (`M_SIZE` in Listing 6.1) and **it**, the number of predictions in the array (`P_SIZE` in Listing 6.1). We will start the investigation using 100 currencies and 10 000 predictions.

The results for this first FPGA implementation are presented in Table 6.1. The parameter values have already been explained above.

With regards to the performance figures, the number of cycles is taken from the AutoPilot report as this will remain constant after the FPGA implementation. The clock period and frequencies are taken from the Xilinx tools. The timing report was generated after the "Place and Route" step in the design flow. Finally, **t** (in milliseconds) is the

Parameters		Performance	
n	100	period [ns]	4.854
it	10000	freq [MHz]	206.02
		# cycles	220001
		t [ms]	**1.068**

Table 6.1: Initial algorithm in hardware - no memory.

time required to process all the prediction results and is calculated by multiplying the clock period by the number of cycles.

6.3.2 Memory Wrapper

I sought advice as to the importance of introducing a memory subsystem and the likely effects that this would have on the design. A discussion with Dr. Thomas confirmed the possibility of significant performance impacts. Therefore, I have included a memory wrapper in the FPGA implementation stage.

One of the shortcomings when working with the AutoPilot tool is that it does not synthesise the required memory resources for the user. If we were to define a local array used only by the function, then the AutoPilot tool would also generate the necessary memories. Unfortunately, this is not the case with the prediction checking algorithm, as we are accessing three arrays (market data, predictions and results) which are all external (as seen in Listing 6.1). In this case, the user must provide an interface to the function by specifying which types of memory the function is interfacing with (i.e. SPRAM, DPRAM etc.). Furthermore, the implementation of these resources must also be specified.

To solve this problem, I took the core function being generated by AutoPilot and wrapped it in a Verilog design containing the required memory resources. This was done by adapting Dr. Tsoi's examples of FPGA synthesis using AutoPilot, as I had no exposure to Verilog prior to my Individual Project. The exact configuration of the RAM used is "Single-Port RAM With Enable", as specified by the Xilinx XST documentation [44].

After ensuring that the memory wrapper contained resources for the three data arrays from the core function and these were correctly interfacing with each other, it was possible to go ahead with the synthesis. To avoid any problems with the optimisation step removing parts of the design I ensured that the market data and prediction arrays could be written to by using global input pins. Furthermore, I implemented a switch which controlled whether the FPGA memory resources were being accessed by the global input or the core function. Finally, I connected the output of the core function to a global output.

The results of the synthesis, when taking the memory implementation into account, are presented in Table 6.2 and underline the importance of using this approach. The final time to calculate the predictions is considerably longer (by approximately 68%) than the original implementation.

Parameters		Performance	
n	100	period [ns]	8.159
it	10000	freq [MHz]	122.56
		# cycles	220001
		t [ms]	**1.795**

Table 6.2: Hardware implementation with memory wrapper.

6.3.3 Loop Unrolling - SPRAM

Following the FPGA implementation of the complete design as described in the previous sub-section, we can now discuss the resource utilisation on the device. This data is presented in Table 6.3, specifically in the first row, Unroll factor 1. As seen, the device utilisation at least in terms of Flip Flops, Look-up Tables and DSP48 slices is very low.

The analysis may not be specifically relevant when considering the Block RAM utilisation, as a larger market data size or greater prediction array would be able to fill the memory (indeed, this will be the subject of the evaluation in Chapter 7).

Unroll	Cycles [#]	DSP48 [%]	FF [%]	LUT [%]
1	220001	4	1	1
2	120001	2	1	1
4	72501	5	1	1
5	64001	5	1	1
8	53751	3	1	1
10	51001	3	1	1
16	46876	6	1	1
20	45501	4	1	1
25	44801	5	3	1
40	42751	5	4	3
50	42401	7	7	4
100	41201	209	18	7

Table 6.3: Loop unrolling using SPRAM.

The concept of "loop unrolling", perhaps more familiar to the field of optimising compilers, is very applicable when it comes to compiling code for the FPGA. We can think of a for loop as multiple executions of a single functional unit.

Perhaps it is best to illustrate this with a short example. Let us suppose we want to add two arrays of length 100. We can solve this problem with a for loop, where each iteration uses a single adder to sum two array elements and iterates over all elements. If we ignore for the time being the necessary control logic, in hardware this could be implemented using the same adder 100 times sequentially. Clearly, this is not the most optimal solution when we consider performance. Looking at the other extreme, we could use 100 adders and would then only require one iteration to compute the solution.

The former approach is actually how AutoPilot treats for loops by default. Specifically, resources required to implement the for loop in hardware are allocated to a single iteration of the loop and are then re-used during each iteration. The user can provide input to the

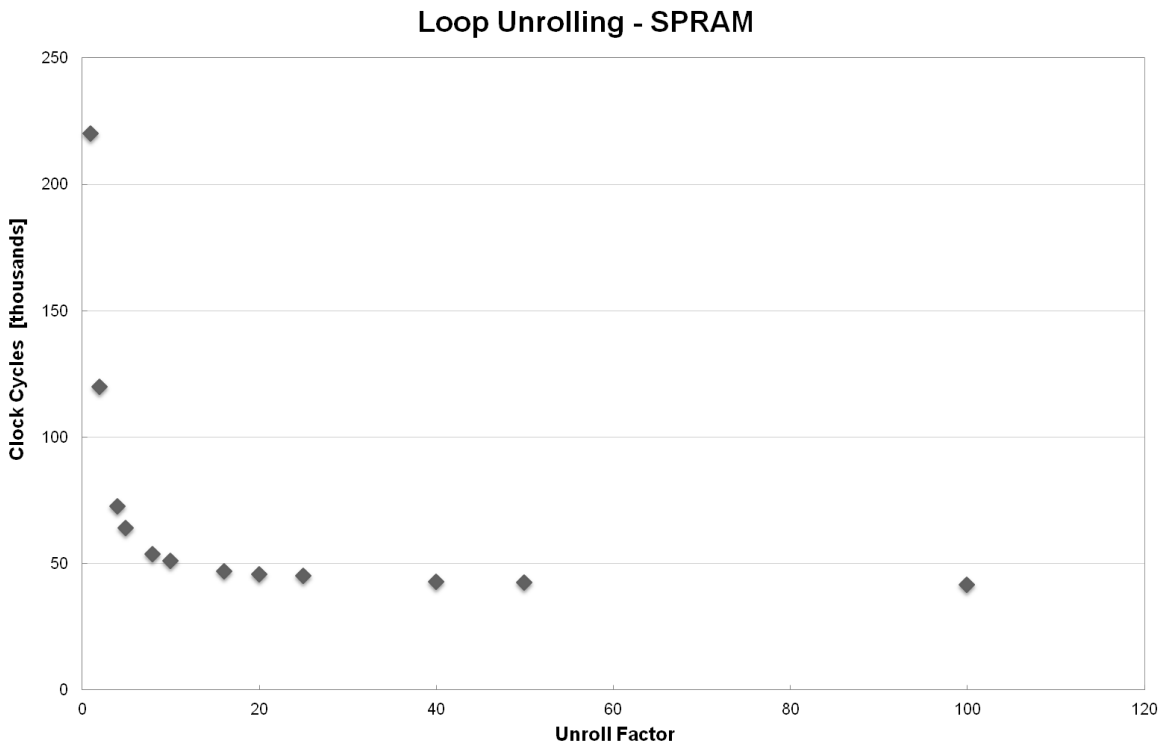

Figure 6.2: Performance of the prediction checking algorithm when using loop unrolling with Single Port RAM.

compilation stage about how many times the loop should be "unrolled". We therefore have the option of using more resources on the device, with the aim of reducing the number of clock cycles for the complete computation.

This approach is therefore particularly well suited to the algorithm being considered as we have already established that the resource utilisation of the device is quite low and the primary objective is to reduce the time required for the computation.

I ran AutoPilot compilations for various degrees of loop unrolling. The results are presented in Table 6.3 and Figure 6.2 (no suitable trend-line could be found). Please note that Single Port RAM has been used in the process. Moreover, apart from unrolling the loop 100 times, all the other designs are small enough to fit onto the device.

Initially, as the loop unroll factor is increased, we see a sharp decrease in the number of clock cycles. This is because more predictions can be checked simultaneously by using the extra resources. Unfortunately, we also see that the returns to unrolling quickly diminish and after a loop unroll factor of 16, we get very little performance improvement from extra unrolling.

Although this approach produces good results, I had originally hoped for even better performance gains. Indeed, it seems that the computation is becoming memory bound and the additional resources are going to waste. This hypothesis will be explored in greater depth in the next section.

For now, let us turn our attention to finding the optimal loop unroll factor for the current scenario. We would like to choose a value that provides a good trade-off between the number of clock cycles and the resources required. It is important to note, that

the more complicated designs will likely result in increased clock periods in the finalised FPGA implementation.

Looking at the data in Table 6.3, a loop unroll factor of 16 seems like a logical choice, as any further increases in complexity see very minor returns with regards to performance. It should also be noted, that the loop unroll factors have been chosen, so that they are divisors of **it** (i.e. the number of predictions). This ensures that no extra logic needs to be generated during the AutoPilot compilation phase, to check whether the algorithm should terminate (in the middle of the unrolled for loop for instance).

Parameters		**Performance**	
n	100	period [ns]	9.408
it	10000	freq [MHz]	106.29
		# cycles	46876
		t [ms]	**0.441**

Table 6.4: FPGA implementation results for a loop unrolling factor of 16.

The results of the FPGA implementation executed for a loop unrolling factor of 16 are shown in Table 6.4. The number of clock cycles has been reduced by approximately 79% when compared with the original implementation.

This is offset slightly by the increase in the clock period (approx. 15%), which is to be expected as we have increased the complexity of the design. However, the time to complete the computation of all the prediction results has been reduced significantly to 0.441ms (by approx. 75%).

6.3.4 Loop Unrolling - DPRAM

In the previous section we dealt only with Single Port RAM and suspected that the computation is memory bound. To check this hypothesis it is worth considering loop unrolling in the context of a different memory system.

Dual Port RAM (or DPRAM in short) should improve the performance of the algorithm further, as twice as many market data values and predictions should be readable during the same clock cycle. Moreover, the BRAM resources on the FPGA device support Dual Port access [44].

To achieve this, the AutoPilot configuration scripts have been adapted to generate an interface for Dual Port RAM on all three data arrays of the function. The results of these compilations are shown in Table 6.5.

By comparing the number of cycles with the implementation from the previous section using Singe Port RAM (see Table 6.3), we can see that the number of clock cycles has been further reduced. If we compare the number of cycles when unrolling the loop 40 times (42751 for SPRAM and 27751 for DPRAM), we see an approximately 35% decrease.

An important observation is that this value is close, but not equal to the 50% improvement suggested by using Dual Port RAM. One possible explanation for this behaviour is that the prediction data array contains three values to be read per iteration, thus the first two can be read simultaneously, while the third one causes the memory to act as in the SPRAM case. This problem warrants further investigation and would be dependent

Unroll	Cycles [#]	DSP48 [%]	FF [%]	LUT [%]
1	210001	3	1	1
2	110001	6	1	1
4	60001	6	1	1
8	38751	8	1	1
16	31876	8	1	1
40	27751	14	6	3

Table 6.5: Loop unrolling using DPRAM.

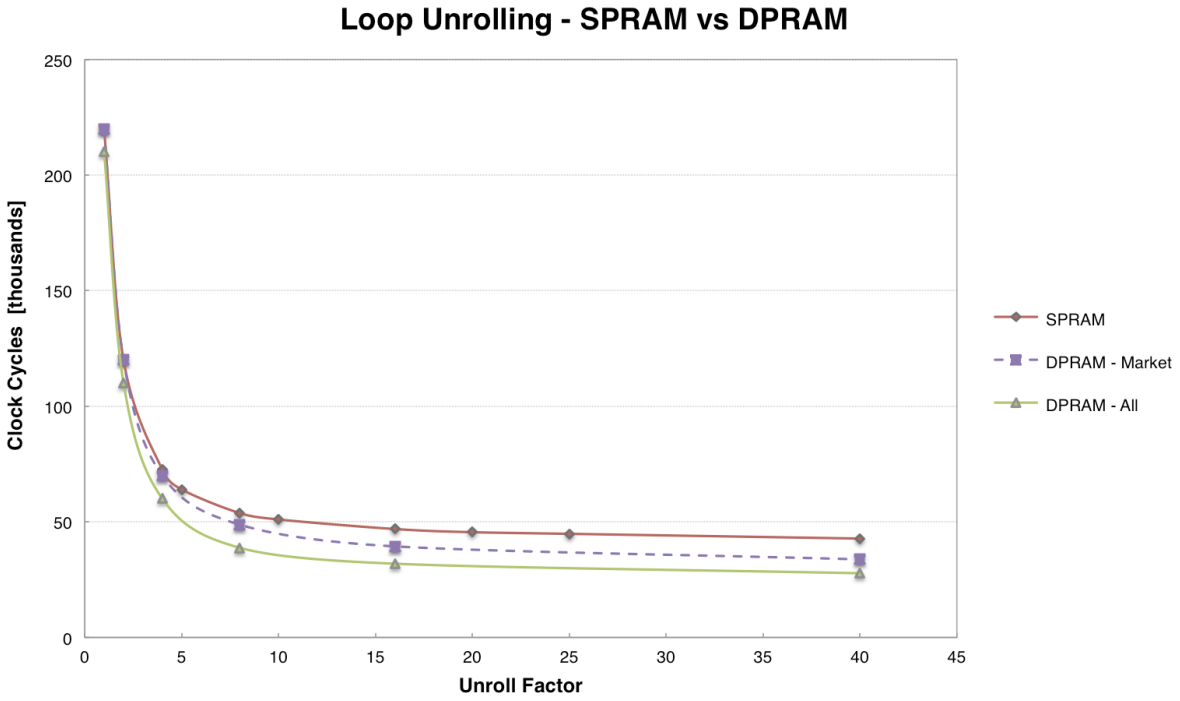

Figure 6.3: Comparison of the prediction checking algorithm when using Single Port vs. Dual Port RAM.

on the exact implementation of the unrolled loop. Unfortunately, as this is a machine generated output I will focus on alternative ways of optimising the code.

For an easy method of comparing the performance of the SPRAM and DPRAM implementations please refer to Figure 6.3. The data series that have been discussed above are SPRAM and DPRAM-All and are presented using curved lines to aid comparison.

Please note, that a third data series (DPRAM - Market) has also been introduced. It represents performance of the system when using DPRAM only for the market data array and predictably, sits between the two scenarios previously discussed.

Unfortunately, I have not had the time to run a complete FPGA implementation of the DPRAM scenario, as this would involve heavy modifications to the memory wrapper Verilog code and extra care to ensure the correct communication with the core function. Therefore we will explore further optimisation possibilities following the SPRAM implementation.

In principle, however, there is no reason that the frequency of a Dual Port RAM design should be lower to that of the SPRAM case. Therefore during FPGA implementation, we would expect the timing results to be similar to the SPRAM case, hence allowing for a further 25 - 35% speedup in the total computation time. These figures are based on the reduction in DPRAM clock cycles when compared with SPRAM for unroll factors of 8 and above.

6.4 Compacting Data

Up until now, we have not directly considered memory utilisation on the device. We are concerned with the Xilinx FPGA device, XC5VLX330T-FF1738, to be exact.

```
Number of DSP48Es                          13 out of 192      6%
Number of RAMB36_EXPs                      64 out of 324     19%

Number of Slices                         1553 out of 51840    2%
Number of Slice Registers                3613 out of 207360   1%
   Number used as Flip Flops             3613

Number of Slice LUTS                     4115 out of 207360   1%
Number of Slice LUT-Flip Flop pairs      5215 out of 207360   2%
```

Listing 6.3: FPGA Place and Route Utilization Summary - SPRAM and loop unrolled 16 times.

The full results for the device utilisation after the FPGA implementation (Place and Route stage) are presented in Listing 6.3. We will not consider all the results at this stage, but we turn our attention to the memory utilisation. As observed (entry RAMB36_EXPs), we are already using close to 20% of the resources available, even for a relatively small number of predictions to be checked (**it** = 10 000).

As our memory utilisation is the highest amongst all other components considered, it makes sense to optimise for it, as a reduction will allow the device to handle larger market data arrays and/or more predictions. The value of 64 BRAM modules is consistent with expectations, as memory with sizes equal to powers of two has been allocated, large enough to fit the underlying data. Therefore, to fit market data (100 times 100 elements), a memory size of 16K entries times 32bits is required (exchange rates stored as 32bit

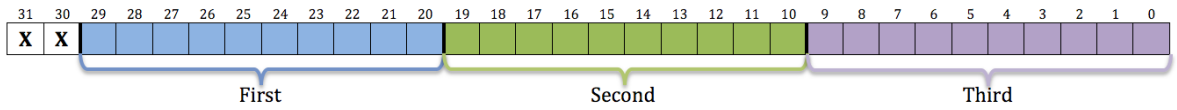

Figure 6.4: Compacting the predictions array.

floats). Since there were 10 000 predictions, a results array of 16K entries was allocated, each also storing a float, hence 32bits. Finally, we need a predictions array and the size of this may not be immediately obvious. Since triangular arbitrage concerns three different currencies, we need to store predictions (10 000 of them) and each of the 3 currencies (currently stored as 32bit ints) associated with the prediction. This makes 32K entries of 32bit words. As the BRAM units on the device are configured as 36K [44], we require a total of 64 units.

We would like to keep the RAM word size the same, that is 32 bits wide, but store more predictions in this space. The effect will not only be a decrease in the memory usage of the design, we should also see an increase in speed as a single memory access will be sufficient to retrieve all the prediction data per iteration.

If we were to use a market data size of 1024, the memory usage would be more than three times greater than that available on the device (this is not even considering prediction data or the results). Therefore, a reasonable solution is to limit the maximum number of currencies that can be stored in our system to 1024, requiring 10 bits.

The arrangement proposed is shown in Figure 6.4. We can assign the ten least significant bits to the third currency, the further 10 for the second and so on, leaving the first two bits of the word unused. The advantage of this arrangement is that it is simple to code (shift operations) and also maps easily to hardware, as we need only to connect the wires from the 10-bit registers to the appropriate pins.

Parameters		**Performance**	
n	100	period [ns]	9.679
it	10000	freq [MHz]	103.32
		# cycles	36251
		t [ms]	**0.351**

Table 6.6: FPGA Implementation results when compacting the prediction data array.

As can be seen in Table 6.6, the clock period is only slightly increased (by 0.271ns) when compared with the results for loop unrolling in the SPRAM case (Table 6.4). Crucially, however, the number of clock cycles has been reduced (by 23%), leading to an overall reduction in execution time by approximately 20%.

Furthermore, the primary objective of this step was to reduce the memory utilisation of the design. Crucially, the results obtained during "Place and Route" confirm this, as the memory utilisation (Number of RAMB36_EXPs) has been reduced to "`47 out of 324 - 14%`" (The full report has been omitted).

6.5 Result Space Reduction

Following on with the theme of reducing the memory utilisation on the device, we notice that the results array being used by the algorithm is not the optimal solution.

As it currently stands, the algorithm loops through all the prediction data and stores the result of each arbitrage calculation in the results array. These would form the basis of placing a trade, therefore we would need a separate algorithm to loop through the results and select the best trade. This approach would be quite slow.

However, we can combine these two procedures into a single algorithm. A logical approach is to limit the number of trades that can be placed during a single execution of the **prediction checking** algorithm. This is convenient, since the output of our design can be fed directly into an Electronic Trading platform, which would place the orders calculated using our approach.

As the prediction checking algorithm will be running continuously and we are minimising the latency to the range of milliseconds, it should be acceptable to limit the number of predictions, that can be sent for execution on the market, to a small number. For the purpose of this Thesis, we consider a single best prediction and two best predictions.

6.5.1 Single Best Prediction

This is the most straightforward modification, with the aim of finding only the best arbitrage opportunity from all the predictions iterated over. It also enables us to do without the results array and represents a considerable memory saving. Furthermore, as explained above, we do not need a second algorithm to iterate over the predictions to place an order, so this version of the design could be considered as an initial complete implementation, since the results could be sent directly to an Electronic Trading platform for execution.

The necessary modifications to the code are quite simple, as we only need to keep track of the best arbitrage opportunity. This can be represented by either the maximum or minimum value depending on whether the exchange rates or their inverses are stored. This will make no difference to the number of arbitrage opportunities found (as the data in our case is symmetrical) or the complexity of the computation.

One important observation when implementing this design is that the comparison variables (i.e. where the running best arbitrage opportunity is stored) should be kept local to the function and not stored in the Block RAMs. Otherwise the AutoPilot compilation results are very disappointing, resulting in considerable increases in the number of clock cycles required.

Parameters		**Performance**	
n	100	period [ns]	10.43
it	10000	freq [MHz]	95.88
		# cycles	44251
		t [ms]	**0.462**

Table 6.7: FPGA Implementation results when finding the single best arbitrage opportunity.

The results for the FPGA implementation are presented in Table 6.7. Initially, the clock cycle values, when keeping the loop unroll factor at 16, were rather disappointing (50626 cycles). This is why the factor has been increased to 40 for the above mentioned synthesis. This has allowed us to keep the total number of clock cycles down to 44251, without having a significant impact on the resource utilisation of the design (which is presented in Listing 6.4).

Number of DSP48Es	14 out of 192	7%
Number of RAMB36_EXPs	32 out of 324	9%
Number of Slices	2670 out of 51840	5%
Number of Slice Registers	5682 out of 207360	2%
Number used as Flip Flops	5682	
Number of Slice LUTS	5452 out of 207360	2%
Number of Slice LUT–Flip Flop pairs	8588 out of 207360	4%

Listing 6.4: FPGA Place and Route Utilization Summary - Single best arbitrage opportunity, loop unrolled 40 times.

The resource utilisation figures are comparable with the SPRAM Loop unrolling by factor 16 case (Listing 6.3). Most importantly we observe increases in the number of LUTs and Flip Flops as well as DSP48E slices. This is to be expected, since we have increased the complexity of the design. In fulfilment of the objective for this section, the memory utilisation has been reduced and is now at 9% (32% lower than the Compacting Data case - Section 6.4).

Returning to the FPGA Implementation results (Table 6.7), we observe that the clock period has been increased from 9.679 to 10.43, which again can be attributed to the increase in design complexity, following the introduction of an extra step of finding the best arbitrage opportunity. Unfortunately, as a result of the increase in the number of clock cycles, the total time for the execution of the algorithm has increased to 0.462ms (i.e. by approx. 32% over the unrolled case). This is not ideal, but we have saved time required for the run of a separate algorithm to find the best arbitrage opportunity. Moreover, we have managed to decrease the memory utilisation on the device.

6.5.2 Two Best Predictions

A logical extension to the above mentioned model is to increase the number of best arbitrage opportunities we store. This could be increased to an arbitrary number, but for the purpose of demonstrating the behaviour, we will stick to the two best values. The advantage from the implementation point of view, is that the logic to find these two values can be hard-coded to avoid any optimisation problems which may arise with AutoPilot.

Although the resource utilisation report for this scenario has been omitted from this write-up, the utilisation of LUT, Flip Flop and slices increases dramatically, to 9%, 11% and 12% respectively (compared with 2%, 4% and 5% for the single best case).

Of greater significance, however, is the dramatic increase in the clock period for the updated design, more than twice the time taken for the single best prediction. Part of this increase can be attributed to the more complicated design. In fact, the effort during

Parameters		Performance	
n	100	period [ns]	22.661
it	10000	freq [MHz]	44.13
		# cycles	48002
		t [ms]	**1.088**

Table 6.8: FPGA Implementation - Two best arbitrage opportunities, using loop unrolling factor 40.

the "Place and Route" phase of the Xilinx design flow had to be reduced, as even after 24 hours of computation the algorithm was nowhere near producing a finished design. It should be noted, that the reduced effort still took over 10 hours to complete.

The substantial decrease in performance (as seen in Table 6.8) of this approach means that we will look towards the single best prediction for the purpose of evaluation. Ideally, the decreased performance of the two best predictions design should be examined, as improving the design should be possible. A feasible approach would be to introduce pipelining, as new arbitrage opportunities can be computed before the output tables (with the best opportunities) are updated. To reiterate, we would expect the performance of this approach to deteriorate when compared with the single best value, but not by factors as significant as shown in this experiment.

Unfortunately, I have not had the time to implement and experiment with these optimisations, mostly as a result of the dramatically increased time required for the FPGA Implementation stage. A comparable process for the single best result case takes in the region of 30 minutes, not taking into account preparation of the design and source files.

6.5.3 Further Modifications

A particular advantage of the approach we have discussed when only keeping track of the best arbitrage opportunities, is as follows. It is possible to interrupt the computation at any time and immediately have a value representing an arbitrage opportunity, which the system deems to be best at that particular point in time. Of course, there is no guarantee, that this will be the best value once the algorithm has iterated through all the predictions. Regardless, it allows a trading system to probe the current maximum value and act by placing orders, for instance, when a pre-defined threshold has been exceeded.

6.6 Evaluation Preparation

As discussed in the previous section, we will continue our discussion based on the single best arbitrage opportunity design, as this is substantially faster than the latter approach. As we are primarily interested in the reduction of latency it is the logical choice.

In preparation for the evaluation, where different market sizes and numbers of predictions will be considered along with the implications of reaching full memory utilisation, we will consider a slightly revised model. As I have been allocating memory to arrays using sizes that are powers of two, these will not always be fully occupied. This might

lead to slightly erratic results, when going from one bounding memory size to the other. In theory, the unused memory regions are optimised out during the FPGA Implementation stage, but in my experience this was not always the case. Therefore, I will present a design which will use all of its allocated memory, hence we shall use market data and prediction arrays that are powers of two.

A further consideration is the loop unrolling factor. As discussed previously, we wish to avoid generating extra control logic when the number of loop iterations is not exactly divisible by this factor. As a result, it is best to choose a value, which is easily divisible by any number of iterations in the new scheme. Again, the choice is a power of two closest to the previously used loop unrolling factor of 40.

The process of arriving at the new design has been split into three stages. First, we look at adjusting the number of iterations of the loop, then increasing the market size and finally finding the optimal unroll factor.

6.6.1 Adjusting Predictions

The number of iterations (i.e. predictions) has been increased from **it** = 10 000 in the previous approach. The exact value of **it** used is 16 360. This is the closest value below $2^{14} = 16\ 384$, which is also divisible by 40 (the original unroll factor).

Thus modified, AutoPilot reports a clock cycle number of 72 394, which corresponds to a predictable linear increase ($\frac{72\ 394}{44\ 251} \approx 1.636$).

6.6.2 Adjusting Market Size

The following step is to increase the market data size, while keeping the number of predictions at the newly established value as above (16 360). The clock cycle count is actually decreased by approximately 0.5%, but this is not a significant reduction.

6.6.3 Loop Unrolling

The final step is to once again find the optimal loop unrolling factor. After running AutoPilot compilations with unroll factors ranging from 1 to 128 (powers of two only), the results have been plotted in Figure 6.5 (please note, no suitable trend-line could be fitted to the data). The value 32 has been selected as the optimal, as it still provides valuable performance increases over the unroll case 16. Please note, that for these runs the number of iterations (i.e. size of prediction data array) has been changed to $2^{14} = 16\ 384$, to ensure that it is divisible exactly by all the unroll factors under test.

6.6.4 FPGA Implementation

Now that all the parameters have been established, the final FPGA Implementation can be carried out. The results are presented in Table 6.9.

We first consider the approximately linear increase in the number of clock cycles for this implementation (only slightly increased due to the reduction of the loop unroll factor). More significantly, however, we experience a reduction in the clock period, a likely consequence of the decrease in loop unrolling. As a result, the time needed to

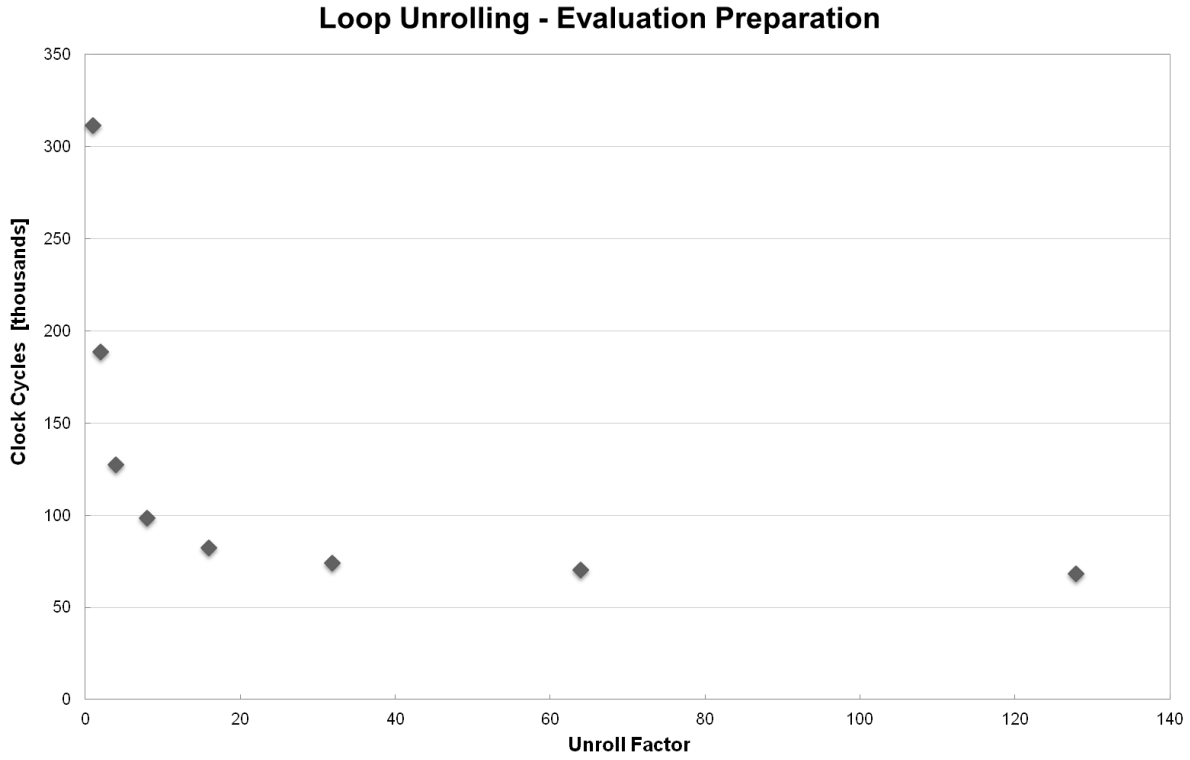

Figure 6.5: Finding the optimal unroll factor with allocated memory fully occupied.

Parameters		**Performance**	
n	128	period [ns]	9.461
it	16384	freq [MHz]	105.70
		# cycles	73729
		t [ms]	**0.698**

Table 6.9: FPGA Implementation for the final configuration. Loop Unrolling factor 32 and all parameters are powers of two.

compute all the iterations has risen less than the linear increase in the number of clock cycles and now stands at 0.698ms.

Having explored different algorithm optimisations on the FPGA architecture, we have reached a final design that will now form the basis of the evaluation in Chapter 7.

6.7 Design Remarks

I would like to take this opportunity to share some remarks regarding the design process as described in this chapter.

With regards to the FPGA implementation process following the generation of AutoPilot output, I feel that this is by far the most error-prone part of the design. As described earlier, multiple steps are necessary to prepare for the synthesis part using the Xilinx tools.

I am encouraged by news that Xilinx plan to incorporate AutoPilot into their design [26] flow. This move has great potential of reducing the amount of time and effort necessary to get from an algorithm coded in C to a viable hardware implementation on an FPGA. This could ideally lead developers to explore multiple optimisation options easily, while getting reliable performance estimates.

When working with AutoPilot, I was slightly disappointed by the increased complexity when designing algorithms using floating point numbers. I appreciate the increased complexity of Floating Point ALUs, but feel that as more designs are tested and implemented in hardware, this process could be made easier.

Having used the Xilinx tools mostly from the command line, I cannot comment on design using the GUI. I will say however, that the sheer amount of configuration options and the lengthy implementation process split up into multiple stages can be overwhelming when working with the tools for the first time. Having had previous experience with GPU architectures, specifically NVidia's CUDA, I feel that some work could be done to simplify this design process (possibly by hiding some of the more advanced functionality from novice users). However, I do appreciate that the FPGA is a more specialised environment.

6.8 Closing Remarks

Although there was not enough time to upload and execute the generated designs on physical FPGA devices, it must be noted that we have stopped just short of this step in the Xilinx design flow.

To give readers a feeling for the tangibility of the designs achieved, attached in Figure 6.6 is a screenshot of how the design would look like once placed on the device. The output has been generated using the FloorPlan tool (part of Xilinx tools) and shows an early design of the original algorithm accompanied by the memory wrapper (red, green and blue areas represent BRAMs allocated to the three arrays of the function).

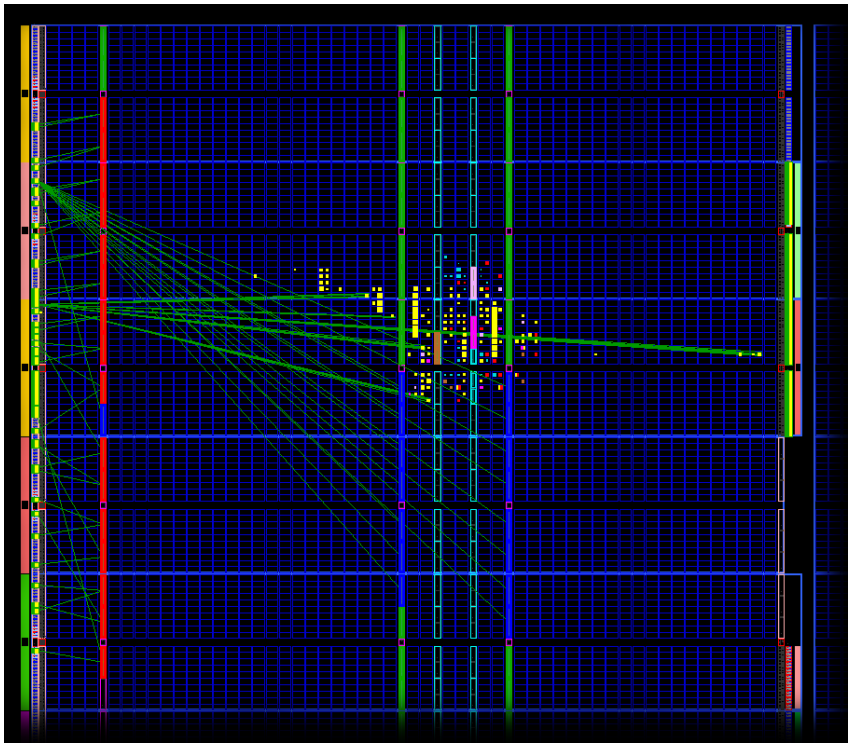

Figure 6.6: Implementation of the original algorithm on the Virtex-5 family FPGA.

6.9 Summary

In this chapter we have presented additional background material and information about AutoPilot, the Xilinx design flow and the implementation procedure used to prepare designs for the FPGA.

We have looked at the optimisation process carried out on the **prediction checking** algorithm, taking into account the importance of synthesising the RAM by using a **memory wrapper**. This is a more realistic approach, but takes longer to execute than the original implementation. Moreover, we have shown how **loop unrolling** can be used to increase the performance of the design and briefly explored using different types of RAM (SPRAM and DPRAM).

Attention has also been given to reducing the amount of the FPGA's BRAM resources necessary for the execution of the algorithms and we have explored compacting the predictions array representation, as well as reducing the number of arbitrage calculations that are stored in the system. Having investigated the performance of **Single Best** and **Two Best** modifications, we observed disappointing results for the latter. Moreover, the implementation process also took over 10 hours to complete, which supports our decision to select the **Singe Best** version.

Finally, we have prepared the FPGA implementation for evaluation (Chapter 7) and have remarked on the experience from the design process, also showing how the designs would look like when placed on the physical device.

Chapter 7

Evaluation

Continuing from detailed descriptions of the individual components, that is **arbitrage detection** (Chapter 4), **market data prediction** (Chapter 5) and **prediction checking** (Chapter 6), and their specific implementations, we can now concentrate on the evaluation.

However, before focusing on the performance of the proposed heterogeneous model, we revisit these components and analyse them from the point of view of varying market sizes, i.e. different numbers of currencies.

- **Evaluation Procedure:** We begin with a brief overview of the evaluation procedure for the individual system components.
- **Arbitrage Detection:** The arbitrage detection algorithms, Trade Size, Threshold and the Reference CPU Implementation are evaluated against different market sizes.
- **Data Prediction:** A short description of the performance of the data prediction algorithm follows, as the accuracy has already been established in Chapter 5.
- **Prediction Checking:** The FPGA prediction checking component is evaluated against different market sizes, numbers of predictions and when approaching full memory utilisation.
- **Heterogeneous Model:** After extrapolating the required data, the proposed system is evaluated as a whole, with both synchronous and asynchronous models being described. Moreover, we examine issues surrounding scalability, profitability and latency.

7.1 Evaluation Procedure

Building on the initial evaluation procedure carried out on the accuracy of the **data prediction** component (see Chapter 5), I have executed the algorithms we now consider over a range of generated market data files with varying numbers of currencies. This is to accomplish the goal of the evaluation, investigating the behaviour under different market sizes. Throughout this chapter, we will omit the exact results gathered from all the executions, but where appropriate will present the data in figures instead.

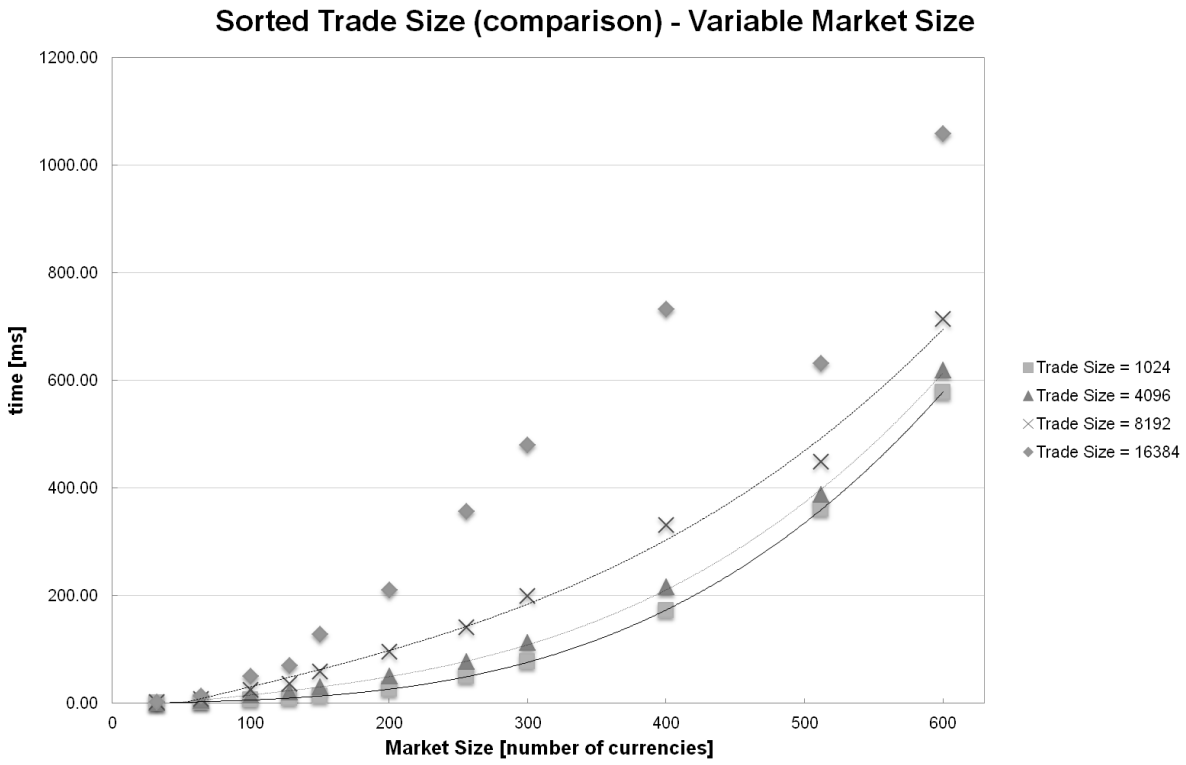

Figure 7.1: Trade Size Modification - Performance under different Market Sizes

The procedure for the **prediction checking** algorithm (FPGA implementation) is different as no market data files are used. However, the effects of varying market size are still observed and will be discussed in greater depth.

7.2 Arbitrage Detection

Firstly, we recall the **arbitrage detection** algorithms introduced in Chapter 4. After considering different implementation directions, three algorithms were selected: Single-Best, Trade Size and Threshold modifications. We now present their more thorough evaluation when varying the market size.

7.2.1 Trade Size

Let us begin with the Trade Size modification, which represents a sorted list of the best predictions available (recall Section 4.4.2). The results are shown in Figure 7.1 and have been collected for four different values of the **Trade Size** (i.e. number of best predictions we consider). Values above and equal to 1024 were selected, as this is where the greatest degree of variability lies.

In Figure 7.1, the data points for Trade Size values below 16 384 follow cubic trend-lines. This is to be expected as the sample space for all arbitrage opportunities is cubic with regards to the number of currencies. Furthermore, it is clear that increasing the number of predictions has a negative effect on the speed of the algorithm.

As the market data and the underlying **arbitrage detection** algorithm have been kept the same when changing the Trade Size value, this suggests that the overheads we are seeing (to increasing the trade size) are a result of the increased computation time spent on sorting the gathered predictions.

The effects become even more pronounced for a trade size of 16 384 (trend-line has been omitted), as we observe that the performance gap widens very quickly even for relatively small market data sizes in the region of 200 and above. Therefore it would be advisable to investigate the performance of alternative sorting algorithms, even though this was not possible due to time constraints.

Finally, we notice that the data points for the market data size of 512 seem anomalous and are below what we would have expected. After investigating this, I found the number of arbitrage opportunities in the generated market data to be the lowest of the runs used for evaluation. This directly impacts the performance of the algorithm, as only permutations that are deemed arbitrage opportunities are considered for sorting.

7.2.2 Threshold

Now we recall the alternative approach that eliminates the need for sorting. In the **Threshold** case (recall Section 4.4.3), only currency products above a certain threshold are considered and are stored in a fixed length list. Before presenting the results of this approach, let us first consider the final algorithm proposed.

7.2.3 Reference CPU Implementation (Max)

The last algorithm we look at is known as the reference CPU implementation (recall Section 4.4.1), also referred to as the **Max** algorithm. In this case, only the best arbitrage opportunity currently encountered is stored.

This approach has been taken to allow a reasonable comparison with the FPGA architecture, where a comparable algorithm was implemented (i.e. the "Single Best Predictor" - Section 6.5.1).

7.2.4 Results

Having briefly recalled the algorithms, let us now compare them under different market sizes. Results are presented in Figure 7.2.

One of the higher trade size values of 8192 has been selected for comparison with the Max and Threshold algorithms. Moreover, in the Threshold case we have enough space to store 16 384 predictions.

We immediately notice that the data points for the Max and Threshold cases are virtually identical (the triangles representing "Threshold" are plotted on top of the squares showing "Max"). This reinforces the good performance of the Threshold algorithm, which we saw when exploring **arbitrage detection**.

The "Threshold" approach does present disadvantages when compared with the sorted Trade Size implementation, because we cannot be certain that the best arbitrage opportunities will be captured in the list. Nonetheless, the performance difference between the

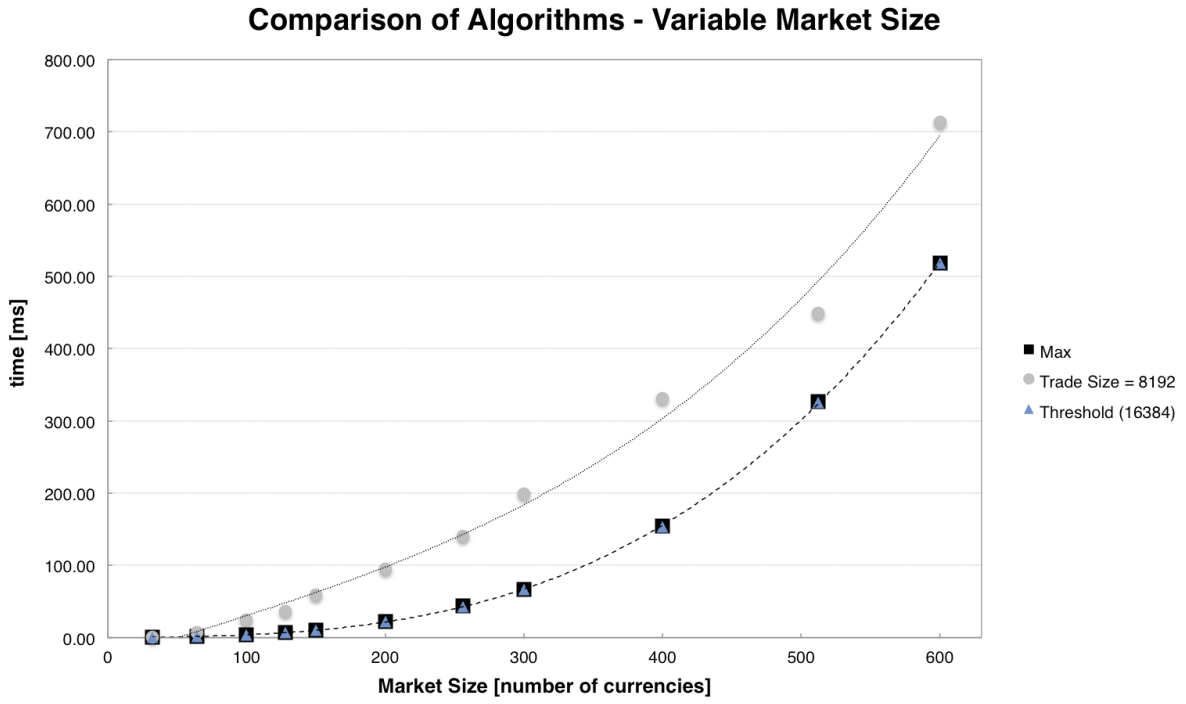

Figure 7.2: Arbitrage Detection Algorithms - Performance under different Market Sizes

algorithms must be considered and an appropriate trade-off reached between execution time and accuracy.

One final aspect to note, is that even the better performing "Threshold" algorithm still follows a cubic trend, which is representative of the complexity of the underlying **arbitrage detection** algorithm and its sample space.

7.3 Data Prediction

We now move onto the next system component, the prediction of market data. The accuracy of **data prediction** has already been evaluated in Chapter 5 (i.e. how well the predictor performs when finding arbitrage opportunities under the heterogeneous model). What we consider now is the execution time of the algorithm under varying market sizes.

Moreover, two alternative approaches were developed for **data prediction**: mean and exponentially weighted. Here we will only consider the performance of the more complicated approach, that is the exponentially weighted solution. Results are given in Figure 7.3. Please note, that the values for the **arbitrage detection** algorithm (**max**) have also been plotted to aid comparison between the components.

Firstly, we note that the exponential weighted predictor points follow a quadratic line of best fit, which is representative of the underlying complexity i.e. we must iterate over all the market data which is squared in respect to the number of currencies. This is one degree less than for the arbitrage detection case already explained and can be seen in the Figure 7.3 as the increasing performance gap when the market size is increased.

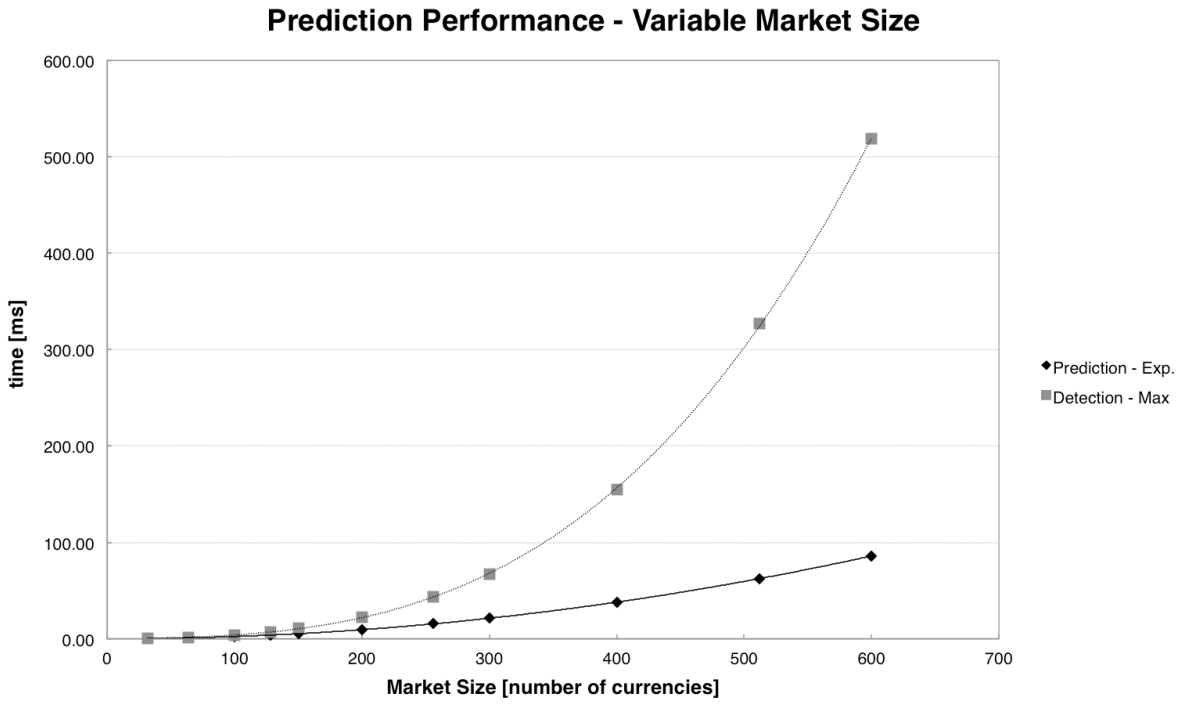

Figure 7.3: Exponentially Weighted Predictor - Performance under different Market Sizes

7.4 Prediction Checking

Before reasoning about the heterogeneous model, we consider the last individual component, the **prediction checking** implementation in hardware, on the FPGA.

As a reminder, this portion of the system is responsible for checking predictions, which were generated by the modelling component, to establish whether these form viable arbitrage opportunities. In Chapter 6, we covered the optimisation process and concluded with a design that should be well suited for the purpose of this evaluation.

Continuing the evaluation procedure presented for the previous two system components, we consider the effect of changing the market data size. However, extra attention must also be given to the effect of changing the number of predictions fed into the system, directly influencing the number of iterations (of the prediction checking algorithm) required.

Moreover, I would like to establish the performance of the system as the design approaches full device utilisation. As we have established the memory to be the limiting factor, this will form the basis of the analysis carried out.

7.4.1 Evaluation Runs

In Chapter 6, it was noted that keeping market size (**n**) and the number of predictions (**it**) as powers of two would be an optimal approach. Possible evaluation run configurations for the FPGA are shown in Table 7.1, representing firstly the number of predictions as a percentage of all the currency permutations (Pred %).

Moreover, the RAM sizes necessary have been calculated for both the market data

(M-RAM) and prediction data (P-RAM). Please note that the compact representation for prediction data has been used, so we only need 32bits for all three currencies that form an entry in the predictions memory. Moreover we are using 32bit floats for the market data. This ensures that both values fit within the Block-RAM resources, specifically the 36Kb blocks.

The device selected for these evaluation runs is the Xilinx XC5VLX330T (Virtex-5 family) and supports a maximum of 324 Block RAMs (of the 36Kb type) [44]. Therefore the RAM sizes noted in Table 7.1 are given in terms of Block RAMs required.

The "Total RAM" required is simply the sum of the market and prediction RAMs and enables us to calculate the expected **utilisation** of RAM resources on the device, considering the maximum number of available BRAMs being 324.

conf	**n**	**it**	**Pred %**	**M-RAM**	**P-RAM**	**Total RAM**	**Util [%]**
1	32	16384	55.05	1	16	17	5.25
2	32	32768	**110.11**	1	32	33	10.19
3	64	8192	3.28	4	8	12	3.70
4	64	16384	6.55	4	16	20	6.17
5	64	32768	13.11	4	32	36	11.11
6	64	65536	26.22	4	64	68	20.99
7	64	131072	52.43	4	128	132	40.74
8	64	262144	**104.86**	4	256	260	80.25
9	128	16384	0.80	16	16	32	9.88
10	128	32768	1.60	16	32	48	14.81
11	128	65536	3.20	16	64	80	24.69
12	128	131072	6.40	16	128	144	44.44
13	128	262144	12.80	16	256	272	83.95
14	128	524288	25.60	16	512	528	**162.96**
15	256	16384	0.10	64	16	80	24.69
16	256	32768	0.20	64	32	96	29.63
17	256	65536	0.40	64	64	128	39.51
18	256	131072	0.79	64	128	192	59.26
19	256	262144	1.58	64	256	320	98.77
20	256	524288	3.16	64	512	576	**177.78**
21	512	16384	0.01	256	16	272	83.95
22	512	32768	0.02	256	32	288	88.89
23	512	65536	0.05	256	64	320	98.77
24	512	131072	0.10	256	128	384	**118.52**
25	1024	0	0.00	1024	0	1024	**316.05**

Table 7.1: FPGA - Evaluation run configurations showing expected memory utilisation on the Xilinx XC5VLX330T device.

The various run configurations represent different market sizes (**n**) and numbers of predictions (**it**) giving an indication of the expected device utilisation. Configurations from this list will be selected during the evaluation of the FPGA implementation.

Let me mention again the reason for not using a greater number of data points for the purpose of the evaluation. This is a direct result of the time required to produce accurate

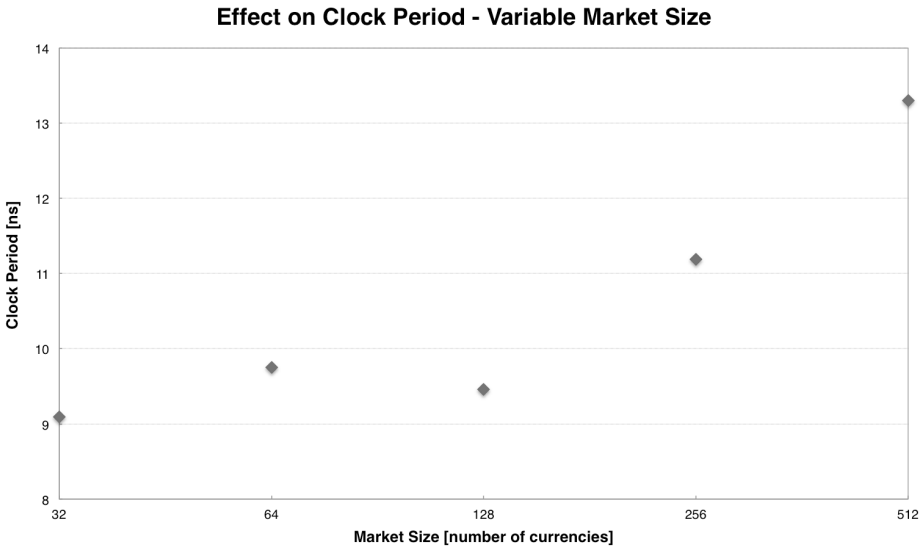

Figure 7.4: FPGA Implementation - Clock Period - adjusting the number of currencies.

FPGA implementation results, following modifications to the source code in AutoPilot.

7.4.2 Market Data Size

The first approach, mirroring the method for evaluating the CPU implementations is to adjust the market data size, while keeping the number of predictions constant at 16 384.

Values used for market size range from 32 to 512 (all powers of two). No smaller values have been considered, as having a market with 16 currencies is too small to be considered relevant. Moreover, taking 16 384 predictions is more than 4 times the number of all currency permutations possible for this market size. Finally, a market size of 1024 could not be considered as even without any prediction data, this would not fit on the device (see Table 7.1, run config 25).

n	32	64	128	256	512
period [ns]	9.094	9.755	9.461	11.191	13.303
freq [MHz]	109.96	102.51	105.70	89.36	75.17
# cycles	73729	73729	73729	73729	73729
t [ms]	0.670	0.719	0.698	0.825	0.981

Table 7.2: FPGA Implementation timing results - keeping iterations constant at 16 384 and adjusting the market data size.

The timing results for this evaluation run are presented in Table 7.2 and Figure 7.4 (only Clock Period). Starting with the more obvious analysis, that is the number of cycles, we observe that these stay constant for different number of currencies. This is to be expected, as it is the number of predictions that governs the number of iterations the algorithm needs to execute.

Much more interesting is the pattern the clock period follows for the above data. If we look at the three rightmost points (Figure 7.4), these form a clear trend. The greater

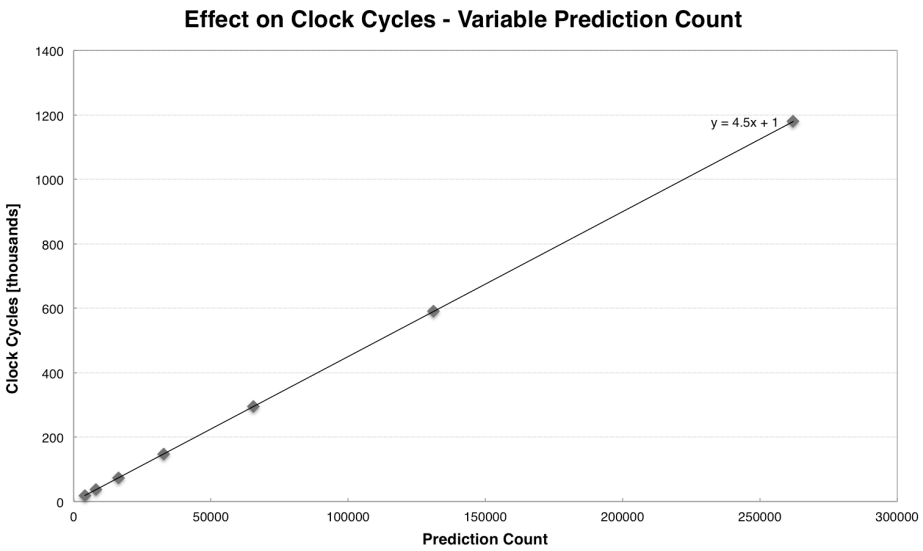

Figure 7.5: FPGA Implementation - Clock Cycles - adjusting the number of predictions.

the market size, hence greater memory utilisation (run configurations 9, 15 and 21 in Table 7.1), the longer the clock period. This effect can likely be attributed to assembling larger memories from the BRAM resources on chip. As these resources are limited, far away regions of the chip must be connected, resulting in greater delays.

Unfortunately, results for market size of 32 and 64 appear to be anomalous, as they do not follow the expected trend. Some reasons for this will be explored in the following section.

7.4.3 Prediction Data Size

Now we shall keep the size of the market data constant (at $\mathbf{n} = 128$) and adjust the number of predictions which will be checked.

log2 (it)	12	13	14	15	16	17	18
it	4096	8192	16384	32768	65536	131072	262144
period [ns]	10.545	10.301	9.461	9.937	10.434	11.265	12.425
freq [MHz]	94.83	97.08	105.70	100.63	95.84	88.77	80.48
# cycles	18433	36865	73729	147457	294913	589825	1179649
t [ms]	0.194	0.380	0.698	1.465	3.077	6.644	14.657

Table 7.3: FPGA Implementation timing results - keeping **n** constant at 128 and adjusting the number of predictions.

Results for this procedure are given in Table 7.3. The number of iterations has been doubled for each consecutive data point starting with $2^{12} = 4\ 096$ and finishing at $2^{18} = 262\ 144$.

In Figure 7.5, as well as from the data just presented, we observe the relationship between the prediction count and the clock cycles to be linear. This was to be expected,

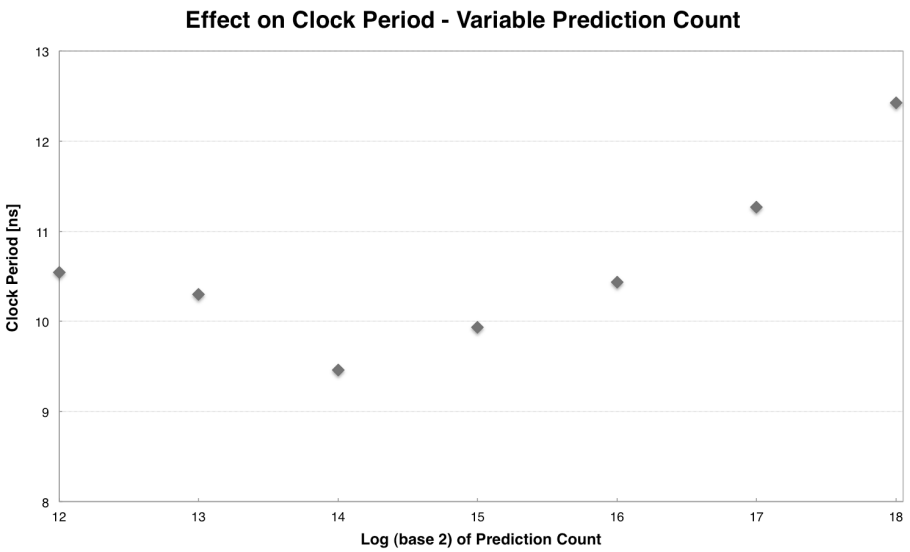

Figure 7.6: FPGA Implementation - Clock Period - adjusting the number of predictions.

as increasing the prediction size is directly proportional to the number of iterations of the algorithm. Moreover, the data (and equation of the trend-line) indicate we require 4.5 clock cycles to compute one prediction.

Having established this relationship, we now concentrate on the clock period for the design. This has been graphed in Figure 7.6. Taking, for the time being, log values of 14 and above, once again these points follow an expected trend, associating the progressively larger memory utilisation (run configurations 9 to 13 in Table 7.1) with an increase in the clock period.

Unfortunately, this cannot account for the prediction sizes of 4096 and 8192 (log values 12 and 13 respectively). According to the above reasoning, we would logically expect these to follow the previous pattern and decrease. We also noted similar behaviour when adjusting the market size, so there at least seems to be consistency in this respect.

There could be various reasons for this behaviour and I present a few hypotheses. One possibility is optimising behaviour of the Xilinx tools used (map and place and route), which might be generating sub-optimal output for very low device utilisation. Furthermore, we also notice that where the clock periods are at a minimum, the BRAM resources required for market data and predictions are equal. Moreover, It could be the case that the optimisation carried out in Chapter 6 have led to a local maximum for best performance. This could be checked by carrying out a similar procedure for these smaller memories.

Even with these anomalies, if we were to plot the time necessary to execute the algorithm (taking both clock period and the number of cycles), the linearly behaving clock cycle trend would be the predominant factor.

7.4.4 Maximum Memory Utilisation

The last stage of the evaluation process for the FPGA is to determine the behaviour of the design when the memory utilisation approaches 100%. Market data sizes of 64, 128,

256 and 512 have been selected for this process, as other sizes (i.e. 16, 32, 1024) would either not allow for close to full utilisation of the memory or simply would not fit on the device (see Table 7.1 for clarification - run configurations proposed are 8, 13, 19 and 23).

Placement and Routing of the 64 and 128 market sizes posed little challenge as these designs have memory utilisation in the region of 80% - 85% . This was not the case for a market size of 256, where the process would fail with errors suggesting insufficient resources on the device, despite Block RAM utilisation being approximately 98% (as predicted). Errors during place and route indicated problems with the cascade, suggesting that the BRAM modules could not be connected. The solution suggested by the tools was to try an alternative placement algorithm, but this did not solve the problem. As an alternative solution, I had to reduce the number of prediction iterations to 250 016 to allow completion.

Finally, the process for the memory size of 512 was equally troublesome. I tried the approach of reducing the number of currencies, but this did not provide compilable results until a reduction to 450. This design was also characterised by a much increased clock period of above 18ns, reinforcing the decision to investigate parameters with values of powers of two. However, a workable solution was reached by reducing the number of predictions to 49 152 (half way between the initial attempt and the prediction count for the lower step of $2^{15} = 32\ 768$).

mem util [%]	80.25	83.95	98.77	88.89	98.77
n	64	128	256	512	512
it	262144	262144	250016	32768	49152
period [ns]	13.707	12.425	12.436	12.991	14.886
freq [MHz]	72.96	80.48	80.41	76.98	67.18
# cycles	1179649	1179649	1125073	147457	221185
t [ms]	16.169	14.657	13.991	1.916	3.293

Table 7.4: FPGA Implementation timing results - approaching full memory utilisation.

The results for the runs which successfully completed the "Place and Route" stage are given in Table 7.4, with the memory utilisation values taken from the "Map" stage. The clock period is consistently high for these designs and likely attributed to the increased distances between the connected BRAM modules. Furthermore, experience from the implementation stage suggests that memory utilisation values of above 95% should be avoided, as it may not be possible to place them on the FPGA device.

7.4.5 Final FPGA Remarks

I now present a few points to quickly summarise the evaluation of the hardware implementation.

Taking the approach of using powers of two for the market and prediction array sizes has been beneficial and allowed easier prediction of the memory resources necessary. However, more experimentation would be necessary to evaluate the precise implications, especially on performance, of using alternative size arrangements.

The data consistently shows a linear relationship between clock cycles and the number of predictions checked. This is an important property, as the algorithm can be tuned easily by adjusting the number of predictions to suit needs, with the effect on performance easy to foresee.

Care should be taken when selecting the market data size for the design. The safest approach would be to re-run the attempted optimisations from Chapter 6 to ensure best performance results for the desired size. However, we must remember that the number of currencies would be changed less frequently than the number of predictions (that can be used for tuning), presenting less of a disadvantage.

Finally, further optimisations to the design could be explored, with pipelining offering the possibility of further improvements, especially in cases where we experienced large clock periods.

7.5 Heterogeneous Model

Having analysed the individual components, we can now turn our attention to the evaluation of the proposed heterogeneous model as a whole.

There are three ways in which we will approach this evaluation, each one considered in the following sections. Firstly, we compare the performance of the individual components and assess their behaviour in a synchronous or asynchronous model. Secondly, we look at the latency of the proposed system by comparing the **prediction checking** algorithm on the FPGA with a reference CPU implementation. Finally, we comment as to the profitability of the heterogeneous model.

Before we are in a position to carry out this evaluation, we must prepare the necessary data through the process of extrapolation.

7.5.1 Assumptions

One more consideration centres around how market data is supplied to the system. As explained in Chapter 3, the data is fed directly into both the CPU and FPGA components. Here we make the assumption that the time necessary for the data to be updated in memory is the same for both architectures. In other words, we evaluate the system from the point in time at which the latest market data is already available in memory.

7.5.2 Procedure

We now briefly consider the evaluation procedure. I propose to carry out comparisons on a range of market data sizes. Each one has associated with it a specific number of currency permutations, $n(n-1)(n-2)$ to be exact, referred to as the **sample space**. The percentage of predictions used in the runs will be kept constant at approximately 0.8% in order to allow for fair comparison between the different numbers of currencies.

Table 7.5 presents the configuration runs with the number of predictions chosen in order to keep an approximately constant percentage of all permutations.

n	32	64	128	256	512
permutations	29760	249984	2048256	16581120	133432320
predictions	256	2048	16384	131072	1048576
%	0.86	0.82	0.80	0.79	0.79

Table 7.5: Heterogeneous Model - Evaluation Runs.

7.5.3 Arbitrage Detection Extrapolation

Unfortunately, the evaluation procedure for **arbitrage detection** did not contain all the results necessary for our runs. We therefore extrapolate the required values. We also note that the **arbitrage detection** implementation chosen is the "Threshold" algorithm, as it offered better performance when compared with the sorted approach.

The initial results have been taken from the evaluation of the arbitrage detection algorithms (see Section 7.2.4 and Figure 7.2). Moreover, values from the **arbitrage detection** implementation (see Chapter 4) have been used, specifically Figure 4.4 with an arbitrage limit of 0.9.

We observed a constant performance for the "Threshold" algorithm for values below 16 384 predictions. A trend-line was fitted to the points above this value and points extrapolated as a percentage change above the baseline performance of the 16 384 case. This is presented in the row "correction" in Table 7.6.

n	32	64	128	256	512
t [ms]	0.31	1.37	7.09	43.33	325.80
correction [%]	0	0	0	**3.75**	**37.78**
detection [ms]	0.31	1.37	7.09	44.95	448.87

Table 7.6: Arbitrage Detection - extrapolated data.

Please note that no correction is necessary for the market data size of 128, as we are using the original data point. Furthermore, as we observed the performance of the "Threshold" algorithm to have a baseline for values below 16 384 (i.e. no performance change for smaller values), no corrections need to be made for the two smallest market data sizes.

After applying the correction to the original data, we obtain performance (timing) results for **arbitrage detection** which correspond to the run configurations selected.

7.5.4 Prediction Checking Extrapolation

We need to apply a similar procedure to obtain **prediction checking** (i.e. FPGA implementation) data.

The original values have been sourced from Table 7.2 and are presented with modifications in Table 7.7. The numbers in bold have been extrapolated.

$$ClockCycles = 4.5 \times Predictions + 1 \tag{7.1}$$

n	32	64	128	256	512
period [ns]	9.094	9.755	9.461	**12.436**	**14.886**
freq [MHz]	109.96	102.51	105.70	80.41	67.18
# cycles	**1153**	**9217**	73729	**589825**	**4718593**
t [ms]	0.010	0.090	0.698	7.335	70.241

Table 7.7: Prediction Checking - extrapolated data.

We have already determined there is a linear relationship between the number of predictions and clock cycles (recall Figure 7.5). As a matter of fact, this can be represented by Equation 7.1, which has been used to extrapolate the number of clock cycles. We note that all the gathered FPGA implementation results followed this formula closely.

As we were either decreasing or keeping constant the number of predictions, the clock periods for market sizes 32, 64 and 128 have been left the same. For data points 256 and 512, they were updated to the largest values seen, observed during the maximum memory utilisation evaluation (see Table 7.4). If we recall this evaluation run, the maximum number of predictions that could fit on the device with a market data size of 512 was 49 152. Therefore, we observe that the evaluation run for $n = 512$ would not fit on the device.

What we would need to do, is either split the **prediction checking** across multiple FPGAs (22 to be exact) or connect an external RAM to the device. In the former case, were we to use the FPGA devices in sequence for iterating through all the predictions, we could expect performance of the degree that has been extrapolated.

7.5.5 Synchronous Model

Having evaluated the individual components and collected all the necessary data, we can finally discuss the performance of the heterogeneous model. A possible approach would be to define a synchronous system.

Looking back at the individual components, we can execute the **data prediction** algorithm, which will be followed by **arbitrage detection**. At this point, we must transfer the predictions array to the FPGA, where **prediction checking** will be carried out.

When defining a synchronous model, we would execute all the components except **prediction checking** in sequence, with the summed execution time representative of how frequently we can execute a complete re-computation of arbitrage opportunities. We can think of this as the time required to prepare all the necessary data for the on-line FPGA component to begin execution.

The results for the individual components are summarised in Table 7.8. The first two rows contain data from the **data prediction** and **arbitrage detection** algorithms. The time to transfer the predictions from the CPU to FPGA has also been computed (**transfer** time). Since every prediction entry can be packed into a 32bit word, we know the size of the predictions array and can estimate the transfer time. A bandwidth of 1000Mb/s has been assumed and corresponds to the speed of the network interface present on the device [44].

n	32	64	128	256	512
prediction [ms]	0.24	0.98	3.90	15.56	62.35
detection [ms]	0.31	1.37	7.09	44.95	448.87
transfer [ms]	0.008	0.066	0.524	4.194	33.554
total [ms]	0.56	2.42	11.51	64.71	544.78

Table 7.8: Heterogeneous Model - Sum of components' execution times.

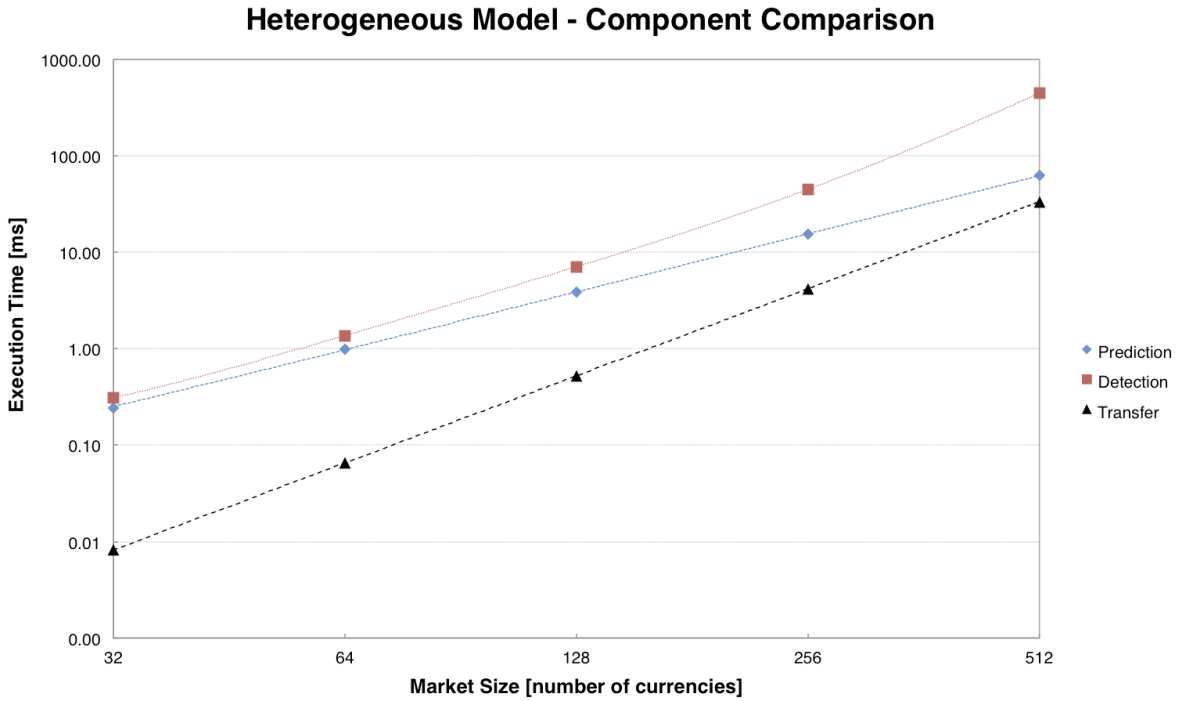

Figure 7.7: Heterogeneous model - comparing the performance of system components.

The results are shown in Figure 7.7 (please note both x and y axis use logarithmic scale). Immediately, we observe that the arbitrage detection algorithm takes longest to execute, with the value increasing quicker than the prediction portion of the model (with increased market size). This is the case, as the sample space for arbitrage detection is cubic, whereas for data prediction we iterate through the market data array, which is squared with respect to the number of currencies.

Moreover, the transfer time is originally marginal compared with the two algorithms, but becomes more pronounced as the market size increases, approaching the execution time of the data prediction algorithm. It would be worthwhile therefore, to consider optimisations possible for the transfer of data to the FPGA. Apart from looking at faster transfer (i.e. different IO, PCI Express) we might optimise the process by starting the prediction checking before all predictions have been transferred to the FPGA. This would reduce the amount of time the FPGA needs to wait before beginning to process new predictions and is possible, as the nature of access to the predictions array is inherently sequential.

Furthermore, we see that the total time necessary to prepare the predictions for execution (Table 7.8) increases sharply, especially when increasing the market size from 256 to 512. If we were to use the synchronous model, this would limit the frequency at which we could execute a complete re-computation (see Table 7.9). Whereas for a market size of 128, we could handle market updates 86 times a second, this is reduced to below 2 for a size of 512. The assumption here is, that upon receiving market data we always execute a complete re-computation and afterwards run prediction checking on the FPGA.

Unfortunately, I have not been able to find data for the frequency of market ticks in the foreign exchange market, or what an arbitrage detection system would need to cope with. However, as we consider larger market size, clearly we would expect there to be more activity. This necessitates more frequent updates, hence the performance of the synchronous model scales very poorly.

n	32	64	128	256	512
model total [ms]	0.56	2.42	11.51	64.71	544.78
synch. freq. [1/s]	1776.86	413.99	86.87	15.45	1.84
fpga [ms]	0.01	0.09	0.70	7.34	70.24
checks till update	53.67	26.87	16.50	8.82	7.76

Table 7.9: Synchronous vs. Asynchronous Model.

7.5.6 Asynchronous Model

Thankfully, we are not restricted to running our system components in this synchronous "lock step". We now present a model for asynchronous interaction.

The idea is, we do not completely recompute the predictions after every market tick. Instead, the same predictions are checked against live data multiple times. These are only updated when there has been sufficient time to recompute. Table 7.9 presents how many live prediction checks can be executed on the FPGA, before new updated prediction data is ready (row "checks till update").

With this approach, we can now handle market data updates as frequently as the time it takes to execute the prediction checking algorithm on the FPGA. In the case of a market size of 256, this would be approximately 7.34ms. If we need to react to market data updates that come in more frequently than this, we might need to reduce the number of predictions that are checked on the FPGA or think about splitting the computation onto multiple FPGA devices. This could be accomplished by dividing the predictions amongst all the FPGAs and concurrently executing the **prediction checking** algorithm on the devices. There would be an extra overhead of comparing for the best arbitrage opportunities between the FPGAs, but we would expect this to be offset by the increase in performance. We must also note, that all devices would need to store the same market data, as we cannot foresee which currencies will form a prediction. However, it is feasible to develop more complicated methods of splitting the computation so that certain currency market data and predictions are associated with a particular device.

Furthermore, the performance of the asynchronous model is closely related to the accuracy of the data prediction. The component would likely need to be tuned to provide predictions which span multiple market data ticks.

7.5.7 Latency

We noted that in the asynchronous case, the latency of the heterogeneous model can be taken as the execution time of the **prediction checking** algorithm on the FPGA. We now evaluate the latency of the proposed heterogeneous model, by comparing with an optimised reference CPU implementation.

For this purpose, we use data from the evaluation of the reference CPU Implementation (see Section 7.2.3 and Figure 7.2). Table 7.10 compares this with the results of the **prediction checking** algorithm running on the FPGA. The results have also been plotted in Figure 7.8.

n	32	64	128	256	512
detection (max) [ms]	0.29	1.38	7.23	43.84	326.75
pred. checking (fpga) [ms]	0.01	0.09	0.70	7.34	70.24
reduction [%]	96.33	93.46	90.35	83.27	78.50

Table 7.10: Heterogeneous Model - Reduction in Latency.

The heterogeneous model exhibits a substantial reduction in latency, ranging from approximately 90% for the case of 128 currencies to 78.5% for a market size of 512. Latency is a critical part of the evaluation, as without achieving speedup in this respect a model cannot possibly solve the problem of arbitrage (i.e. be the first one to fill the order). Thankfully, as is the case with the proposed heterogeneous approach, a considerable reduction has been achieved.

Furthermore, we observe that the performance gap between the FPGA and reference CPU implementation narrows as the market size is increased. Nevertheless, this still stands at 78.5% for 512 currencies.

It is also important to consider the current number of currencies. The list from "XE.com" [41] shows 168 different currencies (ignoring expired ones). Furthermore, we

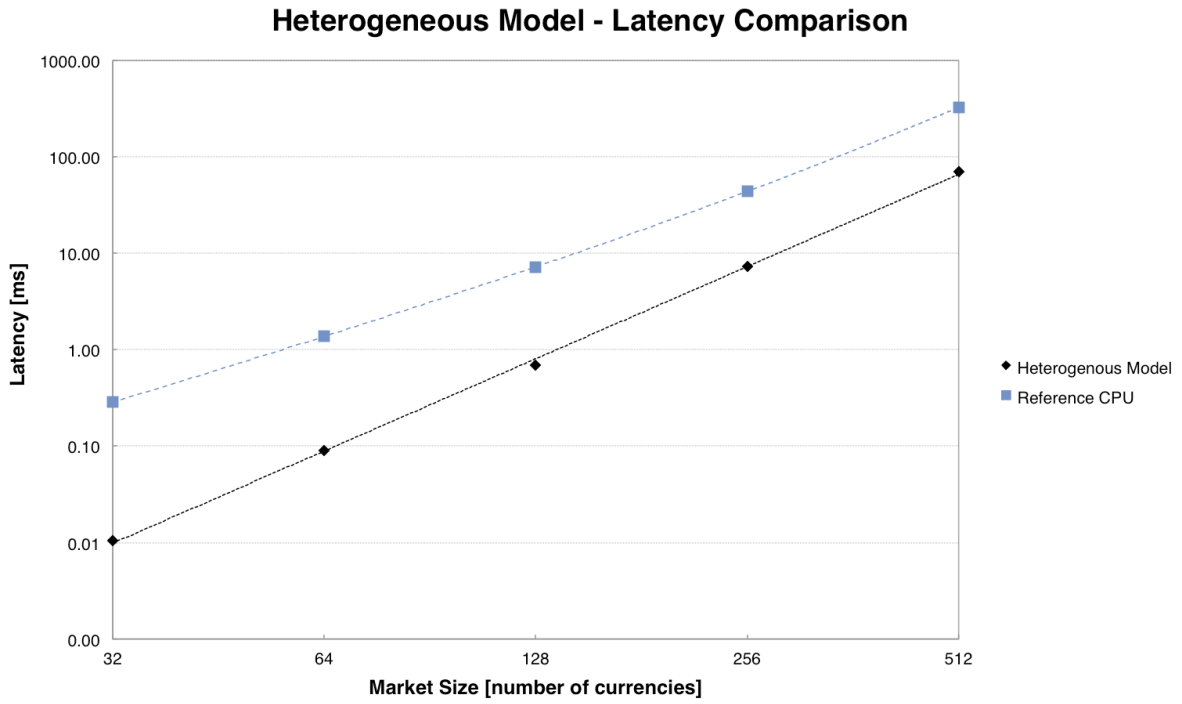

Figure 7.8: Heterogeneous Model - Reduction in latency.

note that the proposed model performs well in the region of 128 to 256 currencies and the implementation can also fit onto a single FPGA device.

The reduction in latency has been achieved by using the heterogeneous model, specifically decreasing the sample space by creating a list of predictions. The second contributing factor, is that these predictions are checked against live data using an FPGA architecture, which is well suited to minimising response time.

7.5.8 Profit

Establishing latency reduction is not enough to show the profitability of the proposed system. Here the predictor performance as well as market characteristics would need to be taken into account.

Regrettably, I have not been able to evaluate the exact profit that the proposed heterogeneous system could deliver. This is partly due to the underlying characteristics of arbitrage, where the first person to execute the transaction gains all the profit (i.e. winner takes all). The next person to execute will in the best case make no profit at all, but could also take a loss. The difficulty lies with not having access to real market data and not being able to assess where this cut-of point lies. If I were to come up with a model for this behaviour, I feel that it would be too simplistic for the purpose of a thorough evaluation.

In addition, we must also recall that the data available for this project has been randomly generated, which provides a challenge to assessing the profitability of the system. Given more time and access to real market data, a more thorough evaluation of the profitability could be conducted. However, due to costs charged for the use of live market

data, I appreciate that this may not be possible in an academic setting. Therefore, an alternative solution would be to carefully develop a model for the behaviour of the foreign exchange market in low latency (high frequency) environments. The profitability of the proposed system could then be evaluated against this model.

However, I would not like to leave the reader without providing at least an estimate of the expected profitability of the system. We turn to a study on low latency interaction in the Equities space, where trading systems are responding to market activity within 2-3ms [9] and apply the data to Foreign Exchange.

We would expect that, as the response time increases, there will be a sharp drop-off in the probability of executing an arbitrage trade. We might fit a model to this market, extrapolated from our considerations of "Exponential Smoothing".

$$Probability\_Of\_Execution = (\frac{1}{2})^{time} \tag{7.2}$$

Equation 7.2 estimates the probability of execution, based on the time (in milliseconds) elapsed since the arbitrage opportunity arose. Therefore, at the initial time step one would expect to gather all the profit and the chance of successful execution decreases exponentially, the longer we wait.

Now we take the average response time of 2.5ms and a 70% reduction in latency (the lowest value from our evaluation was 78.5%, for a market size of 512). If we apply these figures to the model suggested (in Equation 7.2), we would expect an increase in the execution probability of approximately 3.25 times (1.62 times for 30% reduction in latency).

The final step is to consider these values in the context of arbitrage. As already noted, the "TABB Group estimates that annual aggregate profits of low-latency arbitrage strategies exceed $21 billion, spread out among the few hundred firms that deploy them" [23]. Even if we estimate the market participants at 700, this means that an average firm would expect to profit in the region of $30 million. We should also give a generous figure for the development costs necessary to implement the heterogeneous system, at 20% of the average profit.

Taking this information into consideration, we would expect an increase in revenue of $61.5 million when reducing latency by 70%, ($12.6 million for 30% latency reduction). To clarify, these figures are given for a firm, representative of the average market participant.

7.5.9 Final Remarks

Using the heterogeneous model as a starting point, there are multiple refinements that could be made. One might consider an interruptible asynchronous solution, where the **prediction checking** algorithm may, when its performance deteriorates, interrupt the prediction and detection components and request updated (or at least partial) predictions.

Although the results show a reduction in latency, we might deem the comparison between the CPU and FPGA architectures to favour the former. The processor used for benchmarking is part of the latest Intel micro-architecture and is based on a 32nm process [25]. The Xilinx FPGA on the other hand is part of the 65nm Virtex-5 family, representing an older design [44]. A fairer comparison would use the latest FPGA architecture, that

is the Virtex-7 (28nm) [43], and we would expect the performance gap between the CPU and FPGA to increase.

We observed with the FPGA implementation, that the on board-memory becomes constrained for a market size of 512. This problem could be solved by using devices with more BRAM resources, such as the XC5VSX240T. Containing 516 36Kb Block-RAMs [44], this would handle approximately five times more predictions than in our current case. However, following Moore's Law, the latest devices also come with more memory (up to 85Mb on the Xilinx Virtex-7 [43]).

Furthermore, we observed that the performance of the system does not scale well for larger market sizes. Considering the algorithms developed, this is understandable, as the sample space is cubic with regards to the number of currencies. The reason lies partly with our assumption that all possible currency pairs are tradable. This has had the effect of increasing the number of currency permutations we look for, as in the foreign exchange market not all currency pairs are listed as exchange rates (i.e. not all currencies can be exchanged using one trade). Changes in this respect would allow us to speed up the computation, but the underlying trend of considerably increased complexity for greater market sizes should still hold.

We must also remember, that the algorithms can be adapted to search for arbitrage opportunities in markets other than foreign exchange. As the number of traded products increases and the trades are executed more frequently, this makes the problem of a complete re-computation progressively harder considering the cubic order of the proposed algorithms.

In order to solve this problem, we would need to look towards more complicated **arbitrage detection** algorithms. An opportunity for reducing the sample space we consider during each re-computation, is to search for arbitrage only in the currencies (products) that have been traded (updated) since the last execution took place.

It is possible to extend the arbitrage problem, so that we search for opportunities concerning four products [36], that is one dimension higher than we have considered in the course of the project. When adapted to deal with this increased complexity, the original detection algorithm would become $O(n^4)$. In this scenario it would be especially important to look towards an alternative approach.

Finally, we must also remember, that the quality of the **data prediction** will likely have a profound effect on the performance of the entire system, affecting the profitability of the heterogeneous model. To remind the reader, it is assumed that experts in quantitative finance would have access to such predictors and could then integrate their **data prediction** algorithms into the proposed heterogeneous system, giving a better idea of the profitability. As such, the core consideration for the Individual Project is a reduction in latency, which has been achieved and is substantiated in our evaluation.

7.6 Summary

In this chapter we have presented an evaluation of the heterogeneous model, which has been proposed for detecting triangular arbitrage. We proceeded primarily by evaluating the impact of changing the market size (i.e. number of currencies). Initially we revisited the individual system components to assess their performance under different market

sizes, before continuing with the discussion on the heterogeneous model.

For **arbitrage detection**, we compared the performance of the Trade Size, Threshold and Reference CPU Implementation and selected the Threshold modification for further investigation, as it provided reduced execution times over the Trade Size approach.

Next, we briefly discussed the performance of the **data prediction** algorithm under varying numbers of currencies, as the prediction accuracy had already been covered in detail in Chapter 5.

Moreover, we looked at the **prediction checking** FPGA component and evaluated it against different market sizes, while also considering the number of predictions. We found a clear relationship between the number of clock cycles for the design and the amount of predictions processed, providing a simple and predictable way of tuning the performance of the algorithm. Additionally, we explored the effects of approaching full memory utilisation, which resulted in a longer clock period. We also noted that the maximum market size than can fit on the considered Virtex-5 family device is 512 currencies, albeit with a rather low number of predictions (i.e. 49 152).

After extrapolating the required results, we began the evaluation of the heterogeneous model as a whole. We first discussed the performance of the model under a **synchronous** scenario and found that it scales very poorly for larger market sizes. We also presented a modified **asynchronous** approach which performed better in this regard, but relied on reusing the same predictions over multiple executions.

Furthermore, we showed that when compared with a reference CPU implementation, the heterogeneous model provides a reduction in **latency** in the range of 83% - 90% for market sizes of 256 and 128 currencies. This is a critical characteristic for a system dealing with arbitrage, as only the first system (person) to execute a trade can profit.

Additionally, we discussed how the profitability of the heterogeneous model could be evaluated and reasons for not being able to complete this step for the purpose of the project.

Finally, we ended our investigation of the heterogeneous model by concluding that alternative **arbitrage detection** algorithms would need to be developed in order to reduce the sample space under consideration and improve the scalability of the system under increased market size or arbitrage of greater degrees (i.e. trading of four products).

Chapter 8

Conclusions and Future Work

In this thesis I have presented a heterogeneous model for the detection of triangular arbitrage and shown how the collaboration between financial modelling and on-line systems can be accomplished. The approach taken was to separate the problem into **arbitrage detection**, **data prediction** and **prediction checking**.

I developed arbitrage detection algorithms and proceeded to optimise their implementations on the CPU architecture. This was done to ensure fair comparison with other system components. Furthermore, I introduced two market data prediction approaches and found them to perform at above 80% of the perfect predictor accuracy.

Together, the arbitrage detection and data prediction components formed the basis of a modelling system and were responsible for generating predictions to be sent to the FPGA. These recommendations were then processed by the prediction checking algorithm, representative of the on-line system, as the predictions would be evaluated against live market data.

Furthermore, I explained the heterogeneous model and how it could be formed from the individual components. Additionally, I provided considerations as to the operation under both synchronous and asynchronous scenarios and proceeded to evaluate the system under varying market sizes.

8.1 Conclusions

I will now give the most important conclusions that can be drawn from my work, focus on the results which have been delivered and discuss limitations.

On the subject of **arbitrage detection**, the recursive formulation of the algorithm should be avoided, not only on the basis of performance, but also, because the optimised version improves portability of the code for other architectures. I attempted to implement the algorithms on a GPU, but could not draw any benefits from this transition, concluding that as presented, the arbitrage detection algorithm is best suited to execution on the CPU. Moreover, it appears that multi-threading does not provide any performance advantages for the algorithm.

I evaluated different approaches to gathering the predictions and found that the "Threshold" modification performs just as well as the reference CPU implementation. The alternative sorted approach, "Trade Size", scales poorly for larger numbers of predictions. However, its advantage is the guarantee that we get the best predictions out of

all the possible currency permutations. Therefore, a trade-off can be observed between execution time and prediction quality with the performance advantages of the "Threshold" version being favoured for a low-latency system.

Regardless of which modification is chosen (for storing/sorting the predictions), the underlying arbitrage detection algorithm still scales poorly with increased market size and is $O(n^3)$ with respect to the number of currencies, representative of the underlying sample space. As market size or arbitrage degree increase, alternative arbitrage detection algorithms should be considered in order to improve scalability.

I examined two **data prediction** approaches and found that under the arbitrage detection scenario, predictor accuracy is not solely related to how closely the market data is modelled. Based on results from the square difference calculation, I conclude that the detection of patterns leading to arbitrage opportunities should also be considered when developing a refined version of the predictor.

The implementation results of the **prediction checking** algorithms (FPGA architecture) establish the importance of including memory resources in the optimisation process. Additionally, performance of the design can be improved by loop unrolling and utilising Dual-Port RAM.

I found a clear relationship between the amount of predictions and the number of clock cycles required to execute prediction checking in hardware. This is an important realisation as it allows for simple tuning of the heterogeneous model. The fundamental advantage lies in the predictable effect on performance.

Upon evaluating the heterogeneous model, I found the performance of the synchronous approach to scale poorly with increased market size, directly impacting the frequency at which a full re-computation can be calculated. However, I noted that performance can be improved when utilising an asynchronous model, but this comes at a cost of re-using the predictions over multiple market ticks, which can reduce their effectiveness.

Fulfilling the main objective of this thesis, I showed that the heterogeneous system provides a considerable reduction in latency, ranging from 83% to 90% for market sizes of 256 and 128. This was achieved not only by the introduction of the modelling and on-line components, but also by implementing the prediction checking on the FPGA.

One of the observations made during the project is the increased development time necessary for FPGA designs, even when the underlying source-code is relatively simple. However, I am encouraged by the prospect of the integration of AutoPilot into the Xilinx tools, which has the potential of reducing the effort required. I understand that the performance increases will only be as good as the intelligence of the optimisations carried out by AutoPilot and will in all likelihood not surpass efforts of experienced hardware engineers. Nevertheless, what must be considered is the trade-off between development time and outright speed.

Regrettably, the project is not without limitations. Chief amongst these is that the profitability of the heterogeneous model could not be established, a direct result of the characteristics of arbitrage, where only the first system to execute can profit. A further contributing factor, and hence shortcoming, is the use of randomly generated market data for the purpose of evaluation.

8.2 Future Work

The limitations of my work could be used as a starting point for further development. Hereby, I present additional approaches for extending my project:

- **Low-latency Arbitrage Model:** A possible area for further work is to either develop or integrate a financial model for low-latency arbitrage. The major focus area should be to represent the amount of time arbitrage opportunities last, before being executed by market participants. Such an approach would allow for a more thorough evaluation of the profitability of the heterogeneous model.

- **Real Market Data:** The project could be extended by incorporating real market data, adapting the algorithms to use the bid-ask model and re-running the evaluation results. Ideally, live market data would be used with an aim of benchmarking the system under representative load, as well as checking for arbitrage in the low latency scenario.

- **Increased Sample Space:** Based on the idea of adapting the proposed model to deal with different markets, the number of products (currencies) considered could be increased, or the degree of arbitrage raised to four. In order to achieve scalable performance under these conditions, alternative arbitrage detection algorithms would need to be developed. A possible approach is to limit the sample space that is considered during a re-computation of the algorithm and could be achieved by keeping track of changes in the market data.

- **Data Prediction:** There are two immediate ways of contributing to work done on the data prediction component. The most straightforward approach is to modify the "prediction sample" sizes and evaluate how this effects the relative performance of the mean and exponentially weighted predictors. A problem posing greater challenge, would be to develop more advanced data prediction algorithms, focused on finding patterns and correlation in the market data, that lead to the formation of arbitrage opportunities.

- **GPU Acceleration:** The topic of GPU acceleration has only briefly been discussed in my thesis. Although initial results for the arbitrage detection algorithm were disappointing, further investigation could lead to a GPU implementation of the data prediction component. Alternatively, an existing prediction solution could be adapted in order to integrate with the heterogeneous model.

- **Hardware Implementation:** A logical extension would be to run the generated designs on physical FPGA devices. As already mentioned, the Axel Cluster [30] could be used for testing. Further work should be done to implement the designs on the latest FPGA devices (Virtex-7 Family [43]) and evaluate the effect on the latency gap to the reference CPU implementation.

- **Dual-Port RAM:** Further development could confirm the effects of using Dual-Port RAM on the performance of the prediction checking algorithm. Based on AutoPilot results for the clock cycle count, this should allow for a further 25% to

35% reduction in the computation time. However, the effects on clock period would also need to be determined in order to establish the precise performance.

- **FPGA Optimisation:** Additional work could be carried out on the FPGA implementation, specifically dealing with the "Two Best Predictions" approach, where we observed decreased performance (by a factor of two) and considerably longer times for the designs to "Place and Route". The approach of pipelining the design should be investigated, with the aim of narrowing the performance gap to the "Single Best Prediction".

- **Medicine:** More challenging further developments could look at ways of adapting the heterogeneous model to applications outside the field of Finance. It is feasible to implement any system that requires the combination of modelling and reduced response time. A possible application scenario is the use of haptic devices in medical surgery, where the modelling component might predict and govern the surface interactions. The on-line aspect could be realised by providing quick response time to the surgical instrument.

These are just some of the issues that could be explored when expanding on the subject of the heterogeneous model. As the scope of this thesis is relatively broad, there could be numerous additional opportunities for future work.

List of Figures

List of Tables

Bibliography

[1] Yukihiro Aiba and Naomichi Hatano. A microscopic model of triangular arbitrage. *Physica A: Statistical and Theoretical Physics*, 371(2):572 – 584, 2006.

[2] Yukihiro Aiba, Naomichi Hatano, Hideki Takayasu, Kouhei Marumo, and Tokiko Shimizu. Triangular arbitrage as an interaction among foreign exchange rates. *Physica A: Statistical Mechanics and its Applications*, 310(3-4):467 – 479, 2002.

[3] Adrian Buckley. *Multinational Finance*. Prentice Hall, fourth edition, 2000.

[4] Daniel J. Fenn, Sam D. Howison, Mark McDonald, Stacy Williams, and Neil F. Johnson. The mirage of triangular arbitrage in the spot foreign exchange market. *International Journal of Theoretical and Applied Finance (IJTAF)*, 12(08):1105–1123, 2009.

[5] Bank for International Settlements. Report on global foreign exchange market activity in 2010. `http://www.bis.org/publ/rpfxf10t.pdf`, December 2010.

[6] FPGA Central. FPGA - Field Programmable Gate Array. `http://www.fpgacentral.com/pld-types/fpga-field-programmable-gate-array`, February 2008.

[7] fxTrade. Buying and Selling Currency Pairs. `http://fxtrade.oanda.com/learn/intro-to-currency-trading/conventions/currency-pairs`, January 2010.

[8] Engineering Statistics Handbook. Forecasting with single exponential smoothing. `http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc432.htm`, April 2011.

[9] Joel Hasbrouck and Gideon Saar. Low-latency trading. `http://pages.stern.nyu.edu/~jhasbrou/Research/Working%20Papers/HS10-11-10.pdf`, 2010.

[10] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing*. Morgan Kaufmann, 2008.

[11] Unix Help. Time. `http://unixhelp.ed.ac.uk/CGI/man-cgi?time`, June 2011.

[12] Intel. Intel compilers and libraries. `http://software.intel.com/en-us/articles/intel-compilers/`, June 2011.

[13] Investopedia. Arbitrage Definition. `http://www.investopedia.com/terms/a/arbitrage.asp`, January 2010.

[14] Investopedia. Bid-Ask Spread Definition. http://www.investopedia.com/terms/b/bid-askspread.asp, January 2010.

[15] Investopedia. Currency Pair Definition. http://www.investopedia.com/terms/c/currencypair.asp, January 2010.

[16] Investopedia. Statistical Arbitrage Definition. http://www.investopedia.com/terms/s/statisticalarbitrage.asp, January 2010.

[17] Qiwei Jin, D.B. Thomas, and W. Luk. Exploring reconfigurable architectures for explicit finite difference option pricing models. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 73 –78, September 2009.

[18] Kim Kendall. *Electronic and Algorithmic Trading Technology: The Complete Guide (Complete Technology Guides for Financial Services)*. Academic Press, July 2007.

[19] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010.

[20] Alexander Lipton. *Mathematical Methods For Foreign Exchanges - A Financial Engineer's Approach*. World Scientific, 2001.

[21] Linux Kernel Manual. Getrusage. http://www.kernel.org/doc/man-pages/online/pages/man2/getrusage.2.html, June 2011.

[22] GNU Project. The gnu compiler collection. http://gcc.gnu.org/, June 2011.

[23] Lati Rob. The Real Story of Trading Software Espionage. http://advancedtrading.com/algorithms/showArticle.jhtml?articleID=218401501, July 2009.

[24] Mark Salmon and Roman Kozhan. On uncertainty, market timing and the predictability of tick by tick exchange rates. Technical report, Warwick Business School, Financial Econometrics Research Centre, 2008.

[25] EE Times. Intel details Sandy Bridge at ISSCC. http://www.eetimes.com/electronics-news/4213428/Intel-details-Sandy-Bridge-at-ISSCC, February 2011.

[26] EE Times. Xilinx buys high-level synthesis eda vendor. http://www.eetimes.com/electronics-news/4212668/Xilinx-buys-high-level-synthesis-EDA-vendor, June 2011.

[27] Financial Times. Trading goes wild on wall street. http://www.ft.com/cms/s/0/0cbdcdc2-5966-11df-99ba-00144feab49a.html, May 2010.

[28] A.H.T. Tse, D.B. Thomas, and W. Luk. Accelerating quadrature methods for option valuation. In *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, pages 29 –36, April 2009.

[29] Kuen Hung Tsoi. Hitchhiker's guide to the autopilot (2011). `http://www.doc.ic.ac.uk/~khtsoi/prj_autopilot/index.html`, January 2011.

[30] Kuen Hung Tsoi and Wayne Luk. Axel: a heterogeneous cluster with FPGAs and GPUs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 115–124, New York, NY, USA, 2010. ACM.

[31] Wall Street & Technology. The High-Speed Arms Race on Wall Street Is Leading Firms to Tap High-Performance Computing. `http://www.wallstreetandtech.com/operations/198001925?pgno=2`, March 2007.

[32] Feng Wang, Yuanxiang Li, Li Liang, and Kangshun Li. Triangular arbitrage in foreign exchange rate forecasting markets. In *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 2365 –2371, June 2008.

[33] Wikipedia.org. Currency Pair. `http://en.wikipedia.org/wiki/Currency_pair`, January 2010.

[34] Wikipedia.org. Field-programmable gate array. `http://en.wikipedia.org/wiki/FPGA`, June 2010.

[35] Wikipedia.org. Foreign exchange market. `http://en.wikipedia.org/wiki/Foreign_exchange_market`, 2010.

[36] Wikipedia.org. Algorithmic trading. `http://en.wikipedia.org/wiki/Algorithmic_trading`, June 2011.

[37] Wikipedia.org. Exponential smoothing. `http://en.wikipedia.org/wiki/Exponential_smoothing`, June 2011.

[38] Wikipedia.org. Standard deviation. `http://en.wikipedia.org/wiki/Standard_deviation`, June 2011.

[39] Stephen Wray. Exploring Algorithmic Trading in Reconfigurable Hardware. Master's thesis, Imperial College London, June 2010.

[40] Stephen Wray, Wayne Luk, and Peter Pietzuch. Exploring algorithmic trading in reconfigurable hardware. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 325 –328, July 2010.

[41] XE. ISO 4217 Currency Code List. `http://www.xe.com/iso4217.php`, January 2010.

[42] Xilinx. Design flow overview. `http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0013_5.html`, June 2011.

[43] Xilinx. Virtex-7 FPGA Family. http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm, June 2011.

[44] Xilinx. Xilinx DS100 Virtex-5 Family Overview. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, June 2011.

[45] Yahoo! UK & Ireland Finance. Currency conversion. http://uk.finance.yahoo.com/, January 2010.