Department of Computing
Imperial College London

Exploring Algorithmic Trading in Reconfigurable Hardware

MEng Final Year Project

Author:
Stephen Wray
sjw06@doc.ic.ac.uk

Supervisor:
Prof. Wayne Luk
wl@doc.ic.ac.uk

Second Marker:
Dr. Peter Pietzuch
prp@doc.ic.ac.uk

June 2010

Abstract

This report describes an algorithmic trading engine based on reconfigurable hardware, derived from a software implementation. My approach exploits parallelism and reconfigurability of field-programmable gate array (FPGA) technology. FPGAs offer many benefits over software solutions, including reduction in latency for decision making and increased information processing. Both important attributes of a successful algorithmic trading engine. Experiments show that the peak performance of our hardware architecture for algorithmic trading is 377 times faster than the corresponding software implementation, with a reduction of minimum decision latency of 6.25 times. Up to six algorithms can simultaneously operate on a Xilinx Vertex 5 xc5vlx30 FPGA to further increase performance. Using partial run-time reconfiguration to provide extensive flexibility and allow reaction to changes in market conditions, tailoring performance of the system to the work load. I present analysis of the reconfiguration ability and what can be done to reduce price performance loss in algorithms while reconfiguration takes place, such as buffering information. From my results we can show that an average partial reconfiguration time of 0.002 seconds, which given the historic highest market data rates would result in a worst case scenario of 5,000 messages being missed. Additionally I will describe the technical challenges that arose in implementation of the test system and the novel solutions that I found for them. Including optimised array averaging using tree summation, abstract process design patterns and architectures to decrease development time and provide base functionality to trading algorithms.

Acknowledgements

I would like to thank Prof. Wayne Luk as the supervisor of this project for his help and support during the project. His comments and suggestions have been invaluable in increasing the quality and scope of the project. Additionally for his suggestion of submitting two research papers to international conferences based upon my work, both of which were accepted with his assistance. I would like to thank Dr Peter Pietzuch additionally as the second marker for my project. For his ideas and reviewing my work. Also for helping me become a better technical writer when producing and submitting my two conference papers and my final report.

I would like to thank Tim Todman for his help in setting up the Mentor Graphics DK Design Suite development environment. Additionally I would like to thank Dr Qiang Liu for his help acclimatising to the Mentor Graphics DK Design Suite and Xilinx ISE, both of which were used to produce the hardware implementation for this project. Both of whom are at Imperial College London.

I benefitted greatly from the excellent feedback returned by the anonymous reviewers of my two conference papers. Therefore I would like to thank them in enabling me to improve my submissions and their suggestions on how I could improve the quality and direction of my research. Finally I would like to thank my family and friends for their support and guidance throughout this project. Without this I fear that my project may not have been as successful as it has been.

Contents

Chapter 1

Introduction

The scene is a large investment bank's trading floor. You are a trader for the bank and it is your job to not only to achieve the best possible price for your clients, but also to ensure that you do this without loosing your employer money. Perhaps if your good, you'll even make some.

It is 07:55AM and the equity markets are just about to open on what could be a very busy day. A number of key blue-chip firms will be announcing their profits for last quarter and there will be some key economic data from the government on inflation. This is likely to make the day very long and you have already been at work for nearly two hours.

Clients are placing their orders already, before the market open so that you can start immediately trading them. Your trading interface lists the orders that you are managing, as more and more arrive, they start to scroll off the bottom of the screen. This is what is being displayed on only one of your six monitors that you have in front of you, all of which show continually updating information that you must digest immediately if you are to be as successful as possible.

This doesn't include the large flat panel televisions that are suspended from the ceiling all around the trading floor. Alternate televisions are showing the Consumer News and Business Channel (CNBC) and Sky News. Your colleagues next to you are in the same situation, they will also be trading in the same select few of stocks that you are. Even though these are relatively few in number, the number of orders is still increasing. Some change as their owners make amendments, others disappear as their owner decides to cancel the order, but not a lot of them.

Finally 08:00AM arrives and the markets open. Which orders do you trade first? Is the stock price rising or falling? How much of the order should I trade now and how much should wait? What is the expected outcome of the news coming later, will inflation be down or will it rise? Will those companies announcing profits be in the black or the red? What do other traders around the world think is going to happen? These are just some of the questions that you have seconds or minutes if you are lucky to answer. All the while, the information is continually changing, updating and shifting. This endless stream of work to carry out.

You make it through the day, 16:30PM arrives and the equity markets close until tomorrow morning. At which point you will have to carry out the whole process again. Before you can go home for the day, which started at 06:00AM, you need to prepare for tomorrow. What is expected tomorrow and any key factors that will need to be taken into consideration.

Before you leave, you think, "Would it not be great if there was a system which could trade some of the smaller order automatically, which could cope with the ever growing work load? Allowing me to concentrate my attention on the important orders!".

1.1 Project Inspiration

Thankfully for our anonymous trader, there are such systems that are readily deployed and used by large investment banks and other equity trading companies. They arose from specifically the need to be able to cater to the vast increases in electronic trading flow that have been seen over the last decade because of increased use of electronic trading. Unfortunately though, it is becoming increasingly more difficult to be able to maintain system development to the pace of the increase in activity. Amplifying the need

for ever increasing performance is that ever microsecond saved could result in hundreds of millions of pounds profit for the owner. It is therefore within the interests of the operator to possess the fastest and most capable system compared to their competitors. This is where applying reconfigurable hardware to this problem may result in considerable gains, enabling this to be the case. This project's goal is to look at the feasibility of designing and implementing an algorithmic trading engine with its associated trading algorithms in reconfigurable hardware. Comparing and contrasting the implementation against an equivalent software version.

These problems have been receiving increasing media attention recently as institutions look for possible solutions to their computational and latency bottlenecks. These include articles in The Times [1] and Information Week [2] which look at and discuss many of these problems and overview possible directions of solutions. Specifically reconfigurable hardware has been looked at in reasonable depth directly by Automated Trader [3], including why there hadn't previously been much market penetration by the technology. Those barriers are starting to be lifted now with the advent of more mature development tools and high-level programming languages supporting reconfigurable hardware. More recently, it is known that a some large and respected organisations have been utilising the power of reconfigurable hardware, specifically a joint venture between Credit Suisse and Celoxica [4]. This strengthens the argument that the time is right for further research into related areas of the electronic trading flow.

Why does reconfigurable hardware offer a performance benefit over using a conventional software implementation? Well, reconfigurable hardware doesn't have any overheads such as the operating system, this means that all available performance can be utilised effectively. Additionally, reconfigurable hardware has a natively massively parallel architecture, where software has only just recently started become more dependent on being executed in parallel to maintain performance. Finally, logic executing in reconfigurable hardware has a standardised and numerically proven performance. This is partly due to the lack of an operating system or any other processes, but also dependent on that you can physically study the implementation and deduce a very accurate expected performance from it. This performance won't fluctuate over time, resulting in a highly desirable, extremely low standard deviation on performance.

Implementing anything in reconfigurable hardware is still challenging in itself, even for particularly simple problems. This is because it requires learning additional technologies and techniques, as well as a different mind set when programming the device. This provides difficultly to such a project, especially in light of the problem space, which itself is large and complex, even for software. Another difficulty is faced in this project is the lack of research and resources about the area. The nature of electronic trading is that it is very secretive, any performance benefit discovered by an institution would be a closely guarded secret, in hope that competitors would also have the same ideas. This leaves this project with more work on setting out the foundations and basis of the work then many of projects which have large amounts of material available on them.

This collection of problems and challenges provides the prospect for an exciting and importantly very relevant project. The work carried out here carries very significant real world benefits based on the outcome of its success. These not only include a faster system now, but a means to being able to cope with the increased work load of future years.

1.2 Why Algorithmic Trading?

Why algorithmic trading? I choose algorithmic trading specifically because although still as undocumented as many similar problems within the electronic trading flow, it has some simpler concepts which with some background can be understood by the reader. It also forms a good base for further work as more complex scenarios and applications that rely on heavier use of business principles also utilise underlying order management and algorithmic trading solutions. Algorithmic trading can also be looked under the guise of being a *complex event processor* (CEP), this area has had significantly more research published. Although this research is concerned with processing different information, many of the principles can be transferred in to this problem space, aiding the project.

Isn't algorithmic trading evil? Of course many of you may question doing research into these kinds of activities given the current economic climate. The world has just been through its worst recession in more than 70 years and the blame has mainly been placed at the feet of the worlds investment banks. The recession even saw the bankruptcy of one of the worlds largest and successful investment banks, Lehman Brothers [5]. Although the recession had a number of factors, it does not seem that algorithmic

trading contributed in anyway, however this doesn't exonerate it from bad press. More recently, on the 6th May 2010 the Dow Jones Industrial average fell by near 10% in the matter of minutes before rebounding to being only 3-4% down [6]. This was one of the biggest and fastest crashes seen in the history of the equity markets and even though the market recovered some what is still extremely worrying. The blame for this crash has been aimed at algorithmic trading, in particular high frequency trading. Some estimates now place algorithmic or high frequency trading as being nearly 66% of all market volume and increasing daily. At the end of the article [6], the speed of growth in these areas is state for emerging markets. Emerging markets are those that are just beginning to see more activity and attentions and the four biggest are noted under the BRICs umbrella (Brazil, Russia, India and China) [7]. Currently algorithmic trading on the Indian stock exchanges is less than 5%, but growth is expected to be 30% to 40% in 4 to 5 years [6]. This shows the importance of these technologies and this research, as well as amplifying the problem of handling increased work load. Many people automatically brand something as evil if they don't understand it. I am not trying to say that algorithmic trading is completely safe and should be unregulated. However with any new technologies there are teething problems and given the environment of this work, much more work is carried out on validation then many other computing projects. It is also a necessity for systems like this to exist, as without investment banks and equity markets, then the companies that we know and love wouldn't be able to exist, producing the products that we enjoy.

Why reconfigurable hardware? There are other interesting technologies gain ground within the industry that offer increased performance over using a conventional *central processing unit* (CPU). *General Purpose Graphics Processing Units* (GPGPUs) are one such technology which offers significant performance in float-point operations. While this would be another interesting avenue of research, GPGPU doesn't align as well with the algorithmic trading problem space. While float-point operations are useful, they are not critical to trading algorithms with some calculations being integer based. Additionally, given current social pressures on companies and industries to become more environmentally friendly, reconfigurable computing offers better performance over power consumption than both CPU and GPGPU solutions. Finally, GPGPUs are still limited to being used as co-processors, you cannot currently deploy a whole system on a GPGPU which is independent from the host system. For example, the GPGPU solutions would still be reliant on the host machine providing network connections and data transfer. Many reconfigurable computing boards are now supplied with network controllers that allow them to be directly plugged in to the network and consume information from the network interface. This aligns more closely with the long-term goals of this project, of moving the complete algorithmic trading engine into reconfigurable hardware.

1.3 Objectives and Contributions

This reports serves to show case the important work and contributions that I have made throughout this project. I have broken the rest of the report into a number of chapters, each specialising in an area of the project which is critical to its success. I will cover the important design decisions that I have made throughout the implementation, discussing any problems that were encountered and highlighting my novel solutions. Specifically this report will cover:

- **Presenting all of the information and background required for this project as well as analysis of existing research and projects.** I do not expect the reader to have a strong grounding in either the business knowledge or reconfigurable hardware. For this reason I have included a background on the project which will cover these topics as well as looking at the current state of the art for research in related fields. After reading, the reader will have sufficient knowledge to comprehend the rest of this report; in chapter 2.
- **Presenting a framework designed to facilitate the creation of a wide range of electronic trading systems with improved compatibility and flexability.** I will present my *electronic trading framework* which provides an abstract base for modelling different components of an electronic trading system. This has been designed with flexibility in mind and would provide an excellent base for further research in the equity trading field. This is shown through the careful use of software engineering and design principles; in chapter 3.

- **Presenting a method of building electronic trading systems from abstract representations of their activity.** Managing the trading algorithms is a trading engine built on the framework. This provides an extensible platform for quickly deploying different algorithms implemented in different technologies through its connection interface. This provides unparalleled flexibility for the operators and a platform capable of performing research on a wide variety of models and algorithms; in chapter 4.

- **Presenting a design pattern for creating trading algorithms which could be used to increase the number of algorithm implementations supported.** With the underlying system complete, I know present my design pattern for creating high performance and flexible algorithms with shared functionality designed to increase their price effectiveness. Built on top of this process I present three concrete implementations of industry accepted algorithms which assess market favourability individually; in chapter 5.

- **Presenting a solution to implementing trading algorithms in hardware using an abstract base algorithm to speed up design. Using partial run-time reconfiguration to further increasing performance and flexibility of the system.** These trading algorithms specifications where then lifted and transformed into hardware, providing the same functionality to allow for a direct comparison against their software counterparts. The hardware algorithms gain the of the reconfigurable hardware and I will show my solutions to mitigating some of the limitations imposed, such as batch processing; in chapter 6.

- **Presenting a method and design of testing for electronic trading systems with event data generated in real-time.** To properly evaluate the effectiveness of the implementation, a realistic testing harness was required which could supply an unlimited number of pseudo events to which the system could react. I propose such a method of generating these events in a statistically correct manner while maintaining randomness and plausibility, preventing the system from entering cycles; in chapter 7.

- **Presenting two technology independent analysis techniques, results from testing and analysis of the work in the project.** The implementations can now be tested using the testing harness, allowing us to evaluate their performance against each other. I'll look at a number of different scenarios including throughput performance, partial run-time reconfiguration and power consumption of the devices. This will allow us to evaluate the success of the project and provide motivation for further work; in chapter 8.

- **Presenting my conclusions draw from the results presented in chapter 8 and what they mean for this project.** I will wrap up this project and report with a discussion on its successes and failures. This will provide space for me to discuss changes that I would have made to the project given time. This leads to exciting areas of further work that completion of this project has enabled, such as high frequency trading and statistical arbitrage; in chapter 9.

1.4 Published Work

During this project I was fortunate enough to be able to complete a conference paper on some of the project's interim work for the ASAP 2010 conference (the 21st IEEE International Conference on Application-specific Systems, Architectures and Processors) with the help of Prof. Wayne Luk and Dr. Peter Pietzuch [8]. The paper was accepted as a short paper (poster) for the conference and covered the hardware design and initial performance studies.

Additionally, I was also able to complete a second paper while working on the project, again with the help of Prof. Wayne Luk and Dr Peter Pietzuch [9]. This paper was submitted to the FPL 2010 conference (the 20th International Conference on Field Programmable Logic and Applications) and was accepted as a short paper (poster). The paper covered using runtime-reconfiguration of FPGAs to allow the algorithms of my implementation to be changed during the trading day. The paper was concerned with the performance and other issues around using partial run-time reconfiguration.

Chapter 2

Background

The aim of this project is to explore feasible methods of accelerating an Electronic Trading System (ETS) specifically an Algorithmic Trading Engine with Field Programmable Gate Arrays (FPGAs). This project deals with equity trading and the equity trade flow, from clients to exchanges. Therefore, a good understanding of these topics is required to comprehend the work in this project. Additionally a good grounding in computer and software engineering principles is required for the more technical aspects that arise from this work.

The aim of this section is to provide the background needed to understand this project, it will cover both the business knowledge about equity trading, as well as about working with FPGAs. It will also cover the current state-of-the-art, showcasing current work and research in this area and how it relates to this project. In detail, this chapter includes the following topics:

- The concepts behind stock, shares and how they are traded. This is the foundations to the business knowledge required in this project; in section 2.1.
- An introduction to electronic trading, its motivation and the order trade flow. I will set out how equity trading has moved from a manual activity to being automated by electronic systems and the problems that arose with this; in section 2.2.
- Explanation of what algorithmic trading accomplishes and its methods. This will provide scope to the project and why algorithmic trading is important to todays equity landscape; in section 2.3.
- How to choose when to trade, an introduction to Volume Weighted Average Price (VWAP), Time Weighted Average Price (TWAP) and Participation; in section 2.4.
- I will discuss my reasoning for the choice of programming language and host operating system that will be used throughout this project with their effects on the project; in section 2.5.
- Overview of reconfigurable hardware, current FPGA computing platforms and those that are available for this project; in section 2.6.
- Comparison to existing studies into applications of FPGAs in financial applications and other relevant problem spaces. Additionally, the current state-of-the-art in electronic trading; in section 2.7.
- Summary of the available background, the concepts introduced and how they relate to what this project's aims; in section 2.8.

2.1 The Concepts of Equity Trading

Equity trading is the exchange of *shares* or *derivatives* built on *stock* of a particular business. Stock is a number of shares in that business and shares represent the original amount of capital invested in the business, the owners of the stock are therefore *investors* of the business. Stock serves as a security to creditors of the business as it cannot be withdrawn to the detriment of the owners. At the time of creation, the total value of shares is the total initial investment in the company, the price of a share at

Figure 2.1: An example of Vodafone's share price fluctuating over a number of trading days [12].

this time may be know as its *par-value*. There may be different types of shares in a business which may vary certain attributes, such as there their ownership rights and preference status for example. If the company makes a profit then it may issue all, some or none of the profit as a *dividend* to its shareholders, this would be equally divided over the existing shares although different types of shares may receive a greater percentage than other types [10].

This project focuses on native shares or stock when trading, ignoring any derivatives such as *futures* or *options*. As stock in a company has an intrinsic value, it is classed as an *asset* and so they can be sold and bought to or from other entities. If selling, the owner must have permission to do so under the terms of ownership of the shares. To enable free and easy trading of stock in companies, *stock exchanges* were created. These are designed to match buyers and sellers together to trade after agreeing a suitable price. For shares in a particular company to be traded on an exchange, they must be listed at that exchange. For a company to be listed it must meet a set of certain criteria set out by the exchange. Smaller companies that don't meet the criteria can still be traded but this is normally done *over-the-counter*, where the trade is completed directly between the two parties. Orders are normally sent to the exchange by *stock brokers* who work on behalf of individuals or entities wishing to trade shares. Historically, this was because you had to be physically at the exchange to trade, but now with electronic trading, it is more dependent on the cost of joining an exchange. Stock brokers arrange for the transfer of the stock from one investor to another after a trade is completed and charge a small fee for the work, normally a percentage value of the trade [11].

As with any product in limited supply that is traded, share prices fluctuate over time depending on supply and demand. The amount of shares available is referred to as liquidity. Supply and demand depends on how well the company in question is performing. A company that is thriving is likely to have a dividend for its *shareholders*, making the company more desirable to invest in and the share price go up. If the company is struggling however, there is unlikely to be any dividend and people won't want to invest in the company, making the share price go down. The share price of a company is communicated through a *market ticker* which updates every time there is a price change, this information is transmitted to all interested parties. Figure 2.1 shows an example of how the share price of a company can vary over time, in this example the company is Vodafone who listed on the London Stock Exchange. The figure also shows a lot of information such as the number of shares (526.08 million), the day's price range (143.15 - 144.40) and the 52 week high and low (111.20 - 148.00). Prices are in Pounds Sterling (£).

Investors want to make money on their investments, this can be done through receiving the dividend on shares owned. Alternatively, buying shares when they are cheap and selling them when they are more expensive, taking the difference as profit, minus any brokerage costs. Table 2.1 describes in detail the two methods of achieving a profit. If however the market doesn't move in the desired manner then the investor stands to make a loss, how much of a loss is dependent on how long the investor stays in that position. If the investor didn't take any *leverage* (*credit*) when buying the shares though, it is impossible for them to lose more money than the investment they made, as shares have an absolute minimum value of zero. How investments are chosen is outside the scope of this project.

Trading Style Name	Summary
Long	Buying shares cheaply and then selling when they are more expensive is the obvious strategy. The investor believes that the share price will go up and this is referred to as going *long*.
Short	An investor can borrow the shares for a fee from an owner, sell them when they are expensive and wait for the price to drop before re-purchasing the shares, returning them to the owner. This is referred to as *short* selling and the investor hopes that the company's share price will decrease.

Table 2.1: Description of the two different trading methods.

2.2 Introduction to Electronic Trading

Electronic trading removed the need for there to be a physical trade floor, allowing trades to be completed in a virtual environment. This was enabled by the increase in computing power and communications technology in the early 1970s. Exchange servers would maintain lists of buy and sell orders, ordered by their price, finding and executing trades automatically [13]. Stock brokers would connect directly into the electronic market place, passing orders on from clients. Those orders can be telephoned in by an investor but increasingly they are also placed electronically by the client at their computer.

In 2007, 60% to 70% of the volume traded on the New York Stock Exchange (NYSE) was electronically traded [2]. In 1961, the average daily traded volume on the NYSE was four million, however by January 2001, that daily volume was already topping two billion trades a day [14]. The increase in electronic trading had a large number of effects on how the equity markets which are detailed in table 2.2.

On the 1st November 2007, the European Union's *Markets in Financial Instruments Directive* (MiFID) [15] came into force. This had a large effect on equity trading within Europe, making it more like that of North America. The most notable change was now the ability for third-parties separate from central exchanges to create venues where trading of stock was also allowed. This broke up the monopoly that had been held by central exchanges such as the London Stock Exchange (LSE) [16] since their creation, these new venues are called *Multi-Lateral Trading Facilities* (MTFs). Already a number have

Effect	Summary
Decreased Latency	The amount of time it took an investors order to reach the market was heavily reduced. Originally taking minutes by phone call, orders placed electronically could be at the market in under a second.
Reduced Costs	The cost of placing orders dropped sharply as it was significantly cheaper to automate the trading process with computers than employ people to do the work.
Increased Participation	As the price lowered more investors could afford to start trading, increasing the amount of competition in the market place.
Increased Transparency	The market became much more transparent as relaying market data to investors was made cheap by the increased use of computer networks and the Internet.
Increased Activity	The volume of trades increased heavily as more people started trading and cheaper fees allowed investors to trade more frequently.
Increased Efficiency	As the volume and number of investors increased, the amount of liquidity available also increased, allowing the markets to be more efficient.
Decreased Price Spread	With more volume the spread (or difference) between the bid and ask prices shrank because of increased competition and visibility.

Table 2.2: Details of the effects on the equity markets with the increased use of electronic trading.

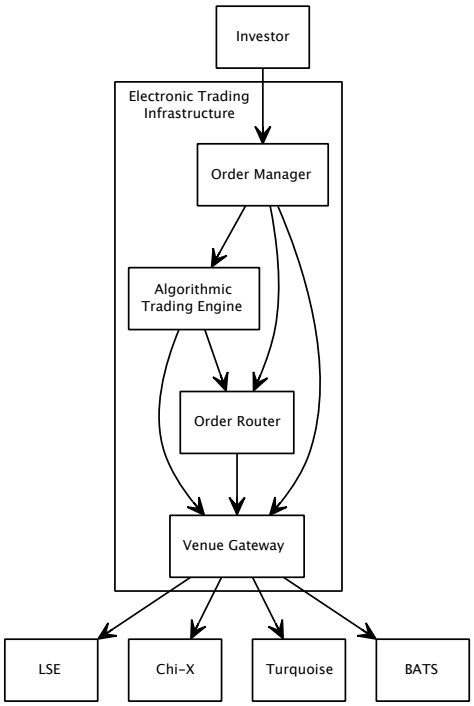


Figure 2.2: Trade flow from investor to exchanges in Europe via the electronic trading infrastructure.

been created in Europe, on London shares, 40% are traded at alternative venues to the LSE [17], such as Chi-X [18].

With more venues available, the liquidity that was once in a single place is now fragmented and requires new techniques to access it. As with all electronic trading there is an automated method which many brokers use that looks at current and historical market data from these venues and decides where to send the order or fractions of the order, in an attempt at getting the fastest and best priced executions. Different venues charge different exchange fees so no longer is the best price what the execution of the trade is at, but also includes the cost of going to that market. To ensure that no bias is held between the different venues and the best possible price is achieved by the brokers, brokers must adhere to the *Best Execution Policy* (BEP).

Part of the reason why volumes in trades have increase so significantly over the last couple of years is because of the number of shares being traded has risen. Investors may want to sell or buy large numbers of shares, but this cannot be done in a single order as it will be against market regulations as it will move the market. A large order will also make it very clear to other investors about what you are doing, which is undesirable in the secretive world of investment strategies. One solution to these problems that arose was venues offered *Dark Pools of Liquidity*, where orders wouldn't be published on market data, preventing anyone from knowing what was available. The second strategy is to work the order in an *automated* or *algorithmic* manner over a period of time, this became *Algorithmic Trading*.

Figure 2.2 shows the flow of an order from the investor through the *electronic trading infrastructure* to the exchanges and *alternative venues*. All orders enter at the *order manager*, which can then go onto automated systems to trade the order or directly to market. The *algorithmic trading engine* can use the *order router* if it wishes or like the order manager, go straight to market. The *venue gateway* just provides a proxy to the venues and performs and conversions between data representations for the specific venue. The majority of electronic trading systems are implemented in C++, running in Unix-like environments on high performance 64-bit servers. This allows them to cope with the high throughput they must handle, while being low latency, have a low standard deviation on event processing and remaining reliable.

2.3 Explanation of Algorithmic Trading

Algorithmic trading was designed to increase the capacity of the number of trades handled by a stock broker. This was to enable the broker to keep up with the number of orders clients were sending. In addition, algorithmic trading provides a more standard platform on return than a human doing the same job, this consistency reduces the risk of the broker making a loss on buying or selling the shares. Finally algorithmic trading allows large orders to be traded more evenly over the trading day, minimising moving the market and being able to achieve a better price [19].

Algorithmic trading engines are similar to complex event processors, they take information in about market conditions normally in the form of market data. They monitor this information and when certain criteria are met they form a *child order*, which is a fraction of its *parent order* which is then sent to market to trade. The attributes of the child order are dependent on the parent order and the algorithm that is trading it, obviously the quantity of the child order must be less than the parent order, but other attributes such as price or order type may differ. There are algorithms that decide which stocks to buy and sell based on mathematical analysis of market conditions, commonly know as *High Frequency Trading* (HFT) or *Statistical Arbitrage* trading methods. These methods are out of scope of this project however and we will only be concerned with parent orders being received from clients.

Orders being algorithmically traded are normally traded over a longer period of time, but within the trading day. Some algorithms will wait a set amount of time to trade, where others will wait for the price of the shares to become more attractive. More intelligent algorithms may be allowed some flexibility on how they decide this, allowing them to run faster in favourable market conditions and slower when the market is against them [20]. Further intelligence can be added by increased processing of market data and attempting to predict how the market is going to move by the current activity. Having good trading algorithms is just as important as having competitive fees when trying to attract clients. Good algorithms may be able to achieve a better price on trades by a client and even with slightly higher brokerage fees the overall cost may be less than using another competitor. It is also for this reason that many of the complex algorithms used by brokers are very closely guarded secrets.

Algorithmic trading engines are often very flexible and offer the ability to quickly develop and deploy new algorithms. This is because an idea for a new algorithm may only be suited to be used for a couple of hours, but might return a sizeable profit and the ability to create the algorithm quickly is the deciding factor of whether it will be possible to make use of those market conditions. Also very large clients may request a special algorithm or feature to be added to an existing algorithm for their use only, as these very large clients may provide a large amount of revenue for the broker, being able to accommodate these requests is very desirable.

2.4 Choosing When to Trade

Three simple algorithms will be looked at in this project, with scope for looking at more advanced algorithms later provided the ground work has been completed. The first and simplest is called Participation, this algorithm attempts to trade an order over a period of time and during that time it will represent a specified percentage of all *market volume* (*quantity*) in that symbol.

$$\text{Market Volume} \quad = \quad \sum_{t=0}^{n} V_t \tag{2.1}$$

Market Volume is calculated by the sum of the volume of all trades of a stock on a market, this is shown in equation (2.1). Where V_t is the volume at time t and the sum is over all values. The algorithm will work to keep the amount of the order traded equal to the specified percentage of the market volume [21], this is illustrated by figure 2.3. Therefore the amount of time that the order takes to complete depends on how much activity there is on that stock, if the stock is quite illiquid and the order large then there is a real possibility that the order will not complete within the trading day. Participation uses the average price of shares as a price benchmark to be compared against, equation (2.2) shows how this is calculated. Where P_t is the price at time t and n is the total number of points.

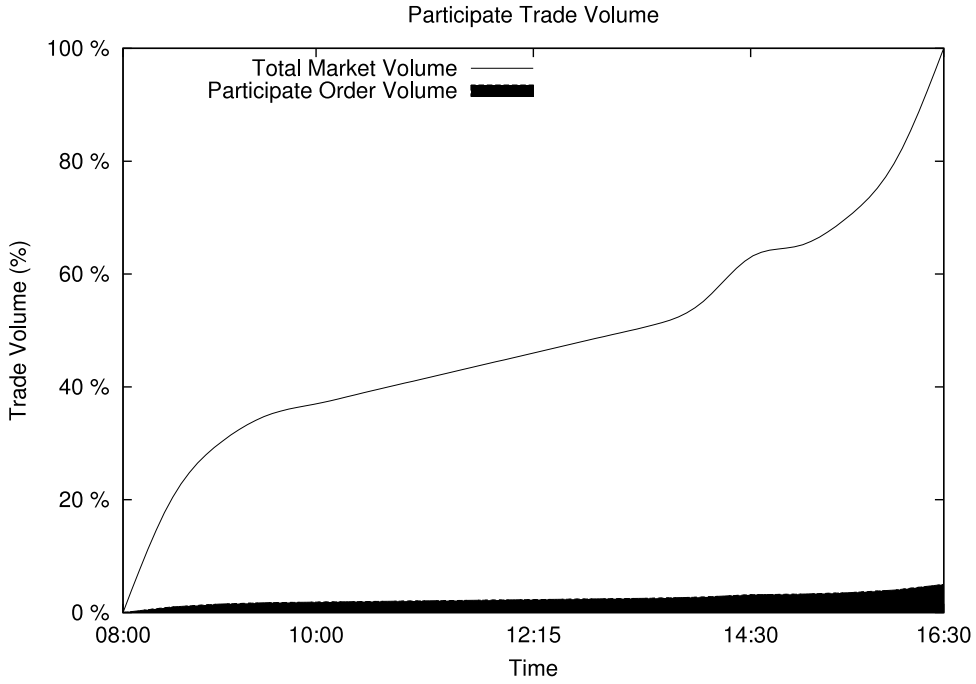


Figure 2.3: A Participate trade of 5% compared against total market volume over the trading day.

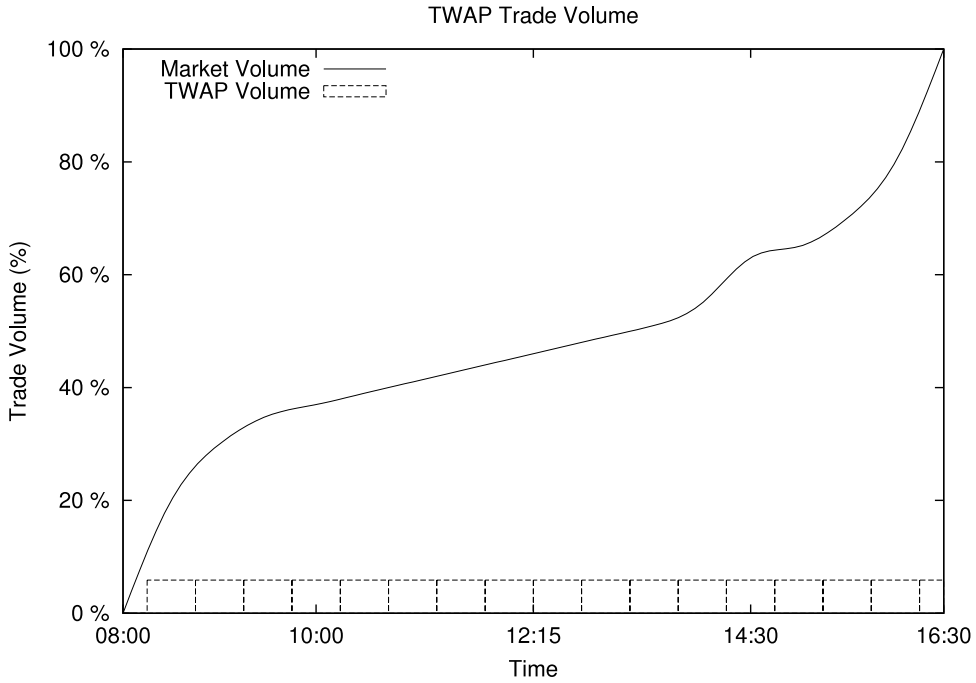


Figure 2.4: A TWAP trade child order volumes and distribution over the trading day.

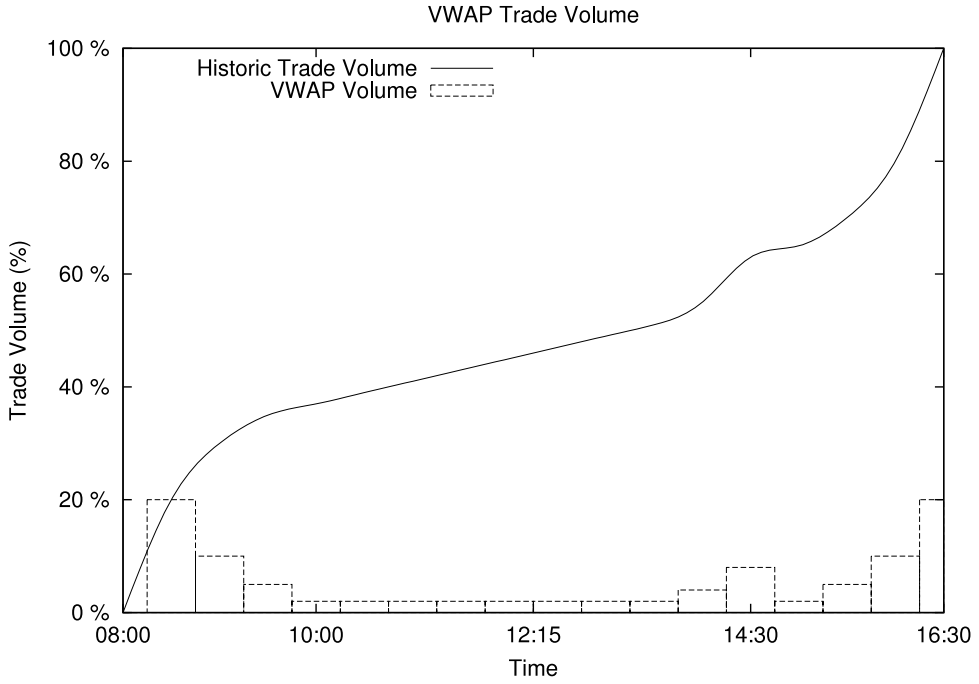


Figure 2.5: A VWAP trade child order volumes and distribution over the trading day.

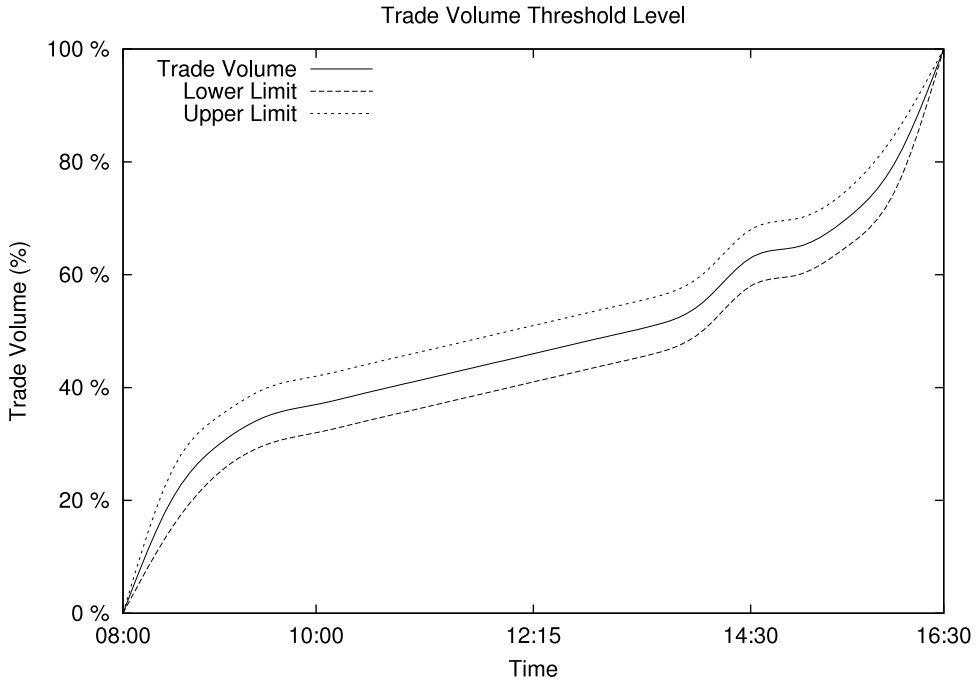


Figure 2.6: The upper and lower boundary thresholds of a trade over the trading day.

$$\text{AP} = \frac{\sum_{t=0}^{n} P_t}{n} \tag{2.2}$$

To prevent the algorithm sending large quantities of very small orders to the exchange, a minimum tranche size is set which specifies the minimum size of an order going to market. Only when enough market activity has occurred will another child order be created.

The second algorithm adheres to the *Time Weighted Average Price* (TWAP) benchmark, which is calculated by the summation of the price of a stock multiplied by the time it spent at that price, divided by the total time, as shown in equation (2.3) [22]. Where T_n is the total time, T_t is the time at which the P_t was available and T_{t-1} is the previous time period. The TWAP algorithm divides an order up into equal sections which are traded evenly over the allotted time, aiming to match the TWAP benchmark. Figure 2.4 shows a TWAP order, where child order size is in equal parts of the total order side, traded over the whole of the day, it is completely independent of market volume. As the order is traded over a specified time it can be guaranteed to complete by the end of the trading day if the end time is before the close of market.

$$\text{TWAP} = \frac{\sum_{t=0}^{n} P_t(T_t - T_{t-1})}{T_n} \tag{2.3}$$

Finally, the third algorithm trades an order against the *Volume Weighted Average Price* (VWAP) benchmark, which is shown by equation (2.4) [23]. VWAP takes into account the volume of each trade at its price of a stock over the total amount traded. Again like all algorithms it splits the parent order into multiple child orders, however a VWAP algorithm varies the quantity of each child order proportionally to the historic average volume traded at that time. The historic average is normally taken over the last 28 trading days. Figure 2.5 shows a VWAP order, where the historical average has been plotted with the orders traded volume over time.

$$\text{VWAP} = \frac{\sum_{t=0}^{n} P_t V_t}{\sum_{t=0}^{n} V_t} \tag{2.4}$$

In addition, algorithms trading styles can be specified to be passive, aggressive or a mixture. Being passive implies that all orders sent to market won't execute immediately as they won't cross the bid ask spread and therefore will take time to execute. However being aggressive means crossing the bid ask spread, this may make the trade more expensive but will ensure that the trade occurs quickly.

As well as monitoring market data to calculate the benchmarks, the algorithms can take into account what liquidity is available on the market. If the price is attractive the algorithm may decide to speed up by sending slightly larger order or another small order, conversely if the price is unattractive it can slow down. The amount the algorithm is allowed to speed up and slow down can be parameterised by a threshold. This can enable the algorithm to improve the final price of the order and react to market conditions more intelligently. Figure 2.6 shows a 5% upper and lower threshold, the quantity that the order has traded at any single point must be within those two limits.

2.5 Programming Language and Operating System Selection

C++ is the selected programming language of the software components of this project. The biggest driving factor behind this decision is that in industry this is the programming language of choice, therefore it makes a logical choice to use now to improve possibilities of system integration with an existing electronic trading networks. The reason why C++ is chosen is because it offers all of the same functionality as many popular high-level programming language but in addition has no *garbage collector*. Some may see this as a downside, resulting in more work and headaches for the programmer, but in this environment where every microsecond counts it is very important. Nearly all garbage collectors require the main application to be halted so that they can perform their duties, at a critical time this could result in an unnecessary delay to a new order being sent to market and the loss of money. Java programmers have

the ability to suggest when the garbage collector should run, but cannot force it to execute at a specified time, but even if they could they would still end up with problem of the application being paused for a short period of time. C++'s memory maintenance happens as it is required, spreading it out over time and minimising the effects of freeing unused memory. This means that every time a C++ application execute, if the inputs are exactly the same then the execution path will be as well. The same cannot be said for Java. C++ has other benefits, such as being able to work with a number of other programming languages, making using legacy code and libraries simpler, Python is another language which is very good at this.

Unix like operating systems are the chosen platform for development and execution of electronic trading system. Again this mirrors what is seen in industry, maintain the close links with existing platforms. Some of the reasons why unix-like operating systems are chosen over their *Windows* equivalents is because of their cost benefit and their openness, allowing for greater performance tuning and control over the system. However, the main attractive is accurate and precise measurement of time. Windows based systems have difficultly measuring any time below the millisecond threshold (10^{-3} seconds). When measuring performance of the systems in this project we will be interested in events of a microsecond scale (10^{-6} seconds). This will allow for performance results to be more accurate and individual events studied and compared against equivalent events to determine any performance differences.

2.6 Overview of Reconfigurable Hardware

I will assume the reader has a good background in general computing but as reconfigurable hardware is a specialist subject, I will take this opportunity to provide some background into this area of computing.

2.6.1 Reconfigurable Computing

Reconfigurable computing allows developers to make use of the high performance of a hardware implementation, while still maintaining the ease of development that software affords. This ability is provided by the hardware being able to intricately change it's data path to suite its application. The most common reconfigurable architecture used today is that of *Field Programmable Gate Arrays* (FGPAs), these are made up of a large number of logic blocks which can be connected together to perform different functions [24]. The alternative to using a reconfigurable architecture, but still retaining the ability to produce a hardware implementation is to use *Application Specific Integrated Circuits* (ASICs). However producing ASICs is still very expensive and so for limited production runs or prototyping it is often uneconomical to do so. Another advantage of using reconfigurable architectures is that the hardware device can be repurposed quickly and simply to test another design or even application. This speed and ability to reuse the hardware has made reconfigurable architecture very successful.

Reconfigurable computing originated in 1960 with Gerald Estrin's paper titled *Organization of Computer Systems - The Fixed Plus Variable Structure Computer* [25] [26]. However it wasn't until 1991 when the first commercial application of reconfigurable computing was available, produced by Algotronix [27]. Applications of reconfigurable computing are a relatively new area in both industry and research.

2.6.2 FPGA Hardware

The two main suppliers of FPGA hardware are Xilinx [29] and Altera [30], both provide the physical chips and integration boards allowing the hardware to be connected up to a computer to be used. FPGAs consist of programmable logic blocks, input, output and internal routing channels. The routing channels connect the logic blocks together and to the input output channels. Logic blocks can perform a large number of operations, ranging from simple operations such as AND and OR, to the complex such as decoders of mathematical operators. Memory is provided by both individual flip-flops and complete blocks of dedicated memory.

To programme a FPGA device, the developer must supply a *Hardware Description Language* (HDL) of the behaviour, one example of a HDL is VHDL which will be covered later with more information on programming for FPGAs. The HDL is converted to a *technology-mapped netlist* which describes the connectivity of the electronic circuit. Using a technique know as *place-and-route*, which is normally implemented by a proprietary application from the manufacturer of the FPGA, the netlist is fitted to

Figure 2.7: One of the ADM-XRC-5T2 FPGA cards used within the Axel Cluster [28].

the FPGA. This map must then be validated by the developer using various verification methods before it is accepted and a binary file created to configure or reconfigure the device [31].

Xilinx's current premier product is called the Virtex-6 and Altera's is the Stratix IV, both are built on 40nm circuits which improves both power efficiency and performance. FPGAs can be arranged in different ways with computing hardware to have additional benefits, the simplest as an external stand-alone processing unit to a co-processor or even the processor embedded in the reconfigurable architecture [32]. Many FPGA boards have additional internal connections and external methods of connection, such as network interfaces allowing computation to place from data received directly from another system.

Possible available hardware for this project includes the Axel Heterogenous Cluster [33] which contains processors, *General Purpose Graphics Processing Units* (GPGPUs) and FPGAs. Alternatively the Cube project [34] offers another platform with a large number of FPGAs available. Both are very new and experimental projects with interesting features. The cube project uses 512 Xilinx Spartan-3 XC3S4000-5-FG676 FPGAs, each offering 27,648 slices [35], the Axel cluster makes use of the Xilinx Virtex-5 LX330T-FFG1738-1 FPGAs, 51,840 slices [36]. Slices make up the FPGA's Configurable Logic Blocks (CLBs) and so can represent the size of the FPGA.

2.6.3 VHDL

VHSIC Hardware Definition Language (VHDL) where VHSIC stands for *Very High Speed Integrated Circuit*, was originally developer at the Department of Defence in the United States of America. Its purpose was to document how the ASICs the received behaved and designed to replace the large manuals of implementation specific component instructions.

VHDL can be simulated in a logic simulator that supports VHDL files and synthesised into physical implementation details of a circuit. Synthesis can have different priorities, such as creating the smallest or more power efficient circuits. VHDL is very similar to the Ada programming language and has native support for parallelism which features heavily in hardware design.

2.6.4 Handel-C and ROCCC

Handel-C is another HDL and is a large subset of the C programming language with extensions to allow manipulation of hardware specific attributes. The technology was developed in 1996 at Oxford University and has been very popular in academic hardware research [37].

Alternatively there is the Riverside Optimizing Compiler for Configurable Computing (ROCCC) [38] which is an open source C to VHDL compiler for programming FPGAs. ROCCC provides some interesting features, such as a modular bottom up design, allowing for extensive code reuse and which works by creating modules in C, compiling them to VHDL and allowing other C or VHDL modules to make use of existing modules. Addition a *Platform Interface Abstraction* (PIA) provides the ability to decouple the implementation from a particular computation platform.

2.6.5 Hyperstreams and Data Stream Manager

Building on Handel-C, Hyperstreams provides a higher level of abstraction when programming for FPGAs and has been used to calculate the price of European options [39]. At compile time, Hyperstreams automatically optimises operator latency, producing a fully *pipelined* hardware solution. Hyperstreams helps to speed up the development time when implementing complex algorithms for FPGA chips.

Data Stream Manager (DSM) enables the software hardware communication, allowing a software program to interact with a hardware implementation. A DSM stream or channel provides high throughput unidirectional communication which is independent of implementation, hardware or software. Alternatively communication can be performed with the FPGA from the CPU via Direct Memory Access (DMA) such as in project Axel [33].

2.7 Comparision to Existing Studies

There is little available material on electronic trading applications implemented in hardware or reconfigurable architectures. This is partly because it has not previously been a big area of academic research and also the nature of the industry, advancements in the field tend to carry a large competitive edge and so go unpublished. Reconfigurable computing is however being used on electronic trading systems, one example is Credit Suisse's latest joint venture with Celoxica which has been widely published [4]. Other research has been performed which is related and may prove insightful to this project.

One study which proved particularly interesting and relevant was *FPGA accelerated low-latency market data feed processing* [40], which explored parsing market data feeds directly from a network interface. The study found that their hardware implementation could handle 3.5 million updates per second, or roughly 12 times the current real-world maximum rate. The implementation also gained constant latency when processing messages, providing a solid service to the application layer which was still implemented in software on the CPU. The relation to this project is that with moving the application layer down into the hardware, further work could join the two solutions into a single hardware implementation which would increase performance further.

A similar study *Compiling policy descriptions into reconfigurable firewall processors* [41], looked at implementing a network firewall using a policy specific language in hardware, taking the network packets directly off the interface attached to the FPGA like in the previously mentioned paper. The relation to this project is in the policy specific language may provide an implementation option for the algorithmic trading application. Firewall rules are normally written as a set of policies which packets must adhere to before being able to continue on their journey. This is very similar to how an algorithm may decide to trade, it can be thought of as constantly wanting to trade, but a set of policies dictate to it when it is allowed and not allowed to trade. Therefore to successfully trade all policies must be met before the order is generated and sent to market.

Finally on the network side of FPGA research, *Reconfigurable Architecture for Network Flow Analysis* [42] looked at processing network information in hardware. Moving slightly further away from this project, however some of the goals are still very similar, such as the ability to parse large amounts of information in real-time rather than produce the answer to a single complex calculation and processing multiple flows in parallel which can be mapped to handling multiple market data and order feeds. The advantages that were seen where a huge increase in maximum throughput, going from about 100 Mb/s for the software to multi-gigabits per second in the hardware implementation.

There has been a large amount of research into other applications of FPGAs within financial computation, but they have mainly been focused on pricing exercises of derivatives such as options. *Accelerating Quadrature Methods for Option Valuation* [43] looked at implementing quadrature methods of finding the price of options in reconfigurable hardware and GPGPUs, finding that the FPGA implementation was considerably faster than the software on CPU (32x increase), while being far more economical on power usage than the equivalent GPGPU implementation (8x decrease). This is particularly interesting if in future, horizontal scaling of system is the only way of keeping up with demand and if the FPGA implementation was far more power efficient, then the cost of scaling over the longer term would be reduced with the added environmental benefits.

In a different approach, *Credit Risk Modelling using Hardware Accelerated Monte-Carlo Simulation* [44] looked at implementing Monte-Carlo methods of accelerating financial calculations, in this credit

risk modelling. This time the speed up observed compared to the software implementation was between 60 and 100 times faster which is an incredible amount quicker. This experimentation used a discrete event based simulation which can also be related to the work this project is outlining. The orders and market data events that the system can will receive can be related to events that must be processed and the different events result in different outcomes.

An algorithmic trading engine can be likened to a complex event processor (CEP) because of it consuming market data information from which data needs to extracted and decisions on a number of activities made. CEP has had a lot of focus from both industry and academia and so there is a lot of available material on the area. Recent focus has been on processing on distributed systems to increase the event throughput and decrease the latency of processing an event. The Next CEP [45] provides a number of additional and interesting features distributed placement of processing based on a cost model of operators. Although not directly related because the algorithmic trading engine has to contain multiple processors, one for each order as each has a different state and each event must be processed multiple times in this relation, it does provide an explain for the current state-of-the-art in CEP.

An interesting avenue of research is heterogenous clusters such as the Axel [33] and QP [46] clusters. As previously mentioned in Section 2.6.2 the Axel project and other heterogenous clusters consist of a number of high-performance computers each of which contain a number of different processing units. In the case of Axel and QP this is a general purpose processor followed by multiple general purpose graphic processing units and FPGAs to assist in computations. The difficulty in this approach is identifying which areas of a problem are best suited to which computational unit, as each have different performance properties. Then implementations needs to be produced to execute on the various components, each of which have different programming methodologies and so is non-trivial. The QP cluster is supplied with the Phoenix application development framework, where the Axel cluster has a Hardware Abstraction Model (HAM) and a Map-Reduce framework to aid application development. Heterogenous systems provide an interesting way of increasing cluster performance, while not significantly increasing the size of the cluster and managing to improving power efficiency. This project may be suited to running on a heterogenous system if parts of the model can be translated to the other computation units.

Although not related to financial aspects of computing, *Parametric Design for Reconfigurable Software-Defined Radio* [47] provided a lot of insight into *run-time reconfiguration* of FPGAs. This is a particularly interesting area and relevant to the aims of the project to create a fully-fledged application. It would be highly beneficial if the reconfiguration times where acceptable so that during the trading day, algorithms could be swapped and specialised further, depending on market and trading conditions.

2.8 Summary

In this background I have given an introduction to the business requirements of this project, starting from an introduction to equity trading before moving on to how algorithmic trading is performed. From this information, the interest in exploring the implementation of an algorithmic trading engine in a reconfigurable hardware because of the benefits that such an architecture has over a purely software based implementation should be clear.

In more detail I have laid out the financial models that this project has explored in hardware. The topic is extremely vast and there is always the ability to make algorithms more complex and intelligent so that they perform better. However without the initial work into simple algorithms, this is impossible to perform. This project provides such work for exploring more complex issues and designs. As this project heavily relies on reconfigurable computing and FPGAs, a background on the current state-of-the-art has been provided with introductions on how current applications are developed for hardware.

Finally I have given information about what current research has been performed in similar and related areas. Although no research has been published in this specific area there is an obvious desire for hardware implementations of electronic trading systems, as shown by recent news and the obvious benefits that FPGAs can afford their applications. These resources present a number of ideas which helped in solving the problems set out by this project and show scope for further work. Through looking at previous work, I have been able to gain ideas on further directions in which to take this project, such as partial run-time reconfiguration of the device. These could provides large benefits to this system and increase performance further over the initial speed boost provided by using reconfigurable hardware. This has helped increase the quality of this project and the report presented here.

Chapter 3

Electronic Trading Framework

Equity orders and system communication underpin the overall design of the project. How these are modelled is important later in how the algorithms of the algorithmic trading engine are designed. Therefore it is paramount that these models are robust and perform the designed operations. As these models are general in the fact that many systems would make use of them, not only the algorithmic trading engine, I decided that this part of the project should be treated as a framework. This allows the work here to be used in many other systems later and ensures that the code is abstract of any specifies to the algorithmic trading engine.

The framework is separated into to two main packages. The first models the equity order, the second models inter-system communication. Although the actual communication is out of the scope of this project, because of the design principles followed in this framework, creating the relevant modules to serialise the data for communication will plug-in to the framework. In addition I propose some ideas which would make communication efficient when implemented. Specifically, this chapter will cover the following:

- I want to show the position of the framework within the structure of the application, helping you picture the application construction when I cover later topics; in section 3.1.
- Memory management is an important part of the application and poses some difficult problems which I will show before describing their solutions; in section 3.2.
- Equity order model, including its functionality and attributes. Two concrete models are available that represent *parent* and *child* orders, with parent order supporting *split* and *merge* functionality to create and destroy child orders; in section 3.3.
- The system communication through issuing of commands and responses to control and provide feedback on orders. Providing interactions with remote systems; in section 3.4.
- How market data is modelled and supplied to the system. The three forms of market data update in the framework, providing some of the information that the system needs to process and form decisions upon; in section 3.5.
- Timer events provide notifications into the system. Callbacks are registered and activities can be triggered on receiving a timer event if other criteria are met; in section 3.6.
- Summary of the work completed within the frame work, its impact on the project and possible uses for it in other applications and research; in section 3.7.

3.1 System Overview

Figure 3.1 shows an overview of the overall structure of the application which will form over the next few chapters. There are five main packages within the structure, and I will discuss each package individually in detail over a chapter. This chapter will focus on the electronic trading framework which underpins the overall design of the application, seen at the bottom of the diagram. The structure contains no cyclic

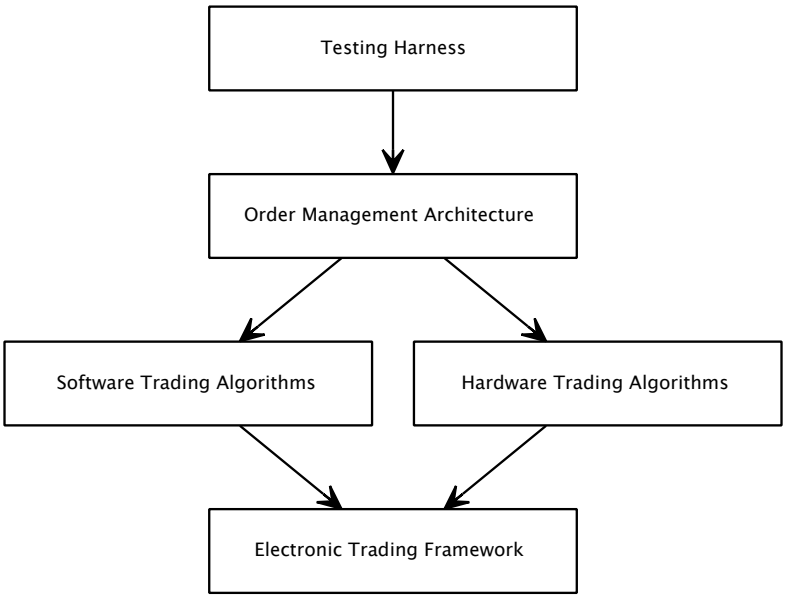


Figure 3.1: Overview of the applications structure, categorised by five main packages.

dependencies at the package level, shown in the diagram. This aids reusability and simplifies maintenance of the system. The structure is built bottom up, from the framework with all components making uses of this functionality. Additionally, the *Order Management Architecture* utilities the algorithms of either technology and finally the *Testing Harness* sits on top of this, providing a testing framework.

Prohibiting cyclic dependencies is important as it allows packages to be used independently of others, only requiring those that are below them. We can see from the diagram that the algorithms could be used in a different system, as long as they shared use of the *electronic trading framework*. This results in packages of high cohesion and low coupling, presenting the layered structure before you. Cyclic dependencies can be checked in a number of ways, there are tools available which produce graphs from parsing the source files of a project and these can be checked manually. Also by default, when compiling a C++ program, if there are cyclic dependencies then the build procedure must be change, these changes haven't been made in my build procedure so I know there are know cyclic dependencies.

3.2 Memory Management

C++ relies on the programmer to carry out memory management to prevent memory leaks. This system requires a large number of objects to have references to the same containers which makes the process of manual memory management more difficult. It would disastrous if one object deleted a container than another object still required. A solution provided by the C++ standard template library, smart pointers. These contain and manage an item, keeping a reference counter of the number of objects that current have references to that item. When that count reaches zero, the item can safely be deleted. There is a small amount of overhead in this solution, most of which is derived from ensuring the implementation is thread safe. I have deemed this a worth trade-off though, as using this solution not only improves faith in the system being correct but also speeds up development time.

As the smart pointer is itself an object (`shared_ptr<T>`), methods that previous accepted a reference to an object must now accepted a smart pointer. Two interesting problems arise from this solution, the first is when singleton objects must be managed by a smart pointer so that they can be used in the

```
template<class T>
class NoOpDeallocator {
public:
  NoOpDeallocator() {
  }

  NoOpDeallocator(const NoOpDeallocator<T>& copy) {
  }

  virtual ~NoOpDeallocator() {
  }

  void operator()(T* object) {
  }
};
```

Code Block 3.1: The no operation deallocator for use with the `shared_ptr` to manage singleton objects.

```
NoOpDeallocator<T> deallocator;
shared_ptr<T> ptr(&singleton, deallocator);
```

Code Block 3.2: Use of the no operation deallocator for managing a singleton object.

```
class ClassA: public AbstractClass {
public:
  virtual ~ClassA() {
  }

  static shared_ptr<ClassA> create() {
    shared_ptr<ClassA> result(new ClassA());
    result->weak_this = result;
    return result;
  }

  shared_ptr<ClassA> getSharedPtr() {
    shared_ptr<ClassA> result(weak_this);
    return result;
  }

protected:
  ClassA() : AbstractClass() {
  }

private:
  weak_ptr<ClassA> weak_this;
};
```

Code Block 3.3: Creation of an object managed under a `shared_ptr`.

```
shared_ptr<ClassA> typedPtr = dynamic_pointer_cast<ClassA(untypedPtr);
if(typedPtr) {
  // untypedPtr has a ClassA object stored.
} else {
  // untypedPtr doesn't have a ClassA object stored.
}
```

Code Block 3.4: Safe checking of a smart pointer to prevent errors.

same methods. The second objects which wish to reference themselves and use these methods which only accept a smart pointer. Singleton objects should never be deleted during the program execution, so if the reference count reaches zero we don't want the object deleted. This can be achieved through the use of a deallocator which doesn't delete the object when asked. Code block 3.1 shows the design and implementation of such a deallocator, with the method on lines 13 and 14 being the critical method that does nothing. Code block 3.2 shows how this is used in combination with a smart pointer to avoid the object being deleted upon the reference count reaching zero.

The second problem is caused by the style of creation for the objects managed by smart pointers. To ensure that all objects required to be managed are managed, I chose to change the scope of all constructors from `public` to `protected`. This still allows classes to extend these classes but prevents the creation of these objects from outside of the class. To allow these objects to be created, I provide a `static` factory method for each constructor, taking the same parameters as that constructor . It will create the new object, place it under management and return the smart pointer. Code block 3.3 shows this design. Now we have the problem of if the object wants to refer to itself through a smart pointer, it cannot contain its own reference counting smart pointer as this would result in the reference counter never dropping below one and the memory resource would be leaked. The solution is a different type of smart pointer which doesn't contribute to the reference count (`weak_ptr<T>`). The object can contain this and will still be destroyed when necessary, more importantly, a reference countering smart pointer can be created from this object and provided to any other class. This is shown in method `getSharedPtr()` in code block 3.3, with the weak pointer on line 22 which is initialised on line 8 in the static factory method.

Other benefits to this form of memory management come when dealing with casting the smart pointers. Later you will see that this is an important part of the structure of the application. As use of exceptions within the code base are forbidden because of their performance degradation, even when not being thrown, confirming a correct cast is more difficult. Casting with smart pointers is carried out by provided template methods such as `shared_ptr<T> static_pointer_cast(shared_ptr<U> const & r);` for down casting and `shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const & r);` for up casting. These don't throw exceptions, instead they will return a null smart pointer on failing. This can be simply checked using an if statement, shown in code block 3.4 and allows for errors to be handled without exceptions in a clear and careful manner.

3.3 Equity Orders

The equity order model is designed as a storage container to hold all of the relevant information required by a general trading system. In a full electronic trading network, additional information may be required which can be catered for by extended the classes provided. Using object oriented principles, the equity order provides encapsulations and data abstraction, presenting a set interface of methods to operate on the data held within. The framework provides an abstract base class, `Order`, which contains most of the functionality, two concrete classes are provided for use in the `ParentOrder` and `ChildOrder` classes. The `ParentOrder` class adds the functionality to *split* itself into multiple, smaller `ChildOrders`. These `ChildOrders` can then be merged back in with the `ParentOrder` at a later time, after taking any changes in the state of the `ChildOrder` under consideration.

The `ParentOrder` class maintains its own record of its `children`, this allows the process of providing functionality across all the children of a `ParentOrder` to be simplified. To avoid any cycle dependencies, `ChildOrders` cannot have a reference to their parent. Instead, in addition to the information held by their base class, they contain the unique identifier to their parent order. Table 3.1 lists the different attributes held by the three order classes, their type and a summary of the reason why the exist. Figure 3.2 also shows the class diagram for the order package. The relationships between the three order classes can be seen on the far left of the diagram. The main functionality of the `Order` class is to handle executions. These take place at a remote location and notification of the event with its details are received by the system maintaining the order. The details consist of the quantity of the shares traded and the price that they were traded upon. An execution reduces the quantity of shares available on the order by the amount traded and increases the transaction cost by the multiplication of the trade price and the trade quantity.

`ParentOrders` posses two critical pieces of functionality. The first is *splitting* the order into mul-

Name	Type	Class	Summary
identifier	`string`	`Order`	Is unique amongst all orders, allowing them to be differentiated and stored by using as a key.
symbolName	`string`	`Order`	Identifiers which stock the order is trading shares in, additionally provides another form of categorisation.
side	`bool`	`Order`	*True* if the order is buying shares, otherwise *false* for selling shares.
quantity	`int`	`Order`	The number of shares that the order currently has available to trade.
originalQuantity	`int`	`Order`	The original number of shares that the order had to trade before taking into account any executions or order splits to get the *quantity*.
removalFlag	`int`	`Order`	*True* states that the order is suitable for being removed, *false* states that the order cannot currently be removed.
orderStatus	`int`	`Order`	*True* states that the order is active (it has been accepted), *false* states that the order is not currently active.
transactionCost	`double`	`Order`	Initially zero, this is increased after every execution on the order by the quantity of shares on the execution multiplied by the price of the execution.
orderTimeInForce	`OrderTimeInForce`	`Order`	Represents the order's time in force, described further in section 3.3.1.
orderTradingDetails	`OrderTradingDetails`	`Order`	Represents the orders trading details, described further in section 3.3.2.
orderType	`OrderType`	`Order`	Represents the order's type, described further in section 3.3.3.
orderTime	`Time`	`Order`	Represents the time that the order was created.
childrenVolume	`int`	`ParentOrder`	Value is sum over all of the children's original quantities.
children	`map<string, ChildOrder>`	`ParentOrder`	Maintains a reference to every current child of the parent order, mapped by the child's identifier.
parentIdentifier	`string`	`ChildOrder`	Identifies the parent of the child order, allowing the parent to be found provided a child order and map of parent orders.

Table 3.1: Overview of the attributes held by an equity order.

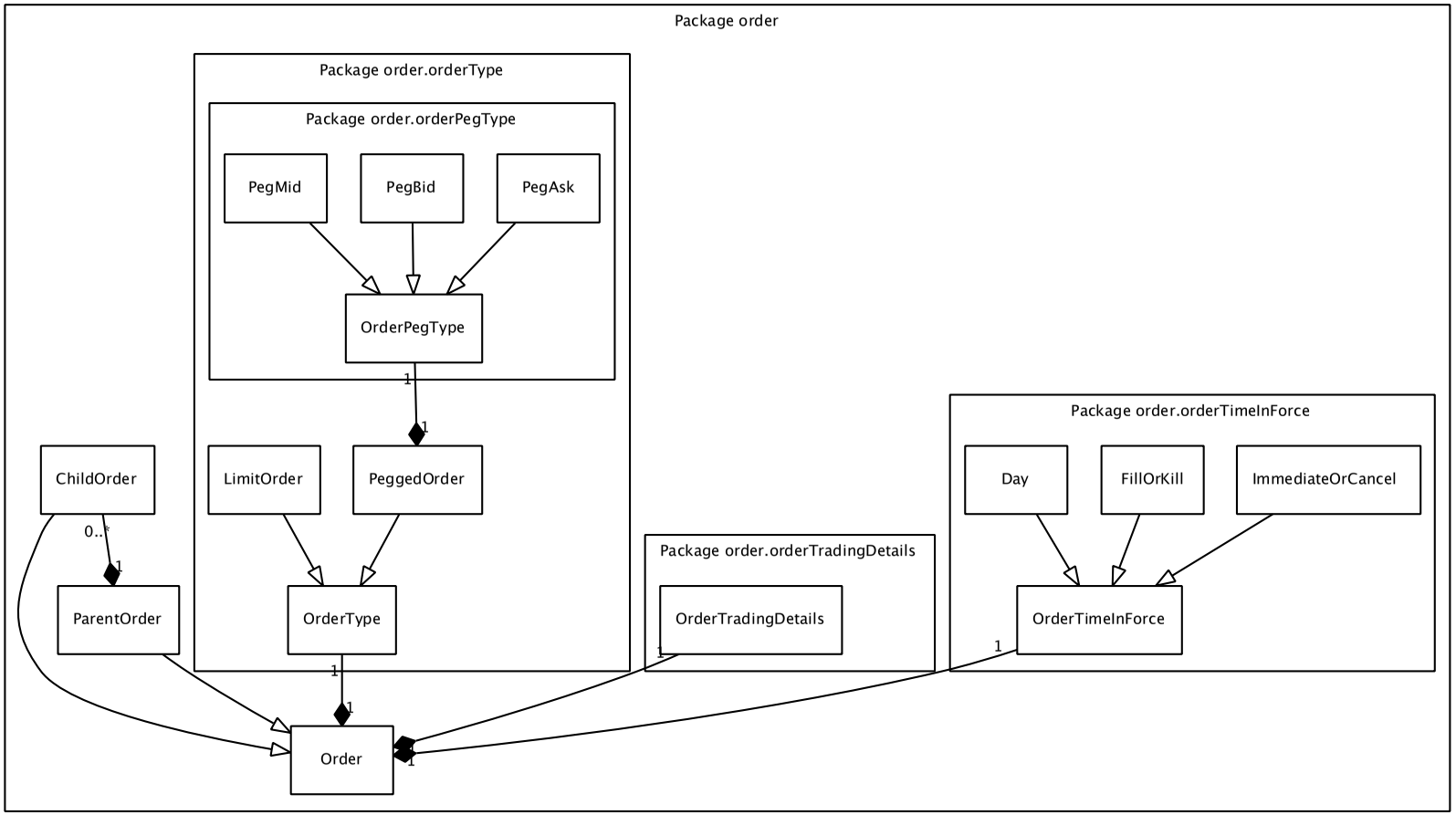


Figure 3.2: Class diagram of the order package, part of the electronic trading framework.

Name	Order Lifetime Restrictions	Order Execution Restrictions
`Day`	None, an order will last until it is either completed, cancelled or the end of the trading day occurs, which ever is first.	None, any executions on the order can be any quantity less than or equal to the order's current quantity.
`ImmediateOrCancel`	The order only lasts one processing event before it is either cancelled, completed or partially completed depending on any execution sizes.	None, an order will last until it is either completed, cancelled or the end of the trading day occurs, which ever is first.
`FillOrKill`	The order only lasts one process event before it is either cancelled or completed.	Any execution on the order can only be equal to the quantity of the order, it cannot be less.

Table 3.2: Overview of the types of the order *time in force* attribute.

tiple, smaller `ChildOrders`. The second is the converse, *merging* which joins a `ParentOrder` with one of its children. When a `ParentOrder` is split, the `ChildOrder` created effectively copies the `ParentOrder`'s attributes. These can then be modified by the process that prompted the split, if desired. The decision to split and any modifications are made automatically by the trading algorithm in the algorithmic trading engine or a human if being traded manually. The only precondition is that the `ChildOrder`'s quantity must not be greater than the available quantity on the parent order. Merging destroys the `ChildOrder` and the `ParentOrder` receives the available quantity and the transaction cost from the child back onto its own. Merging is the final act of a completed `ChildOrder` (an order is completed when its available quantity reaches zero). In this case, the parent receives no extra quantity but the full transaction cost of the child order. When the available quantity and the number of children of a `ParentOrder` reaches zero, then the `ParentOrder` is complete.

When a split or a merge takes place, the `ParentOrder` must update its map of references of its children to ensure that it is correct. The reason why I choose to allow the `ParentOrder` to maintain a reference to each child, is that certain operations on the `ParentOrder` require an operation on each child. For example, if a `ParentOrder` is requested to be cancelled, all children of that parent must be successfully cancelled before itself can be. This is simpler to accomplish with this implementation as `ParentOrder` can perform the cancel action across all of its children. This ensures that all data structures remain in a consistent state between transactions and no children are left as *orphans*. Any orphaned children could be potentially costly bugs to the owner of the trading system.

The next three subsections cover the three larger attributes of the `Order` class that contain multiple pieces of information. What information they store is also dependent on their type. Therefore it was a natural choice to model the as independent class hierarchies. Each has its own abstract base class which the `Order` class refers to, but each concrete class implements the functionality differently, exhibiting its own behaviour. I was tempted to allow the `Order` classes to be *templated* using these three classes, but after careful consideration I decided that the advantages did not outweigh the disadvantages. The resulting implementation would have been very verbose and required significantly more code. It would also have made changing the orders information represented by these classes more difficult as it would now be integral to the orders type. Finally C++'s template functionality isn't as good as Java's generics functionality to work with, so overall, provided little extra benefit.

3.3.1 Time In Force

There are three different order *time in force*s supported by the framework. They are *Day*, textitImmediate or Cancel and textitFill or Kill. Table 3.2 describes their differences over two categories. Both *Immediate or Cancel* and *Fill or Kill* can be described in terms of a *Day* order with some additional functionality and parameter setting. Although this would require further functionality at a lower level within the application. The design of the classes follows the same hierarchical design which is often used in the framework and is down in figure 3.2, the further right package. Each of the two categories in table 3.2 is implemented using a pure virtual function which must be implemented by the concrete class extending

Name	Summary
`PegAsk`	An order which is pegged against the ask price has the same price as the cheapest selling order on the trading venue.
`PegBid`	An order which is pegged against the bid price has the same price as the most expensive buying order on the trading venue.
`PegMid`	An order which is pegged against the mid price has the same price as the average price between the cheapest selling order and most expensive buying order on the trading venue.

Table 3.3: Overview of the order *peg types*.

the abstract base class. In the case of checking if an order has completed, when the `Order` class checks its status, it also takes the *or* with the result from the time in force. Both the *Fill and Kill* and *Immediate or Cancel* classes return *true* as they always consider themselves complete. The converse, *Day* always returns *false*, it will only consider itself finished if it has no remaining quantity, which the `Order` class checks. In the case of checking the execution quantity, both the available quantity and the suggested execution quantity are input. The method returns the boolean result of whether they are equal if it is a *Fill or Kill*. Both *Day* and *Immediate or Cancel* return true if the value of the execution quantity is less than or equal to the available quantity.

3.3.2 Trading Details

I won't cover order *trading details* in much depth here as it is more relevant when discussing the different algorithm designs. This is because the different order trading details provide a place where additional information can be stored and used by different modules. They can also store state so that they can keep track of individual orders through their own pipeline. Again, the design is a class hierarchy, with the individual types providing a method of identifying order parameters which may affect the systems decision to accept an order or not. For example, if the order trading details on an order received doesn't match any known trading style, then the order is rejected as it is unsupported. Unlike with the order time in force, there are no abstract methods specified by the base class, this is because the set of attributes that each of the concrete implementations will have is going to be very diverse. The disadvantage with this is that the class will have to be cast when required by a module to use that information stored. Figure 3.2 shows that the `Order` class owns a single copy of an implementation of the *OrderTradingDetails* abstract base class. Further details on the specific concrete implementations are provided in chapter 5 with their relevant algorithms.

3.3.3 Type

There are two order *types* available initially within the framework. It is these that control the price of the order, which is a notable omission from table 3.1. The first order type is a `LimitOrder`, this has a set value for the price which does not change unless the order is amended. The second type is a `PeggedOrder`, where the order's price is depending on the current market price of the order. Table 3.3 describes the three different forms of pegging that a `PeggedOrder` supports. Instead of each pegging option directly extending the abstract base class `OrderType`, they are owned by the `PeggedOrder` class. This is shown in figure 3.2 in the left most sub-package.

A pegged order's price also depends on a number of other attributes. An *offset* which moves the price in either positive or negative direction and a *limit* which prevents the price from exiting a predetermined boundary. Each individual concrete pegging type prices itself against market data updates that it is provided with, the different implementations select the correct information for themselves. To streamline pricing and prevent reduce casts, `OrderType` contains a method which accepts market data updates to price itself against, pegged orders override this to price themselves. However `LimitOrders` ignore this data as it is of no concern to them.

Name	Summary
`NewOrder`	Provides a new order that the owner would like the system to put under its management.
`CancelOrder`	Specifies an existing order that the owner would like the system to no longer manage.
`AmendOrder`	Specifies changes to an existing order which is under the management of the system, initiated by the owner if the order.

Table 3.4: Overview of the commands available through system communication.

3.4 System Communication

The main components of the system communication package are *Commands* and *Responses*. Additionally both *Market Data Updates* and *Timer Events* are considered part of the communication package. They are distinctly different though from commands and responses though so I will discuss them later in this chapter. The hierarchical theme from modelling equity orders carries on throughout the system communication package. An abstract base class is used to specify common functionality with concrete classes implementing specific behaviour. This design structure aids further development at a later stage with the addition of new concrete classes and behaviours.

System communication at the lowest level would be concerned with how the data is serialised before being transmitted across a network. In this project though I have concentrated on a single system so their is no need to implement this functionality. However because of the design techniques used, adding these modules to the framework should be straightforward. Communication would most likely be across a network, as electronic trading systems would be distributed across multiple hosts for performance and redundancy. Alternatively, the systems could occupy a single host and communicate using inter-process communication methods provided by the operating system.

Commands are used to initiate actions on orders, responses are sent as notification of the outcome of commands. Additionally there are three other responses which are sent independent from receiving a command and these are used to notify the owner of the order of a state change. Both commands and responses contain the order that they are relevant to. It would be expensive to continually transmit the full orders contents between the systems on every communication. Therefore I propose that if the low level communication modules are implemented that they provide a dictionary of orders previously seen. This would allow only a unique identifier and any changed information to be communicated, where at the other system, the module could look up the reference to the instantiated order to pass through and use within its application instance.

If this framework was to be integrated into an existing infrastructure, their would undoubtedly be additional information required to be passed with updates or other forms of system communication. The best solution to this is to make use of C++'s multiple inheritance, creating a new bespoke abstract base class to store this information. Now after extending all of the frameworks concrete classes with your own bespoke classes, these can additionally extend your new abstract base classes. Figure 3.5 illustrates this design principle on an example set of classes.

Table 3.4 outlines the three different commands supported by the framework. They are `NewOrder`, `CancelOrder` and `AmendOrder` commands. The first six responses outlined in table 3.5 are directly related to these three commands. There is a response *accepting* and *rejecting* each command. There are three extra responses which are used to notify the owner of a state change on an order. They are `Complete`, `Execution` and `ForceCancel`, they are the last three responses in table 3.5. Figure 3.5 shows the response packages class hierarchy, this ensures code duplication is kept to a minimum and that implementing behaviour based on receiving different responses is quickly definable.

Responses carry a significant piece of extra information, they declare whether after being seen, the order they refer to has ended. The third column in table 3.5 states for which responses this is true. I added this functionality so that as the response passes through the system architecture, the varying levels which may hold state about that order know whether they need to update themselves. This ensures that all data structures are maintained in a consistent state and memory leaks do not occur with unwanted objects not being deleted. All of the `Reject` responses carry a string field denotes a message depicting the

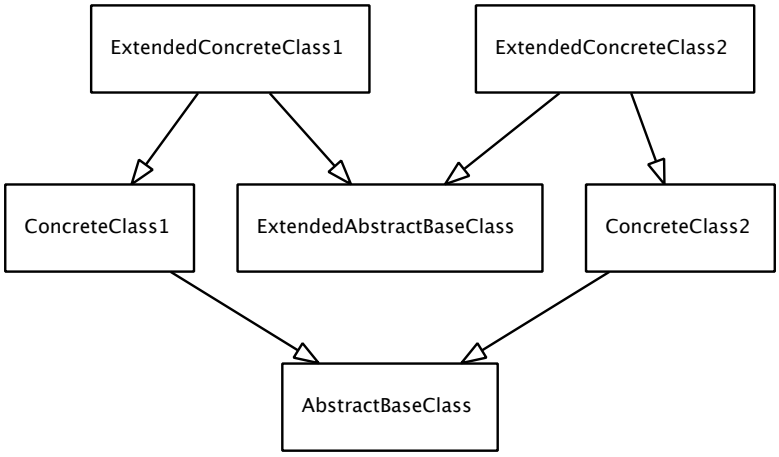


Figure 3.3: Extending a class hierarchy for integration with an existing system.

Name	Cause	Order Ended	Summary
`AcceptNew`	`Command`	No	Responds to a `NewOrder` and confirms that the order was created successfully
`AcceptCancel`	`Command`	Yes	Responds to a `CancelOrder` and confirms that the order was cancelled successfully.
`AcceptAmend`	`Command`	No	Responds to a `AmendOrder` and confirms that the desired changes to the order were completed successfully.
`RejectNew`	`Command`	Yes	Responds to a `NewOrder` and notifies the owner that the order was not created on the system.
`RejectCancel`	`Command`	No	Responds to a `CancelOrder` and notifies the owner that the order could not be cancelled at that time.
`RejectAmend`	`Command`	No	Responds to a `AmendOrder` and notifies the owner that one or more of the order attribute changes could not be completed.
`Complete`	`Update`	Yes	Independent from a direct command, sent to the owner of an order which has just been completed and removed from the system.
`Execution`	`Update`	No	Independent from a direct command, sent to the owner of an order which has just had an execution taken place.
`ForceCancel`	`Update`	Yes	Independent from a direct command, sent to the owner of an order which has been removed from the system without a `CancelOrder` having previously been received.

Table 3.5: Overview of the *responses* available through system communication.

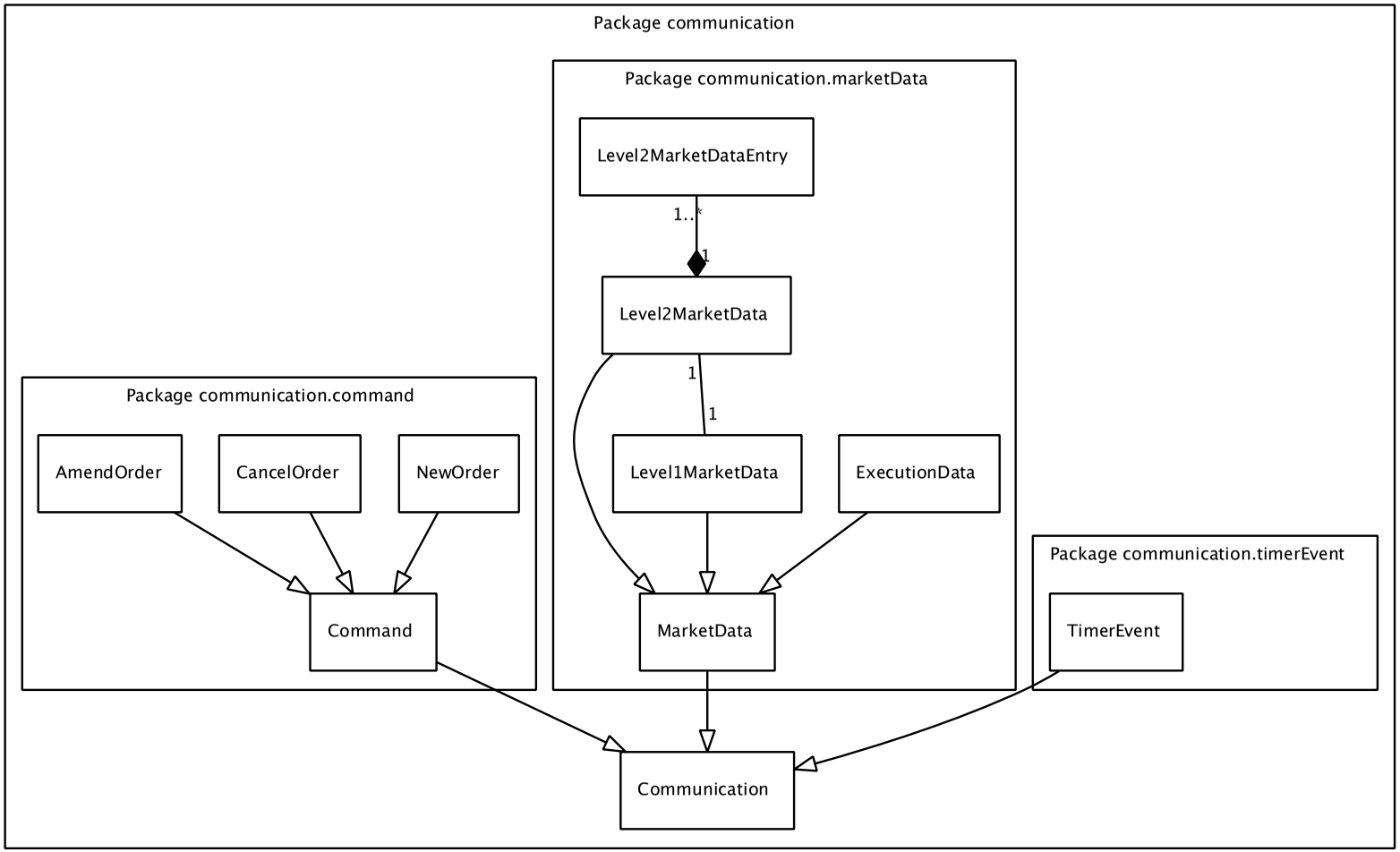


Figure 3.4: Class diagram of first half of the communication package, containing *commands*, *market data* and *timer events*.

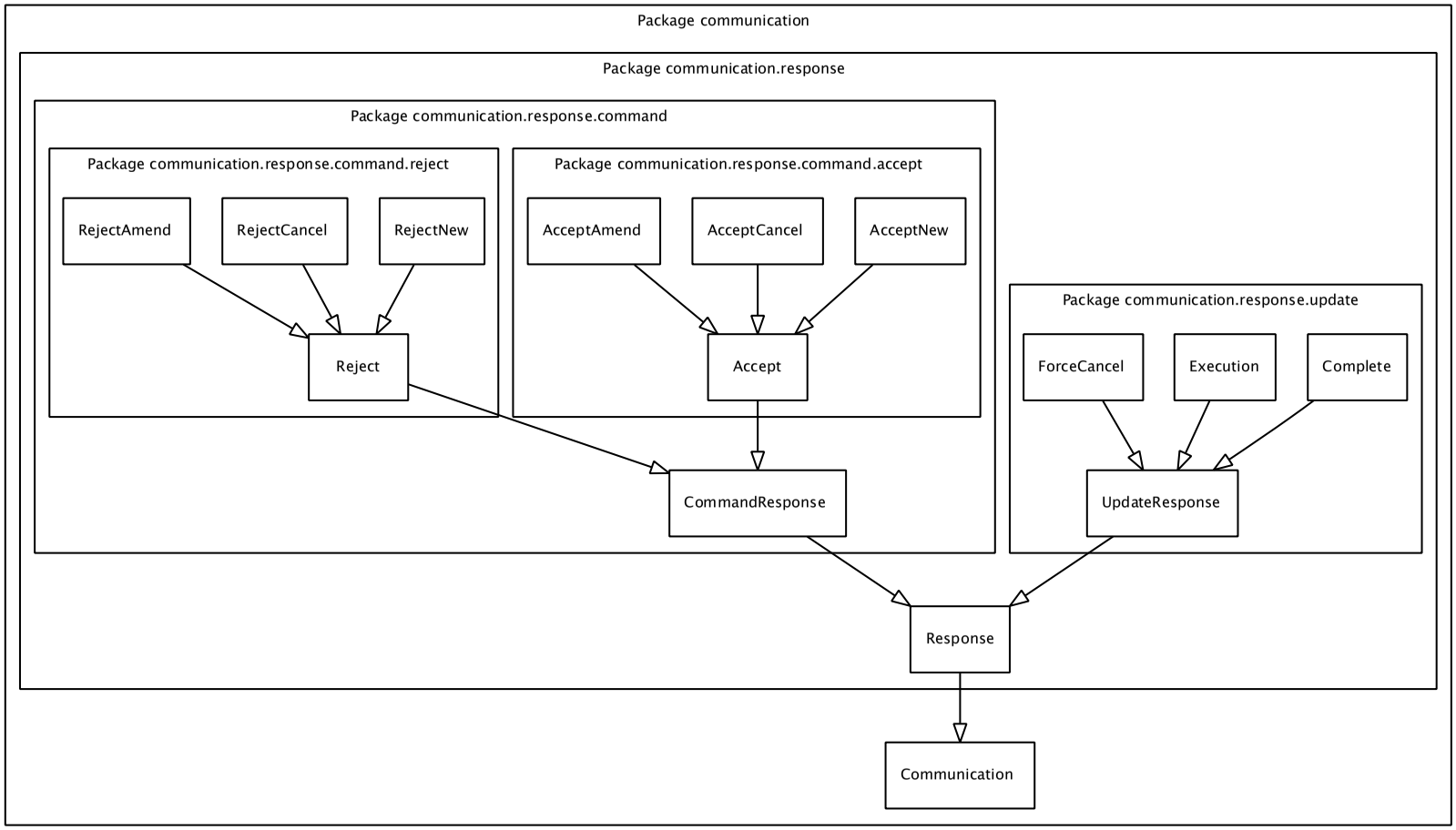


Figure 3.5: Class diagram of the second half of the communication package, containing *responses*.

Name	Summary
`Level1MarketData`	Carries two pieces of information, the best buying price on the market and the best selling price, from this the mid price can be calculated from the average of the two prices. The best buying price is the most expensive price that an entity is willing to pay for the stock. The best selling price is the cheapest price that an entity is will to sell stock at.
`Level2MarketData`	Instead of carrying only two prices, level 2 market data carries the full information of all of the orders on the market. In addition to all of the prices, this also includes the size of all of the orders, allowing the amount of liquidity on the market to be calculated. The information is provided in two sorted lists, the buy side is sorted in ascending order where the sell side is sorted in descending order.
`ExecutionData`	Like level 1 market data updates, it carries two pieces of information. However this a price and quantity pair, but unlike level 2 market data updates, this represents a trade that has taken place. The price is the agreed price for the trade and the quantity is the number of shares exchanged. This allows the price of a symbol to be calculate over all of the trades that have taken place.

Table 3.6: Overview of the three different market data update types.

reason for the rejection. This is useful when debugging the system and might enable systems to automate their reaction to a rejection. However in production use, rejections should be rarely seen as it is probably either due to a malformed order or the system receiving the information encountering problems. Neither which are desirable. Commands and responses don't provide any additional functionality and are mainly used as an abstraction layer and container. Their responsibilities would need increased with the addition of more low level functionality such as network communication. Further behaviour to handle serialisation and additionally information required by other systems taking part in the communication would need to be added.

3.5 Market Data

Market data provides critical information to an algorithmic trading engine. It has three forms within the frame work, `Level1MarketData`, `Level2MarketData` and `ExecutionData`. These are outlined further in table 3.6. Level 1 market data updates contain a subset of the information in a level 2 market data update created at the same time. Therefore, a level 1 market data update can be created from a level 2 market data update. Figure 3.4 show the class structure of the market data package as the central package in the figure. The only additional class which isn't directly part of the class hierarchy is the `Level2MarketDataEntry` class which is used to store the pairs of values for each order on the market, ensuring the pairs remain together. The pair of information is the quantity and the price of that entry, there exists an entry for every order on the market.

Market data updates are a member of the system communications package. This is because the updates are normally created in a remote location and are sent through their own infrastructure before reaching the system via the network interface. Unlike commands and responses though, market data is a purely unidirectional form of communication. The system subscribes to the information that it is interested in, the market data infrastructure then pushes new information to the system as it arrives. Normally the market data infrastructure and receiving the updates would be carried out by third-party tools and applications. The two biggest market data providers are Thomson Reuters [48] and Bloomberg [49], both offer extensive amounts of support when working with their data feeds. Additionally, both supply libraries that implement the low level code to receive and convert the data received into provided representations. This would have to be integrated in with the system if it were to receive real market data, again due to the design of the framework and later applications, this should be a straight forward proce-

dure. One the data is consumed, it would need to be transformed to the data representation presented here so that it could be used throughout a system built on this framework. The reason why market data is taken from providers like Thomson Reuters is because they provide a consolidated market data feed. This means that information is received from multiple exchanges and sources, aggregated by them and then provided to us. This simplifies and speeds up the process of getting all of the information that the system requires.

3.6 Timer Events

The timer events are designed to provide a callback into the system to allow selected functionality to execute. These callbacks need to be registered with the module that will inject the timer events back into the system. I have decided to integrate them within the system communication package, even though it is not likely to be communicated externally but because it passes (all be it, limited) information to the application. The simple timer event class structure is shown in figure 3.4 on the far right. The algorithms described later make use of the timer events to ensure that their state is updated regularly and that if they wish to trade orders at specific times. A schedular is suitable to injecting the events at the desired times. Timer events could be extended to include further information which could be read by the user to determine the specific behaviour initiated by the timer. The advantage of modelling them as part of the system communication is that they could then be communicated externally. This would involve deploying the injector of timer events on a remote system, but this could then supply events to multiple systems and further redundancy could be built in.

3.7 Summary

In this chapter, I have presented my solutions to modelling the low level data types within an electronic trading system. To make the most of this work presented here, I have treated these implementations as a framework. This required the models to be made as general and as robust as possible to ensure that they are correct. The benefits though are significant, with this work providing a base for a lot more further work and speeding up development time by reducing repetition. Further projects to developer order routers and applications that perform high frequency trading or statistical arbitrage would all benefit from making use of the framework presented here.

Before setting out the framework, I have shown out it helps to form the foundations for the complete system. This also ties together the work in the other modules which will be covered in later chapters. Now you will be able to picture the system as it is being built, knowing the coarse interactions which hold the application together. The design shown provides a solid layer based pattern which is provides part of the reason why this application is very flexible and extensive. This is important in allow the work completed here being as general as possible, meaning that it can be utilised fully for many further projects and research avenues. I have shown how I have solved the problem of memory management with the careful use of C++'s standard template library. In this I have laid out a scheme for memory management that could be used in any number of applications programmed in the C++ language. This scheme ensures that memory resources are never leaked and the objects that are created are always under management correctly, preventing anyone from misusing the application framework. This provides greater trust in the final product, speeding up development time and reducing the amount of time that is required for maintenance on the software. The memory management routines are spread out, ensuring that the system doesn't spend large amount of time maintaining itself, which could result in a price performance impact of the trading algorithms described later.

I have taken time to discuss my design decisions that I have taken over the process of producing this framework and shown the limits of the framework. Where these limits have been reached I have proposed ideas on how when implementing further modules they can be made more efficient, such as with keeping a dictionary of current orders at the communication layer bridge between the system and the network. This would prevent unnecessary transfer of information already known between systems, reducing bandwidth and improving network performance. Using C++'s multiple inheritance, I have shown how developers how are working with this framework can specialise it to their own applications and systems. By creating their own abstract base classes to contain all common functionality specifically

needed, their new concrete classes can extend both their abstract base class and my concrete classes. This provides a simple and scalable method to adding features and behaviour to the framework, without adding non-abstract concepts to the framework and reducing the generality of the code base.

Additionally, I have proposed a method of providing callbacks into the system from a remote location using timer events. These allow developers of a system built on top of this framework to implement behaviour on a regular basis. Like the rest of the communication principles presented here, they do not need to be communicated over a network from a remote location. These models and containers would be just as suited to remaining in a single host, this may provide performance benefits for systems working with a limited scope of orders as it would eliminate any bottlenecks acquired when communicating over an ethernet network. Finally, much of the work presented here formalises the interactions and terminology of many of the business principles that I introduced in chapter 2. These should provide concrete examples to the read of what is expected when trading equity orders, although the reader may not use understand completely why and when they would be used. This provides the grounding for understand the rest of this project which is reinforced by the extensive use of this framework presented in the rest of the applications described in this project.

Chapter 4

Order Management Architecture

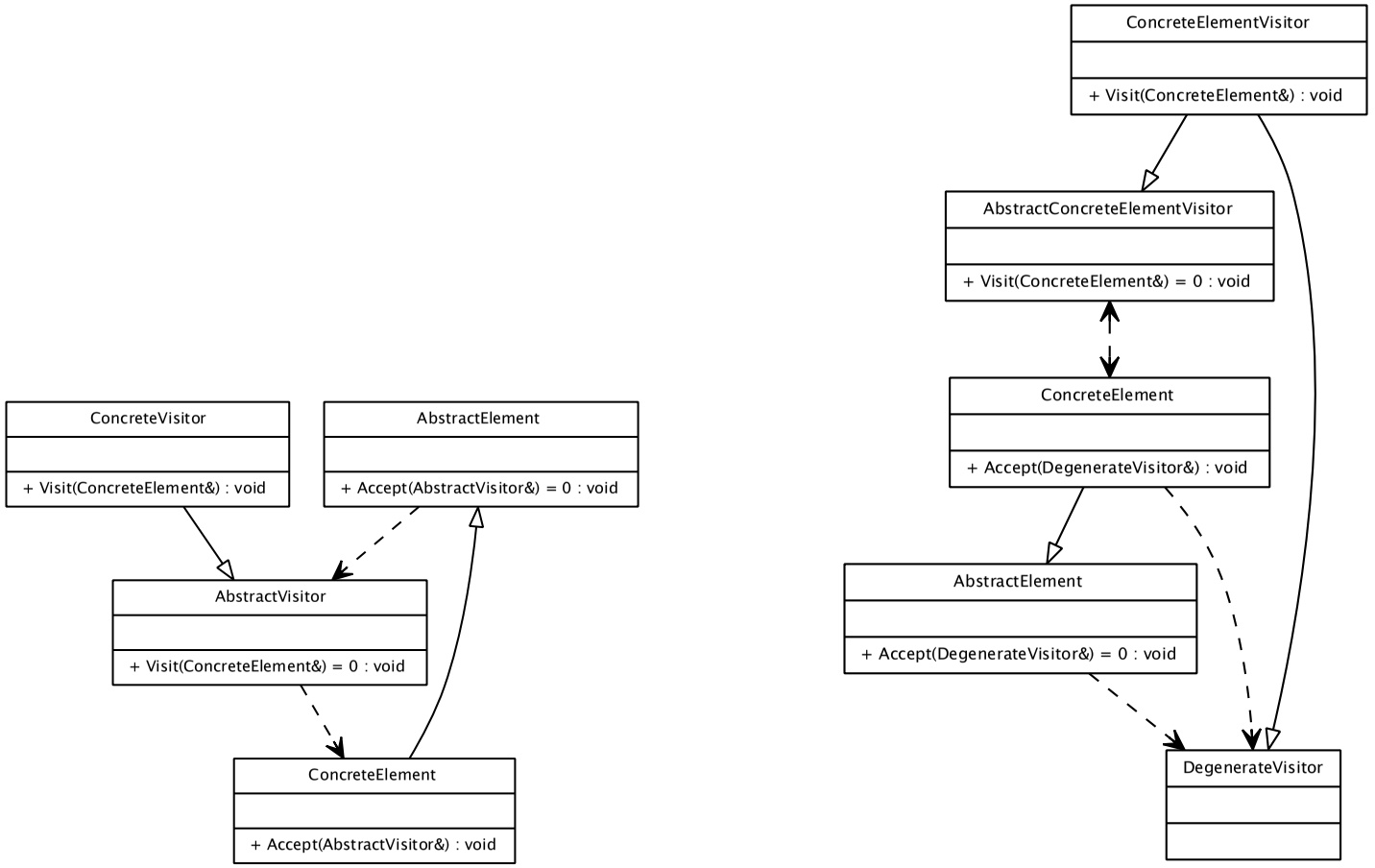
This chapter presents the underlying controller of the algorithms that will be investigated. We need to be able to handle the communication principles set out in the chapter 3 and formalise the effects that they have. This is so we can organise the data that we have and provide it to the relevant modules for processing in a quick and timely fashion. Therefore the success of the work presented here and a significant impact on the performance and therefore the quality of later work. We also require processes to maintain the information that we hold and keep it consistent.

It is the solutions to these points that I will set out in this chapter. Although the algorithms have no knowledge of the implementations carried out here, I feel that it is best to carry on this story of how the application was built up, providing the basis for the algorithms. The algorithms only use the framework and don't know about any of the order management to avoid cyclic dependencies, where the order manager requires to know about the algorithms so that it can pass information to them. The system could have been alternatively structured with the algorithms talking to the order managers, however this would have resulted in a far more complex and less abstract structure. Specifically, this chapter will cover the following:

- Design patterns and themes possess a large role in the architecture so it is worth spending some time discussing their use within the project. Specifically the use of the *acyclic visitor pattern* to remove cyclic dependencies within the system; in section 4.1.
- Providing a second abstraction layer for building course grained applications which handle multiple forms of different data. These provide a design pattern for building any number of large scale electronic trading systems; in section 4.2.
- Maintaining abstraction, I present my order manager implementation which caters for the absolute minimum amount to be able to successfully trade equity orders. It deals with interpreting communication and maintaining data; in section 4.3.
- Building on the order manager, I present the necessary extensions required to create a complete algorithmic trading engine, providing a home for any number of trading algorithms. I will also discuss advantages gained from system structure used; in section 4.4
- A summary of the contributions that the work presented in this chapter afford and what they mean for this and further work. Such as developing further trading applications; in section 4.5.

4.1 Visitor Design Pattern

The framework described in chapter 3 models the different actions and communication methods as individual classes. This has many benefits, but causes one main problem when handling these classes, the ability to differentiate them from other concrete classes of the same abstract base class. This is required so that different behaviour can be executed depending on the type of class received. The *Visitor* design pattern [50] solves this problem by providing a structure and method of determining individual types



(a) The standard visitor pattern.

(b) The acyclic visitor pattern.

Figure 4.1: Class diagrams of the two visitor design pattern.

```
1 void ConcreteElement::Accept(DegenerateVisitor& visitor) {
2   AbstractConcreteElementVisitor* typedVisitor = dynamic_cast<AbstractConcreteElementVisitor
        *>(&visitor);
3
4   if (typedVisitor) {
5     typedVisitor->Visit(*this);
6   } else {
7     // 'visitor' cannot visit the AbstractConcreteElemeentVisitor, handle error.
8   }
9 }
```

Code Block 4.1: Example implementation of the `Accept` method within the `ConcreteElement` class.

without large `if-else` blocks. Unfortunately they visitor pattern introduces systemic cyclic dependencies within the code base, causing a large amount of programming overhead. The cyclic dependency can be seen in figure 4.1a between the `AbstractElement`, `AbstractVisitor` and `ConcreteElement` classes. An example of the overhead later incurred is if another concrete element needs to be added which extends the `AbstractElement` class. This means that the `AbstractVisitor` class needs updating. Since the `AbstractElement` class depends on the `AbstractVisitor` class, every module that extends `textitAbstractElement` additionally needs to be recompiled. This recompilation can be very expensive, particularly in a large system. The process of adding additional concrete implementations of the `AbstractElement` class is also more time consuming and difficult.

4.1.1 Acyclic Visitor Pattern

The solution that I choose to implement in my application was to use the *Acyclic Visitor* design pattern [51]. Figure 4.1b shows the class diagram for the acyclic visitor design pattern. It is clear to see that there are no cycles within the dependencies except for between the `AbstractConcreteElementVisitor` and the `ConcreteElement`. However this does not provide the same problem as before as the dependency can be separated from the header file of the `ConcreteElement` into its source file. It is also localised to a two classes and the minimum amount of extension required when adding another concrete class extending the `AbstractElement` class satisfies all further work needed.

Code block 4.1 shows an example implementation in the `ConcreteElement` class of the `accept` method. The two notable features used by the acyclic visitor pattern are multiple inheritance and run time information from the `dynamic_cast`. Although the `dynamic_cast` does have a little computational overhead associated with its use, this is outweighed the design advantages. Additionally, the `accept` method should only be used at most once per element being processed which can be seen later in the design of the other components in the order management architecture.

4.1.2 Communication Package Visitors

The next step is to apply the acyclic visitor pattern to the classes made available within the communication package in the electronic trading framework. Figure 4.2 shows the four visitor classes required in the bottom half of the diagram to support all the communication paradigms provided by the framework. One exists for each of the sub-packages within the communication package. Each of the visitor classes contains a `handle` method in place of a `visit` method which takes a single parameter of a reference to one of the concrete classes found in that package. I chose to rename the method simple to describe its function better and in less abstract terms.

One notable additional I have made to the design pattern is that the `visit` method returns a boolean value instead of `void`. This is used to enable error handling rather than using *exceptions*. Exceptions can have poor performance associated with them, even when no exception is thrown. I made the decision to avoid using exceptions within the implementation to minimise any performance impact. Although a single boolean value doesn't describe any detail apart from the success of the operation, it can be used to clean up any previous work up to the failure, making sure all data structures remain in a consistent state. With careful logging at the point of the problem being encountered, a more explicit message can be output for the user of the system to view. This also adds to the benefit of using an automatic logging monitor, which will alert the administrator immediately to any problems.

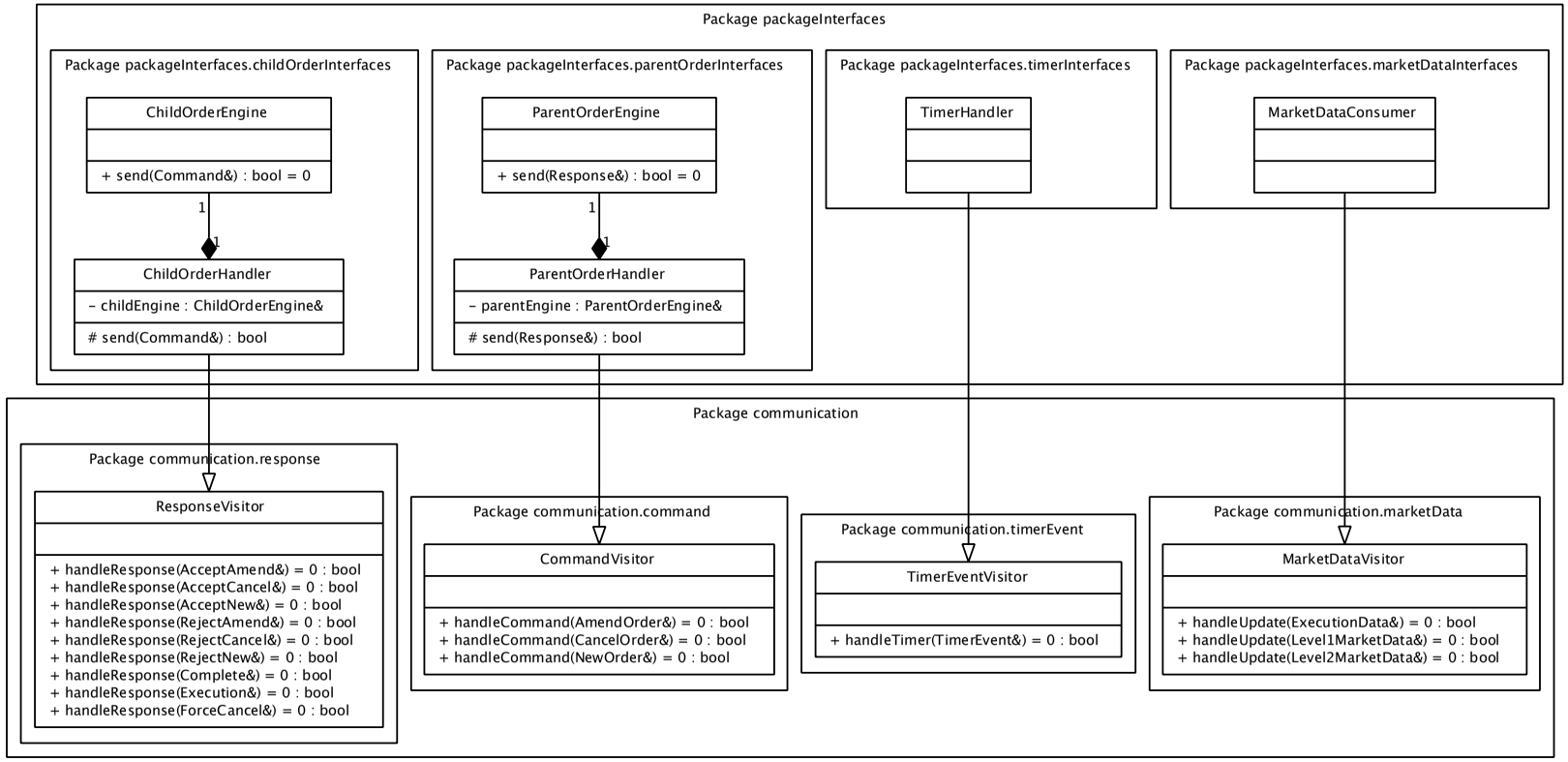


Figure 4.2: The relationship between the package interfaces and communication visitors.

Name	Summary
Parent Order Interfaces	The `ParentOrderHandler` is used for receiving commands from upstream systems to manipulate orders within the system. The `ParentOrderEngine` provides the facility of being able to send responses back to the upstream clients which are notifications of the outcome of the commands received.
Child Order Interfaces	The child order interfaces are equivalent to the parent order interfaces, except they work downstream and the communication handled by the engine and the handler is reversed. The `ChildOrderEngine` sends commands downstream to other systems while the `ChildOrderHandler` receives the responses about these commands.
Market Data Interfaces	The `MarketDataConsumer` carries out the same functionality as the `MarketDataVisitor`. There is no `MarketDataEngine` because the algorithmic trading engine doesn't produce any updates that need to be sent. If the project was later to be expanded to require this functionality then the best solution would be to create a `MarketDataProducer` class. This would implement the functionality to produce updates and control the `MarketDataEngine`. Much like how is done within the parent and child packages but with a clear separation between being able to receive and being able to send updates.
Timer Event Interfaces	The `TimerEventHandler` follows the same ideas of the `MarketDataConsumer`, it is just an abstraction of `TimerEventVisitor`. There is little need for a `TimerEventEngine` as timer events are injected at the base of the system from a schedular. If however the timer events were to be fired from a remote system, it would be beneficial to follow this design ideology.

Table 4.1: Summary of the four package interfaces and their roles.

4.2 Package Interfaces

The package interfaces provide basis for all of the trading application to be built up. They state what functionality must be supported by the application if it wishes to complete certain tasks and receive certain communications. These allow applications to be built up quickly and then implement the abstract methods specified by the interfaces. This is how the order manager is built up first, then later on the algorithmic trading engine adds further interfaces that it requires. It still inherits all of the work from the order manager though, preventing code duplication. Both the order manager and the algorithmic trading engine take parent orders and produce child orders, sending them further down the trading flow. However not all systems produce child orders, for example an exchange which matches parent orders does not split them and so doesn't require to send any child orders on or receive their responses. The same goes for consuming and producing market data updates, lots of systems consume market data, but few actually generate it.

There are two main categories of interface, a *Handler* and an *Engine*. Handlers are responsible for sending information up the application, where engines send information down the application and eventually onto another system. Only the `ParentOrderInterface` and `ChildOrderInterface` have an engine, this is because only these packages require to send information out of the system, where timer events and market data are just consumed (unidirectional). Adding these additional engines though would be straight-forward however, but isn't needed for this project. The four packages are shown in the top half of figure 4.2 with their relationships to the relevant visitor classes. It also shows that the relevant handler classes own an engine class where they exist, this determines how the system is assembled. You will notice that both `MarketDataConsumer` and `TimerEventHandler` add no further specification or implementation. I made the decision to include them for completeness and to further abstract the visitor classes away. This improves the simplicity of designing a system which extends these interfaces as they are in a single package. Table 4.1 summaries the four different package interfaces.

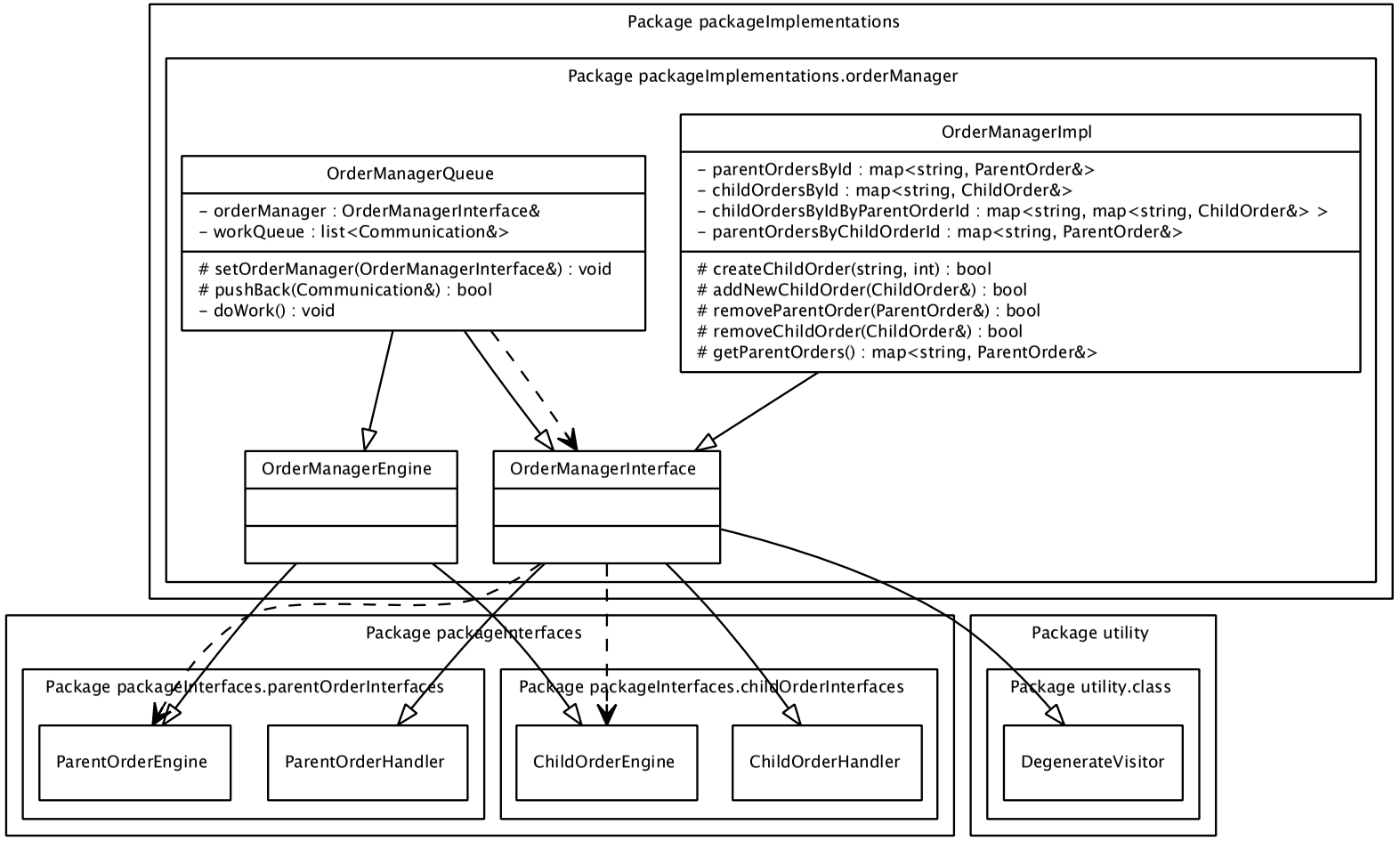


Figure 4.3: The class diagram and interactions for the order manager package.

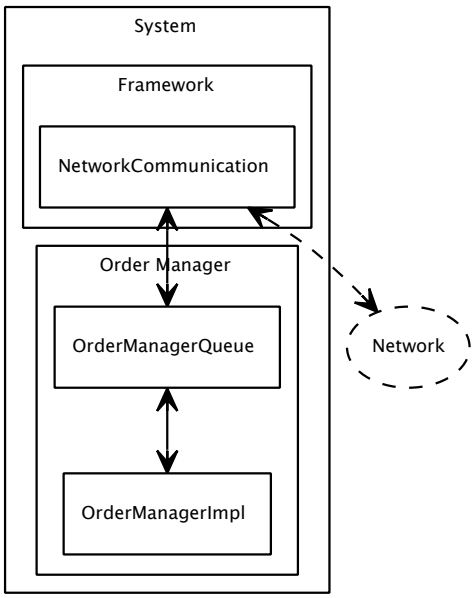


Figure 4.4: A single threaded order manager arrangement.

4.3 The Order Manager

The order manager provides functionality for maintaining orders and provides an interface for manipulating them. This manipulation is only manually at this point, a human trader would use an order manager to simplify the process of trading orders. The algorithmic trading engine automates this process, using the same interface provided by the order manager but also adding to it, enabling it to consume data from additional feeds. Therefore, I choose to develop the order manager first, which would provide the foundations for the algorithmic trading engine which would extend the order manager's functionality. I will use this opportunity to explain some of the design decisions and notable features of the order manager, then build on this for the algorithmic trading engine section.

Figure 4.3 describes the class diagram for the order manager package, I will refer back to this diagram. In the diagram I have included at the bottom, the relevant package interfaces which are extended by the classes in the order manager package. Rather than repeat all of the functionality that the interfaces specify, I have omitted this information to keep the diagram more succinct. Any new functionality that the classes add I have included. The package contains four classes, two are additional abstract classes which link interfaces from the package interfaces. They are `OrderManagerInterface` and `OrderManagerEngine`. The `OrderManagerInterface` links both the `ParentOrderHandler` and the `ChildOrderHandler`, allowing a class that extends `OrderManagerInterface` to receive commands from upstream and receive responses from downstream. The `OrderManagerEngine` links both the `ParentOrderEngine` and the `ChildOrderEngine`, extending `OrderManagerEngine` allows a class to send responses upstream and send commands downstream.

It might not be initially clear why I there is separation between the `OrderManagerInterface` and the `OrderManagerEngine` classes. The reason is because that although a class wishing to receive information must implement the relevant handler classes, it does not need to implement the engine classes to send the information, it just requires a reference to an engine that it can send its messages through. This can be seen in figure 4.3 that the `OrderManagerInterface` additionally uses a

ParentOrderEngine and a ChildOrderEngine. This is because of its extending the handler interfaces and represents possessing this reference. This is the reason why the OrderManagerImpl class only extends the OrderManagerInterface class as there are no further classes to send information up to after the implementation class, therefore it is not required to send any information down from above, as there is no above.

The overall design for the instantiations of the classes forms a ladder. Information is passed up the ladder for processing and the results are passed down the ladder for sending. This is best represented in figure 4.4, which shows the implementation class at the bottom with no further classes after it. The NetworkCommunication class hasn't been implemented but shows where the information is coming from into the system. The OrderManagerQueue's job is to buffer the information from the network before it is sent on to the order manager implementation. This is where the threading for the order manager is created, allowing for a network processing thread to take information off the network interface and place it on the work queue in the OrderManagerQueue class. In the order manager thread, work is then taken off the queue and processed. As the OrderManagerQueue needs to send information as well as receive it, it also implements the OrderManagerEngine class, as show in figure 4.4.

Cyclic dependencies are avoided by careful use of abstract class references within the concrete classes. This leads to an elegant and flexible solution. One possibility is *chaining* multiple OrderManagerQueues together, although not particularly useful, the extra buffering may result in a system able to cope better with extremely high loads, this requires further investigation. However we will later see with the algorithmic trading engine extension that chaining is more effective with the addition of another component which provides multiplexing and routing between orders.

The OrderManagerImpl provides the implementation of the order manager functionality. The class maintains all of the parent orders that it has received and any child orders that have been created from those parent order. The orders are stored in a number of abstract data types, three are *maps*, mapping *string*s to orders and one is a map from strings to another map, which maps strings to ChildOrders. All operations and transactions performed by the system must keep these containers in a consistent state. ParentOrders are stored under their own unique identifier, where each parent order will only appear once in the map. Additionally, parents are stored in another map by their child's unique identifier. This map will have duplicate values in it as it is common for a parent order to have multiple children.

Child orders are stored by their own unique identifier, like parent orders are. Additionally children are stored in a map of maps, which is referenced by their parents unique identifier first and then by their own unique identifier. This allows all of the children of a single parent order to be addressed without having to ask the parent order for those references every time. This is a trade off that I have made between speed and memory footprint, the additional storage containers decrease execution time at the expense of memory usage. This is an trade off is enabled as computer memory is relatively cheap and plentiful currently, where computation power is expensive to increase. Additionally, limits on throughput of orders processed are more likely to be bounded by computational power before limits on storage size as single order is relatively small.

The implementation provides three pieces of functionality for managing the orders it contains, createChildOrder, removeParentOrder and removeChildOrder. Only createChildOrder isn't used in the automatic management of orders. For example, if an execution comes in for an order, which completes that order then the system will automatically close and remove the order, using either removeParentOrder or removeChildOrder depending the type of the order. Use of the createChildOrder method is what will be automated in the algorithmic trading engine. Where the OrderManagerQueues implementation of all of the abstract methods for handling commands and responses was limited to passing them up to the next level. The OrderManagerImplementation has a full implementation of the logic for each of the commands and responses. Such as on receiving a NewOrder command, it adds the parent order to the storage containers and sends an AcceptNew back, assuming the order could be accepted. I won't explicitly describe the control logic for the rest of the commands and responses, as it is straight-forward from their previous descriptions. It is suffice to know that the order manager implements this logic and updates its storage containers as required, depending on the action.

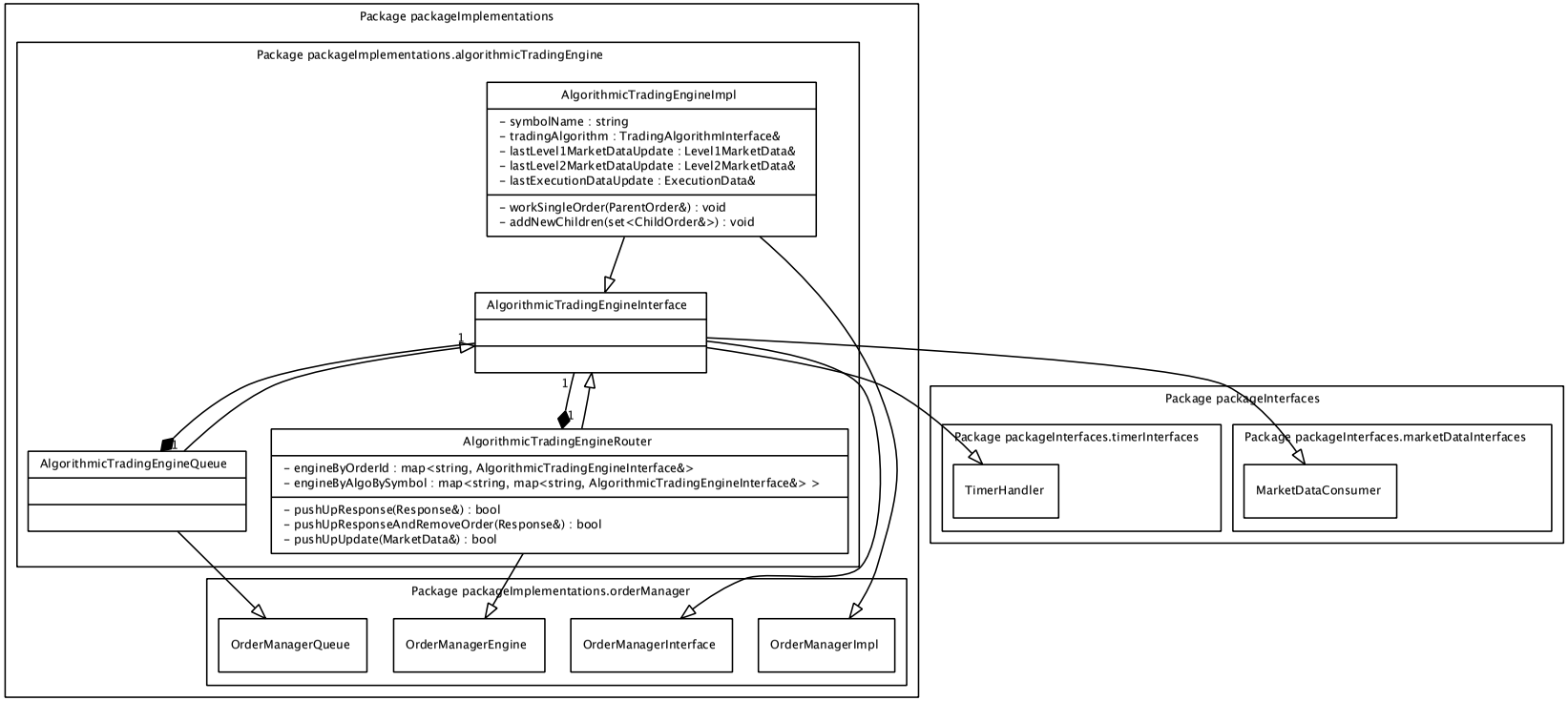


Figure 4.5: The class diagram and interactions for the algorithmic trading engine package.

4.4 Algorithmic Trading Engine

The algorithmic trading engine is an extended version of the order manager which is a novel solution, increasing code reuse while not suffering from many drawbacks. A bespoke designed system might provide a margin improvement in performance and or a slightly simpler interface. However the increase in development time, code and maintenance time required outweigh these advantages. Figure 4.5 shows the class structure for the algorithmic trading engine package. I have included the classes available in the order manager package to show which classes are extended. The additional package interfaces used that the algorithmic trading engine makes use of are included as well. The algorithmic trading engine needs to consumer market data updates, which is specified by the `MarketDataConsumer` class. In addition, the algorithms make use of timer events and so the `TimerHandler` class is included. Like with the order manager, an interface is created from which the other classes will inherit their partial specifications. The `AlgorithmicTradingEngineInterface` extends the `OrderManagerInterface`, the `MarketDataConsumer` and the `TimerHandler` to specify all of the functionality required within the algorithmic trading engine.

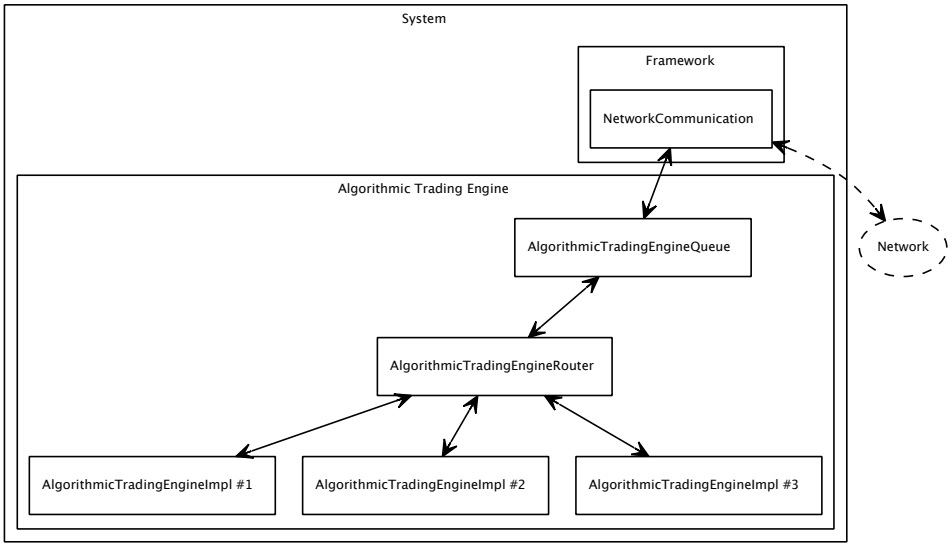
There is no need for an equivalent class to the `OrderManagerEngine` in the order manager package. This is because the additional handler classes are unidirectional in travel of information, information is only received and not sent. This means there is no requirement to have an engine to send information. The `AlgorithmicTradingEngineQueue` extends the `AlgorithmicTradingEngineInterface` and the `OrderManagerQueue`. As the `OrderManagerQueue` implements all of the functionality for the parent and child handlers this can be reused. Therefore the `AlgorithmicTradingEngineQueue` only needs to add the control logic for passing up the market data updates and the timer events. As both of them are also representations of the `Communication` class, they can be stored in the work queue in the `OrderManagerQueue`.

4.4.1 Multiplexing Information Between Instances

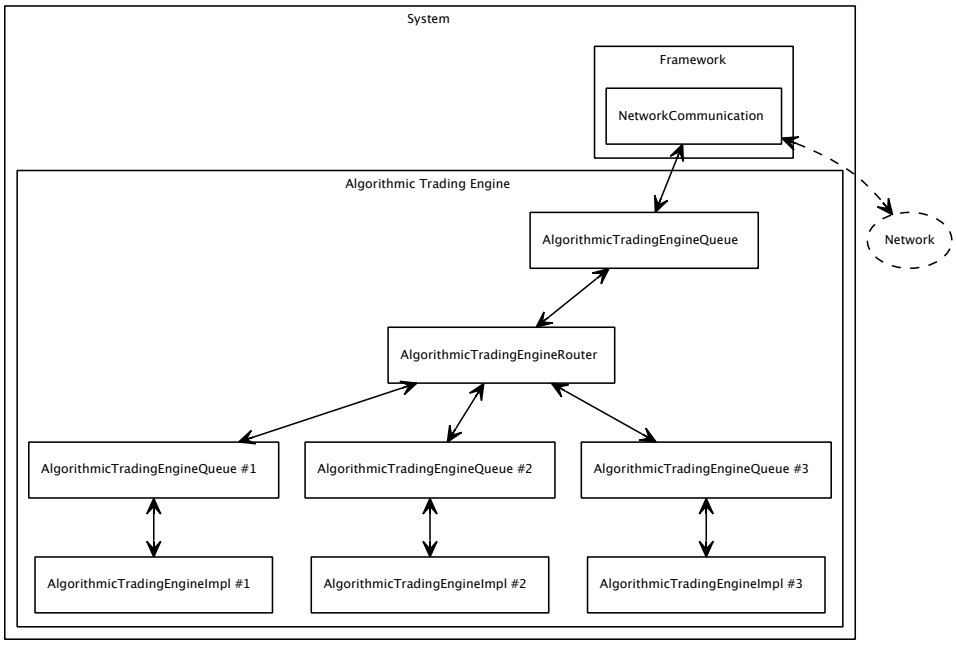
One of the large difference between the algorithmic trading engine and the order manager is that orders are required to be sorted by two additional criteria. We are now interested in what symbol the order is trading shares in, as we now need to know what market data updates are relevant to that order. Additionally we need to know what algorithm is trading an order so that the order is given to the correct algorithm. One possible solution would have been to extend the `OrderManagerImpl` class, adding additional storage containers which would implement these extra sorts. However large amounts of the functionality for handling the varying commands and responses would have to be rewritten to cater for these extra containers, which is undesirable.

My solution is to implement a class whose purpose is to sort the information that it receives and pass it onto the correct algorithmic trading engine implementation (order manager). Instead of a single order manager existing, now multiple instances are created, one for every combination of symbol being traded and algorithm available. For example, if there were three trading algorithms available and orders were accepted that trading over five different symbols, then there would be 15 instances of the order manager running. This is implemented in the `AlgorithmicTradingRouter` class. It maintains two storage containers, one which maps order managers by symbol and by algorithm being used and another which maps order identifiers to order managers. The reason for the first container is clear and allows access to the order managers for new information, such as market data updates and new orders. The second container exists to speed up finding an order manager for repeat or previously existing information, such a responses to child orders and amend or cancel commands for parent orders. This is again part of the memory footprint and execution speed trade-off described earlier.

Now with the current design there is a large benefit that can quickly be reaped and could lead a sizeable increase in performance. This is best explained as a diagram, figure 4.6a shows how the original configuration of the objects would be, which is based on the order manager's configuration discussed and show in figure 4.4. However because of the abstract classes used within the concrete classes, instead of directly connecting the implementation to the router, we can place another queue object between them, as shown in figure 4.6b. The extra layer of buffer may not help much but as the queue implements the threading, it now enables all of the order managers to run in their own, independent thread. This has obvious benefits on multi-processor or multi-core computers. Another benefit over performance is fairness in the system. Now with the operating system controlling the scheduling more directly, less



(a) A single thread for the whole of the algorithmic trading engine process.



(b) An individual thread for each algorithmic trading engine implementation.

Figure 4.6: Threading levels of the algorithmic trading engine.

active order managers should not have to wait as much as if they are in the same thread as a very active order manager. The theory behind this is that on the single threaded implementation, if a symbol receives ten updates in a row, then these must be processed before any other symbols updates can be. Now with the multi-threading setup and the routing, these ten updates can be passed to the correct thread, while other the thread now receives its update sooner.

4.4.2 Handling Market Data Updates and Timer Events

Like with the rest of the algorithmic trading classes, the order manager needs to be updated to handle market data updates and timer events. The `AlgorithmicTradingEngineImpl` class extends the `AlgorithmicTradingEngineInterface` to add the extra handlers and the `OrderManagerImpl` class to provide the functionality for the management of the orders. Little of the inherited functionality requires modification. The only additions are when a parent order is created or amended. When either of these actions occurs, we want the order to immediately be put through the algorithm, to see if it can be traded. For this reason, the implementation maintains a record of the last market data update seen, as this will be required to be provided with the order to the algorithm. This is because all of the algorithms are stateless, they rely on the order maintain its own state of how it has been traded and being provided with the latest information. It is desirable for an order to be tested immediately because on illiquid stocks (those that receive little activity), there may be a long duration between market data updates being received. If the order was only traded on a market data update, this would mean there would be a long period of time before the order came up for trading, possibly missing out on some liquidity available on the market when it arrived.

The new functionality added to the class handle receiving market data updates or timer events. On either of these situations, the order manager submits this information, with the details of the orders that it is currently managing to the algorithm that it is possess. In return the algorithm provides a list of child orders to create, their size and any additional changes to their parameters that should be made. The manager then carries out the task list, creating the child orders using the `createChildOrder` functionality provided by the `OrderManagerImpl` class and sends them downstream. Additionally, if the event as a market data update, then the relevant reference to the last seen update is updated with the new one.

4.4.3 Calculating Market Metrics

Some algorithms require information and data which is calculated from information that is received by the system or some historic data. I choose to group the two requirements into a separate subsystem. The first job of the system is to provide access to the information that it carries, this may include current information and historic information. I won't deal here with how historic data is produced, but this system could be used in that process by recording its current information through the trading day. The other activity is that it receives all of the market data updates and calculates the various metrics required. The information can be accessed from anywhere within the system as the class is a singleton with a `static` access method. This is also partly due to stop multiple instances from existing and calculating the same equations. The class is plugged into just the market data update feed and receives all of the updates that the main system does as well. To prevent the system from using old data, the metric calculated always gets the update before the main system. This is so that if it requires information from this object, the latest information has been calculated and can be returned. On old information the algorithm may not trade when it should have done or conversely, traded at the wrong time. The metrics that are calculated are explained later in the report when I describe the three concrete trading algorithms that I have created. Further work to improve this module could include abstracting the calculation away from the main class and allowing it to store a collection of calculators. A calculator could then be specified with a trading algorithm and then only a calculator needs to be added to this class, rather than modification of its internal representations.

4.5 Summary

In this chapter I have presented my design and implementation for an algorithmic trading engine which is based upon an equivalent order manager. The two systems are built up upon a collection of interfaces, which themselves are built from a set of interfaces which provide the ability to distinguish between different communication classes. These interfaces form what I term as a design pattern for building electronic trading systems. They can be used in different configurations depending on the activities and behaviours of the system being implemented. This has the concrete example in what I have produced here to show their use.

The design pattern heavily depends on the use of the *acyclic visitor pattern* to ensure that no cyclic dependencies are introduced into the implementation. This carries on the theme of guaranteeing that the code produced is of the highest quality and the maximum flexibility. As previously mentioned that these interfaces can be used in further project and systems, so can the order manager and the algorithmic trading engine. Many electronic trading systems such as those dealing with high frequency trading and statistical arbitrage could easily be tailored to being build upon either system. Allowing the time from conception to deployment to be heavily reduced.

Further still, through the clever arrangement and design of classes, I have provided a simple and effective method of coarsely threading the order manager. This is echoed and amplified in the algorithmic trading engine, which can provide a simple thread to each algorithm independently. Not only leading to increased performance but additionally increased fairness. Both of which are very important traits desired of an electronic trading system. Finally, I have introduced new concrete concepts on how orders are managed and the actions performed on them to manipulate them. This will allow the reader to fully comprehend the decisions made in the next chapters dealing with implementing the trading algorithms that this algorithmic trading engine will use.

Chapter 5

Trading Algorithms in Software

In this chapter I will present my trading algorithm design pattern for algorithms implemented in software. These algorithms are supported by the algorithmic trading engine and the electronic trading framework which I have previously discussed. Building on this design pattern, I present my implementation for threshold trading functionality. This is designed to get all algorithms extending this pattern a base increase in price performance which will help them to become more effective. Finally, I will present three concrete trading algorithms that I have created to demonstrate these principles and show how quick and simple it is to rapidly prototype a new algorithm given the work provided here. This chapter completes the actual software implementation of the application and secondly provides the specification for the hardware implementation to follow. Specifically, this chapter will cover the following:

- The interface that I have designed to facilitate communication between the trading algorithms and the algorithmic trading engine. I'll also show how the algorithms are managed in such a way that it lends itself to them being either in software or hardware; in section 5.1.
- I will now present my design pattern for creating trading algorithms and the functionality that it provides. This will include how threshold trading works and how I decided to evaluate the favourability of the market at trading time; in section 5.2.
- Using this design pattern, I will provide three concrete example trading algorithms with their models and specifications that I have produced. I will also cover what opportunities it gives the developer and show case its flexibility; in section 5.3.
- I will then wrap up with the benefits from the design and implementation, specifically looking at what they afford to the user. Such as how the trading algorithm design pattern can be used for future implementations and research; in section 5.4.

5.1 Trading Details and Algorithm Interfaces

It is important to know how the algorithms are built and the interface with algorithmic trading engine. This interface is shown by figure 5.1, depicting the class structure of the relevant packages. I have included the `AlgorithmicTradingEngineImpl` class and its package, this shows that each implementation possess an algorithm that it solely uses for trading. The implementation receives this algorithm from the `AlgorithmManager` and then maintains a local reference. This can be seen in the middle package of the diagram. This forms the *abstract factory design pattern*, in which the `AlgorithmManager` is a *factory* class. It is solely responsible for the creation and assignment of algorithms. A simpler solution would have been to allow the *AlgorithmicTradingEngine* to create its own instance of an algorithm to use. This would have resulted in a tighter coupling between the packages. As we want to use the same algorithmic trading engine when using the trading algorithms in hardware, this would have resulted in more code to control the creation of those interfaces. This also answers the question of why the abstract factory pattern is used, this is because we will require a factory for the software algorithms and one for the hardware algorithms. At the moment, the choice of which factory is used is dependent on the parameters

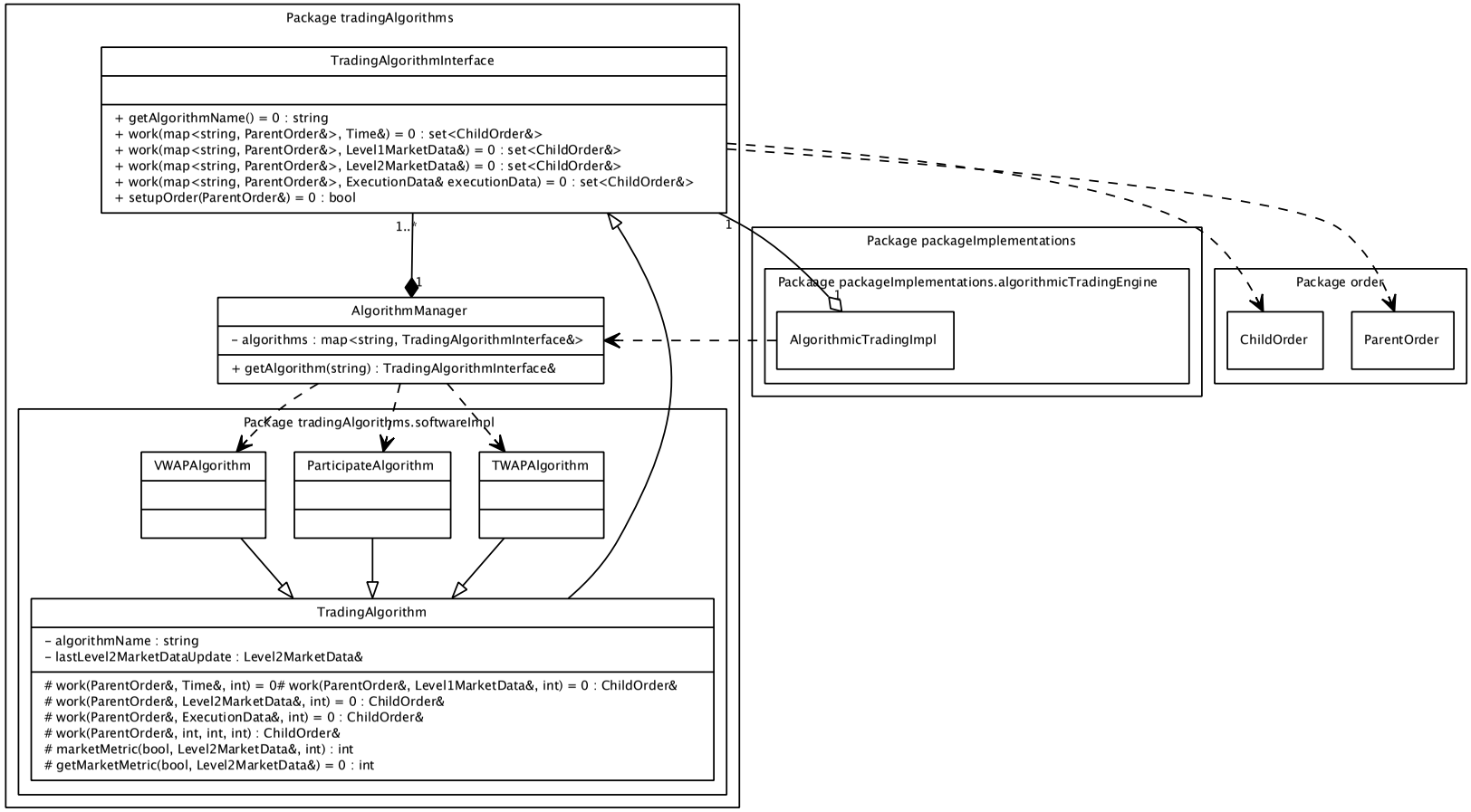


Figure 5.1: Class diagram of the algorithms class hierarchy and interactions.

of the system at start up. Although a *factory* for the hardware algorithms is bit of a misnomer however, it will have to create and manage the interfaces to the hardware algorithms, instead of creating them.

Now that algorithms can be managed and assigned, we need to know what interface an algorithm must provide so that the algorithmic trading engine can use it to trade automatically. This is provided by the `TradingAlgorithmInterface`, the main methods specified are a collection of `work` functions. These all take a map of the parent orders to process and an event to process them against. These events are either timer events or the three different forms of market data update. In addition, the interface must provide an unique identifier in the form of a name and additionally, a method to setup a new order. This last method is critical and is used when a new order is received. It makes sure that the order being setup has the correct state for when it was accepted and then tries to trade the order initially. This class and methods can be seen at the top of figure 5.1. Although the `AlgorithmManager` knows about the different algorithm implementations, this is only during creation, algorithms are otherwise addressed through their interface. At a later stage, a plug-in manager could be designed allowing for algorithms to be quickly deployed and added to the system without the whole system requiring recompilation. This still wouldn't be possible at run-time though, as to add a new algorithm the whole application would have to be bought down and started again.

The order trading details provide a storage container for an algorithm to maintain the orders state. This is shown in figure 5.1 on the right. There is an abstract base class, `AlgorithmicTrading` which extends the `OrderTradingDetails` class, allowing the order to store it. If orders were being manually traded, there would be an associated class to the `AlgorithmicTrading` class which would store the trading details for manually trading the order. Each of the concerete algorithms implemented has their own extension of the `AlgorithmicTrading` class. This maintains the parameters and state that the algorithm is interested in when trading an order. The base `AlgorithmicTrading` class supplies all of the information for the algorithm trading pattern functionality to perform its duties.

5.2 An Algorithm Design Pattern and Threshold Trading

I'll now present my trading algorithm design pattern for the algorithmic trading engine. This allows for quick development of additional algorithms by implementing the common functionality for this class of algorithm. In additional it layers out a specification that the programmer must implement in the algorithm, providing them with the flexibility to carry out any sort of trading they wish. The concrete algorithms that I will investigate are similar, they differ in two directions, the price metric used for comparison when calculating the market favourability and the event that an order is traded on.

All of the algorithms can trade a basket of orders, each order then traded independently. The iteration through the orders to trade is handled by the base algorithm being common functionality. The `TradingAlgorithm` abstract base class implements this functionality and specifies the interface for the concrete algorithms. These extend the `TradingAlgorithm` class and implement the interface. Most of the interface is within the `private` scope, this is because the `TradingAlgorithm` class will call up to its concrete implementation to use that functionality. The `TradingAlgorithmInterface` provides the `public` face of an algorithm to the user. The basic *TradingAlgorithm* does nothing on receiving any event to the orders it has to trade. In fact because of the implementation, the method returns a null smart pointer of a child order, which the receiver can check the validity of before carrying out its operations. Concrete algorithms override this behaviour, but if they decide that they are not interested in the event then they can call down to the base class to carry out the no operation.

The main attraction of using this design pattern is that it supplies with it the *threshold trading* functionality. Its purpose is to try to improve the price of the complete traded when being compared against a calculated price metric. The current market state will be evaluated against this price metric, if it is found to be favourable then the amount of the order traded at that point should be increased. However if it is unfavourable then the amount to be traded should be reduced. This is kept in check by boundaries preventing the amount of change in the quantity traded becoming more than a specified percentage. This is required because if the market suddenly became very favourable, then the algorithm might be tempted to complete the order very quickly. However over time, the price might become even more favourable and if the order has already completed then the price achieved won't have been the best possible. The converse is true, in that the favourability might decrease, but we don't want to stop trading because it might become even more worse. There is a trade-off between working quickly and

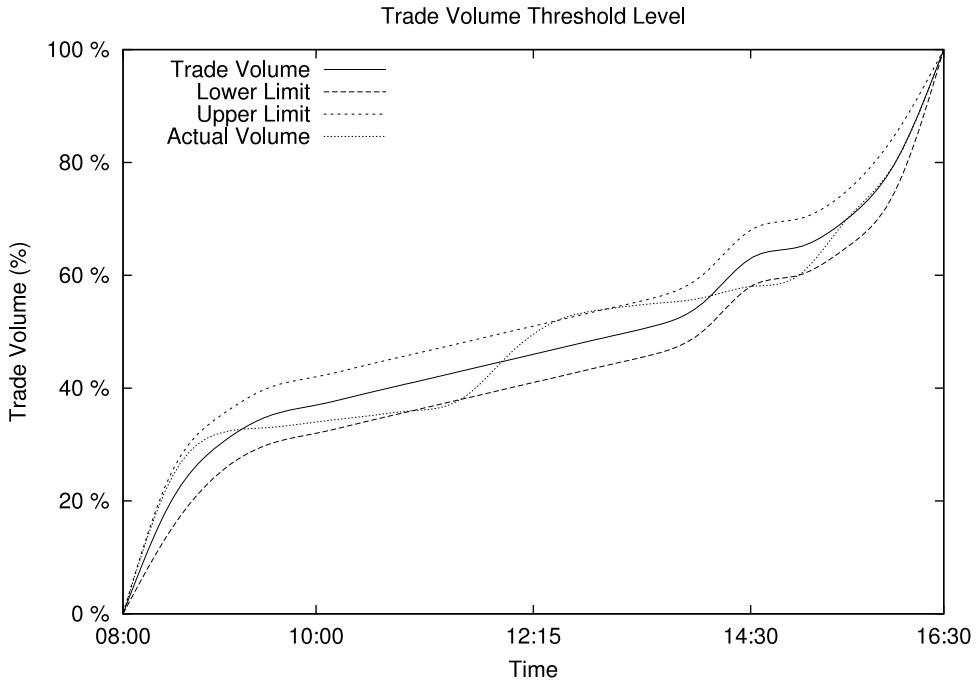


Figure 5.2: A threshold trade over a complete trading day with its boundaries and target.

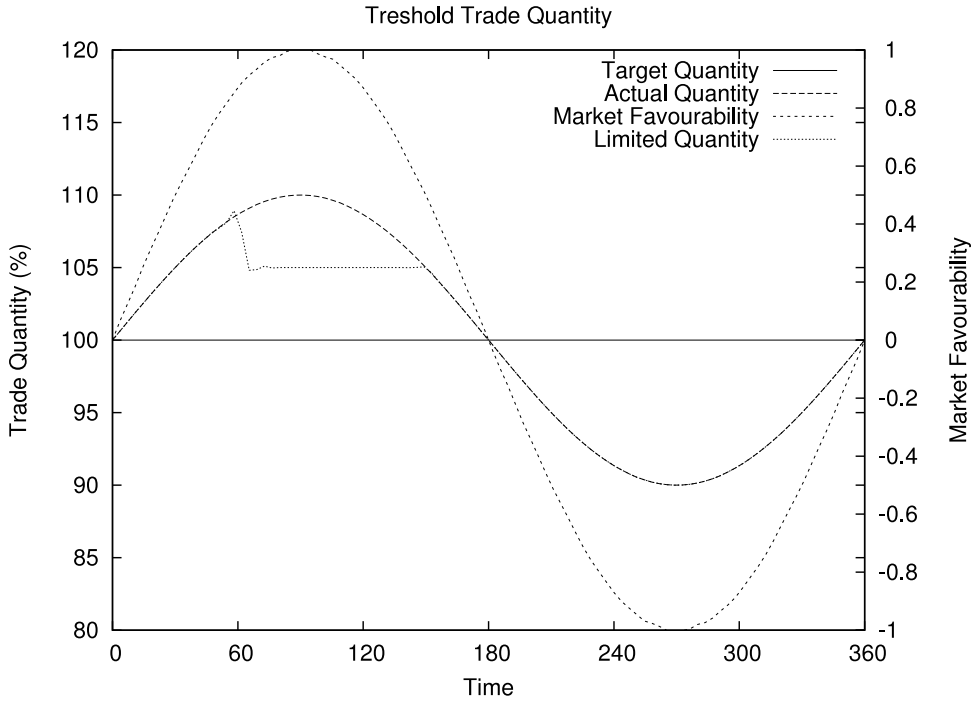


Figure 5.3: Order trading quantity when threshold trading with market favourability.

```
1  int TradingAlgorithm::marketMetric(bool side, shared_ptr<Level2MarketData> level2MarketData,
       int priceMetric) {
2    int result = 0;
3    bool originallyFavourable = side ? level2MarketData->getBegin(side)->get()->getPrice() <
         priceMetric : level2MarketData->getBegin(side)->get()->getPrice() > priceMetric;
4
5    int numberOfEntriesUsed = 0;
6    for (vector<shared_ptr<Level2MarketDataEntry> >::iterator iter = level2MarketData->getBegin(
         side); iter != level2MarketData->getEnd(side); iter++) {
7
8      bool currentlyFavourable = side ? iter->get()->getPrice() < priceMetric : iter->get()->
           getPrice() > priceMetric;
9
10     if (originallyFavourable && !currentlyFavourable) {
11       break;
12     }
13
14     if (side) {
15       result += (priceMetric - iter->get()->getPrice() == 0) ? 100 : (100 * priceMetric) / (
             priceMetric - (priceMetric - iter->get()->getPrice()));
16     } else {
17       result += (iter->get()->getPrice() - priceMetric == 0) ? 100 : (100 * iter->get()->
             getPrice()) / (iter->get()->getPrice() - (iter->get()->getPrice() - priceMetric));
18     }
19
20     numberOfEntriesUsed++;
21   }
22
23   return result / numberOfEntriesUsed;
24 }
```

Code Block 5.1: Calculating the market favourability of an update against a price metric.

slowly, it is expected that trading over a longer period of time will result in a more consistent price which is closer to specified metric price. Maintaining a price which is better than the price metric means that the trade was more successful than the rest of the market that day, when performance is measured against that metric.

Figure 5.2 shows an order using threshold trading and the boundaries that it must remain in, the upper and lower limits. Figure 5.3 shows another view of this process, the solid line is the target child order quantity, at every child order trade we should try to trade 100% of the calculated value to trade. The dotted line shows the current market favourability, varying between 1 and -1. Finally, the dashed line shows the quantity on the child order, taking the market favourability under consideration. It shows that while the market favourability is positive, the child order quantity increases to take advantage over this. If the favourability is negative, then the child order quantity is reduced. In this example, the traded volume is the same for both the target and actual. However the price achieved by following the actual quantity should be better than the final price metric at the trade completion. The final line that rises before being cut off in the positive part of the sine wave, shows the use of trading boundaries. These are designed to further restrict the order quantities difference from the target quantity. They work against the total amount of the quantity the order has traded. In this example, the order trades ahead at the beginning, but reaches the limit, then carries on at the maximum before dropping again as the favourability does. When it comes to trading behind when the market is unfavourable, because the order is already ahead, it never reaches its lower limit, so the change from the target quantity is never restricted. Looking back at figure 5.2, the actual volume line which varies over the trading day, shows these principles. As the line is initially above the target trade volume at the beginning, we know the market was favourable then. Later, past 10:00AM, the market is unfavourable and the order slips back under the target value before trading ahead again past midday. This occurs until either the trade is complete or the end of the trading day arrives.

The market favourability is calculated from the last level 2 market data update seen, which contains the details of all of the orders available on the exchange. We are only interested in the information on the opposite side of the market that our order is on, as it is these orders we could trading with. The

information from the update is provided sorted, from the first entry we will be able to tell whether the market is going to be favourable or not. If the first price is less than the price metric, then the market will be favourable, otherwise not. What we need to calculate now is how favourable or unfavourable the market is. If the market is initially favourable then we will only consider entries that are favourable. For example, if there are three entries better than our price metric, those will be factored into our calculation, the rest will be ignored. However if the first entry is as unfavourable then all entires will be considered. The reason why unfavourable entries are ignored when the market starts as favourable, is because the market is favourable and taking unfavourable entries into consideration will reduce the market favourability.

Code block 5.1 shows the implementation to calculate the market metric. The three parameters provide the trading side the order, the level 2 market data update and the price metric. All of the algorithms shown later use this functionality, they can provide their own price metric to differentiate themselves from others. The market is initially evaluated for favourability on line 5. Then each entry in the update is iterated over from line 13, favourability is examined again and if the market starts favourable and becomes unfavourable, the iteration is ends on line 21. Lines 26-34 calculates the incremental market metric and finally the average result is taken and returned on line 38. You may notice that integers are used to represent the numbers used. This is because I want to directly compare the software implementation against the hardware. The hardware implementation that I will be studying initially will not support floating-point arithmetic. This solution provides a simple method of calculating a market favourability measure. There are many more complex functions that could be used in its place. However that is the scope of a financial based project.

5.3 Algorithm Design and Implementation

In this section I will cover the three concrete algorithm implementations that I am presenting in this project. All of them are derived from an abstract base class which implements all shared functionality. This allows the concrete implementations to contain their specific equations and control flow. Figure 5.4 outlines the storage containers that each of the algorithms require on orders they trade. They store the state of the order as it passes through the algorithm and additional parameters that the algorithm requires. The `AlgorithmicTrading` class provides all of the parameters to allow the order to threshold trade and so if a new algorithm is created which utilises this design pattern then its relevant trading details class should extend `AlgorithmicTrading` class. Each algorithm using differs the way that it calculates the orders the difference in what it has traded and what it should traded. This requires a complex equation that depends on the algorithms trading the order.

The algorithms share some control flow, such as the many checks that are required to prevent illegal child orders being sent to market. The `AlgorithmicTrading` class stores these attributes as well, such as *minimum* and *maximum tranche size*. These attributes decide the range of quantities that a child order is allowed. If a child order is below the minimum tranche size then it will not be traded, if it is above the maximum, then its quantity will be reduced to the maximum amount allowed. Simpler checks are also included, such as the child orders size being positive and less than the remaining quantity on the order. If the child orders quantity is greater than the remaining quantity, it is reduce the remaining quantity. The only exception on the minimum tranche size limit, is when the remaining quantity is less than the minimum tranche size, then a single order of the remaining quantity is allowed to be sent to market, bypassing this restriction. The three algorithms that I have provided for this investigation follow here. I have started with the simplest, the participation algorithm. In the later algorithms, where they share features of the participation algorithm, I have stated but omitted these details to avoid repetition.

5.3.1 The Participation Algorithm

$$AP = \frac{\sum_{t=1}^{n} P_t}{n} \tag{5.1}$$

Participation uses the average price of executions on the market as its price metric for calculating the market favourability with. This it provides to the market favourability calculator, which returns the market favourability. Equation 5.1 shows how the average price is calculated. Where AP is the average

price, P_t is the price at time t and n is the number of prices seen since the beginning. After calculating the market favourability the order can try to trade. If the market is unfavourable then the order will automatically trade nothing if being process on a level 2 market data update. Trading behind when the market is unfavourable and the event processed by the algorithm is an execution notifications from the market. If the market is favourable then we need to calculate the difference between what the order has traded and what the order should have traded. If the difference is positive then the order is trading ahead, otherwise it is trading behind. This difference is used to ensure that the order does not exit its trading boundaries, even if the market is very favourable.

$$\Delta_{Traded} = \Big(\sum_{Children} Q_{Child} \Big) - \Big((V_{Current} - V_{Initial}) \times R_{Participation} \Big) \tag{5.2}$$

Equation 5.2 shows how the difference in traded values is calculated, represented by Δ_{Traded}. Q_{Child} is the quantity of a child order and the sum is taken across all children. $V_{Current}$ is the current traded volume in the market for today, $V_{Initial}$ was the traded volume when the order accepted, at the start

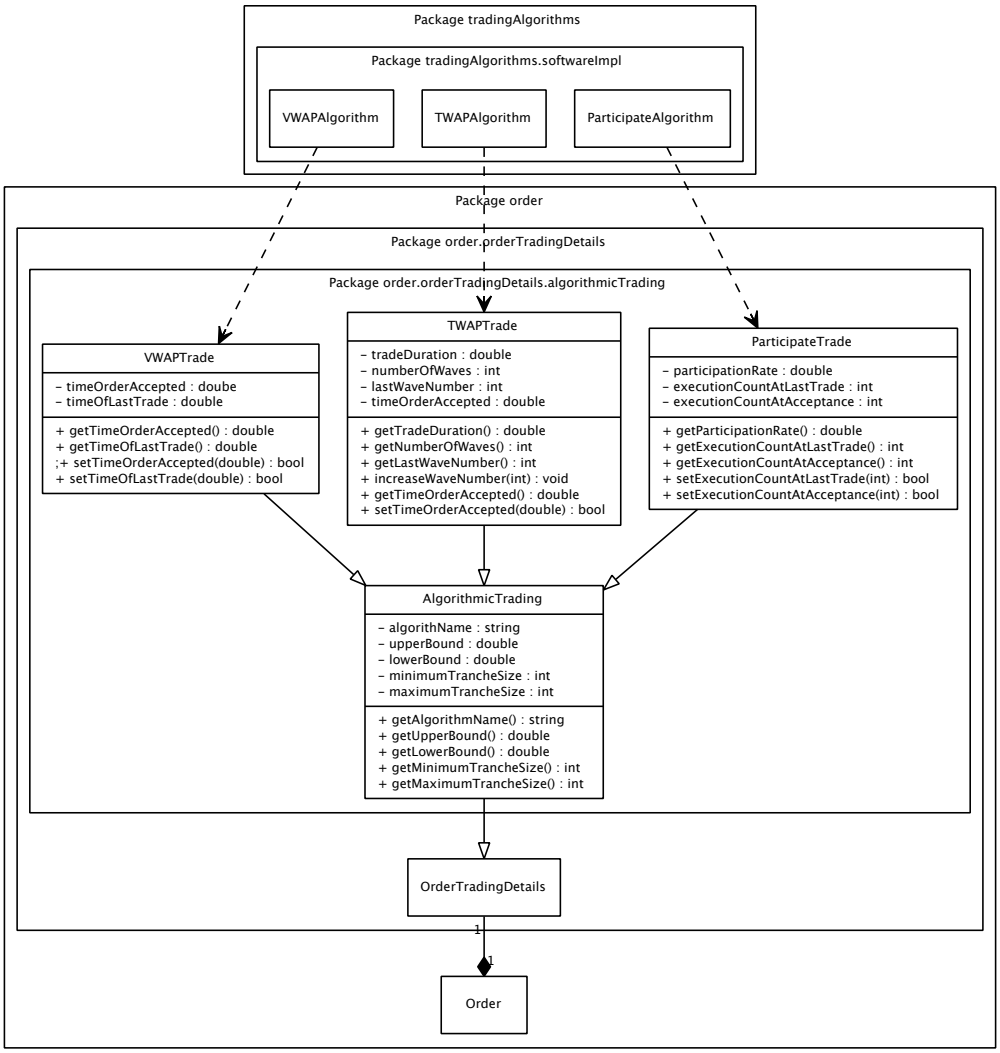


Figure 5.4: Structure diagram of the order trading details enabling algorithmic trading on equity orders.

of trading, $V_{Current}$ and $V_{Initial}$ will be equal. $R_{Participation}$ is the rate of participation the algorithm is attempting to meet as a percentage. This with the parent order, the market metric and the upper trading ahead boundary are passed to the abstract algorithm, which completes the decision to trade or not. This includes all the checks to ensure the child order volume is valid.

$$Q_{NewChild} = (V_{Current} - V_{Last}) \times R_{Participation} \tag{5.3}$$

The second event that the participate algorithm trades on is a notification of an execution on the exchange. This arrives in the form of an execution data update from the market data feed. The first calculation performed is to find out the quantity of the child order that we are going to try to create, this is found by equation 5.3. Where $Q_{NewChild}$ is the quantity of the new child order to be created, $V_{Current}$ is the current market volume, V_{Last} was the market volume last at the last trade and $R_{Participation}$ is the participation rate.

The difference in traded volumes is then calculated with equation 5.2. This is used to prevent the order exiting the bottom trading boundary when trading behind. If the market is favourable then we traded the full amount calculated, $Q_{NewChild}$. Otherwise we reduce this amount proportionally to how unfavourable the market is currently, to a minimum which is the amount the order is allowed to trade behind. The amount to trade is then updated by the amount to trade behind. This value is then checked against all of the parameters and modified accordingly and the order is only traded if it is greater in size than the *minimum tranche size*.

5.3.2 The Time Weighted Average Price Algorithm

$$TWAP = \frac{\sum_{t=1}^{n} P_t(T_t - T_{t-1})}{T_n} \tag{5.4}$$

The *time weighted average price* is used as the price metric when calculating the market favourability in the TWAP algorithm. Equation 5.4 shows how this is calculated. Where $TWAP$ is the time weighted average price, P_t is the price at time t. T_t is the current time and T_{t-1} is the time of the previous price update. T_n is the total time that has elapsed and the sum is taking across all values. Like with the participate algorithm, we also require an equation to calculate the difference in traded values. This is different between algorithms as the amount they should have traded is different depending on their trading style.

$$\Delta_{Traded} = \Big(\sum_{Children} Q_{Child}\Big) - \Big(\frac{W_{Last} \times W_{Total}}{Q_{Original}}\Big) \tag{5.5}$$

Equation 5.5 shows this calculation for the TWAP algorithm. Where Δ_{Traded} is the difference in traded volumes, Q_{Child} is the child quantity and the sum is taken across all children. W_{Last} is the last wave number that was traded and W_{Total} is the total number of waves that are going to be traded in this order. Finally $Q_{Original}$ is the original quantity on the parent order. The market metric is calculated by the abstract algorithm and the $TWAP$ metric, combined with the Δ_{Traded} which allows the order to be traded ahead on a favourable market. All of the same checks on the calculated order quantity are carried out before the volume is submitted.

$$T_{Next} = T_{Accepted} + \Big(\frac{W_{Last} \times T_{Total}}{W_{Total}}\Big) \tag{5.6}$$

The participation algorithm traded on execution updates, the TWAP algorithm trades on receiving timer events. These notify it that a period of time has elapsed and the algorithm will check to see if it should trade. If enough time has past since the last wave was sent to market then we can trade. Therefore $T_{Current}$ must be less than T_{Next}. T_{Next} is calculated by equation 5.6. Where $T_{Accepted}$ is the time the order was accepted, W_{Last} and W_{Total} are the same as before. T_{Total} is the total time the order is going to be traded over. For example, T_{Total} would be two hours if the owner of the order wanted it traded evenly and completed in two hours.

$$Q_{NewChild} = \left(\left(\frac{T_{Current} - T_{Accepted}}{T_{Total} \times W_{Total}} \right) - W_{Last} \right) \times \frac{Q_{Original}}{W_{Total}} \tag{5.7}$$

If it is time to trade, then we can calculate the new child orders quantity, show by equation 5.7.Where $Q_{NewChild}$ is the quantity of the new child order, $T_{Current}$ is the current time and $T_{Accepted}$ was the time the order was accepted and started trading. T_{Total}, W_{Total}, W_{Last} and $Q_{Quantity}$ are the same as before. As with participation, we check to see if the amount should be reduced because of an unfavourable market. For this we need the market metric and difference in traded volumes as described before. This is the factored into the proposed volume of the new child order. If all of quantity checks are passed after the market favourability weighting is taking into account then the child order can be created and sent.

5.3.3 The Volume Weighted Average Price Algorithm

$$VWAP = \frac{\sum_{t=1}^{n} P_t V_t}{\sum_{t=1}^{n} V_t} \tag{5.8}$$

The price metric for the VWAP algorithm is the *volume weighted average price*. This takes into account the size of an execution, rather than just its price or the time spent at that price. The VWAP can be found by using equation 5.8. Where P_t is the execution price at time t and V_t is the volume of execution at time t. The sum is taken across all values provided. The control flow of the VWAP algorithm is similar to that of the TWAP algorithm.

$$\Delta_{Traded} = \left(\sum_{Children} Q_{Child} \right) - \left(\frac{(V_{Last} - V_{Initial}) \times Q_{Original}}{1 - V_{Last}} \right) \tag{5.9}$$

During threshold trading we also require the difference in traded volumes which is found with equation 5.9. Where Δ_{Traded} is the difference in traded volumes, Q_{Child} is the quantity on a child order with the sum taken across all children. V_{List} is the market volume at the previous trade as a percentage through today and $V_{Initial}$ is the market volume when the order was first accepted. $Q_{Original}$ is the initial quantity of shares on the ordered, as the market volumes are represented as percentage (fractions between 0 and 1), 1 is used within the equation as a normalising constant. The VWAP algorithm uses historic data to help make its decisions. This is why $V_{Current}$ and $V_{Initial}$ are represented as percentages of the progress through the expected daily volume. This value may be inaccurate depending on how the days activity differs from the historic value. If it is less then expected, then the order may trade more slowly and therefore possibly not finish before the end of the trading day. Conversely, if it is more, then the order may trade more quickly, completing earlier. The child order amount is then submitted to the normal round of checks.

$$Q_{NewChild} = \left(\frac{(V_{Current} - V_{Initial}) \times Q_{Original}}{1 - V_{Initial}} \right) - \left(\sum_{Children} Q_{Child} \right) \tag{5.10}$$

Like TWAP, VWAP trades on timer events. Unlike TWAP, we care about the amount of time that has elapsed, just that the expected market volume has increase sufficiently to allow us to trade. This allows VWAP to trade more during the busy periods of the trading day (the morning, mid-afternoon and before the close of markets) in the hope of achieving a better price which is closer to its price metric. Equation 5.10 shows how the initial amount to be traded is calculated. Where $Q_{NewChild}$ is the quantity that the new child order will have and $V_{Current}$ is the current market volume as a percentage through the trading day. All of the remaining variables have the same definition as before. You may notice that equations 5.9 and 5.10 are nearly identical, except for the minus operators arguments switching sides and the use of $V_{Current}$ instead of V_{Last}. The operand switch is because we now want to calculate the difference between what we should have traded compared to what we have traded. We use the market favourability metric to reduce the child order quantity if the market is currently unfavourable like the other algorithms. If the value is larger than the minimum tranche size specified then we can send another child to market. The proposed child order quantity is then submitted to the rest of the checks to prevent illegal trading.

5.4 Summary

In this chapter I have presented an trading algorithm interface which would allow any application which can communicate over this interface to utilise the power of these algorithms provided. The interfaces builds upon known design patterns, such as the *abstract factory* pattern to create an environment with suitable abstracts that mean that the algorithmic trading engine has no knowledge of the algorithm that it is using. This is important so that swapping between software and hardware implementations is seamless and can be accomplished with a parameter change. This would also afford the ability for a system to support systems with and without reconfigurable hardware attached by automatically detecting the presence of any hardware and selecting the optimum profile of algorithm implementations to use.

As the design of algorithms is difficult, to speed to process up from design to implementation I have created a trading algorithm design pattern. This allows a developer to use selected and powerful functionality which can be individually specialised. Additionally they have full power over the algorithm to implement any number of complex features that are required. This gives incredible flexibility to the platform and allows it to support a huge number of algorithms. This gives clients greater flexibility of algorithms to choose from which would make the system a more attractive proposition to use. This should attract customers to the operators of the system and increase operating profits.

To confirm the success of the trading algorithm design pattern I have created three concrete examples using these methods, the *Participation*, *TWAP* and *VWAP* algorithms. I have created the specification for these algorithms and provided the mathematical models used by them to calculate their state and make their decisions. These specifications aren't available anywhere else and I have had to develop the logic from scratch to be able to show these. They are all based on industry standard approaches, meaning that I have not had to spend large amounts of time generating trading ideas, however this is still a considerable piece of work. I have shown ideas on how the trading algorithm design pattern can be utilised and of course, these three specific algorithms themselves can be extended and modified to suit clients better. The algorithms provide a basis for research into their performance from a technical aspect and will be used as a benchmark which the hardware implementations must eclipse to warrant further research into the area.

Chapter 6

Trading Algorithms in Hardware

With the software implementation of the algorithmic trading engine and its software trading algorithms complete, we can move onto implementing the specification described before in reconfigurable hardware. Rather than repeating the specification again, I will concentrate on the problems that arose during the translation from software to hardware and the designs that I put in place to speed up development time when working with reconfigurable hardware. Some of these solutions are general to problems or optimisations in reconfigurable hardware and have many other possible applications outside of electronic trading systems. Specifically, this chapter will cover:

- The limitations to programming for reconfigurable hardware and my novel solutions to these problems. I will also cover the mind set that is required when programming reconfigurable hardware to ensure the best possible code and how it differs from programming software; in section 6.1.
- I will present my hardware architecture for trading algorithms, taking many cues from the software algorithm design pattern in the previous chapter. I will cover in detail new process in algorithms which optimise for parallel computation, such as my array averaging function; in section 6.2.
- It is useful to understand the different layers of parallelism available in reconfigurable hardware, using concrete examples I will describe the three that are available in this platform and how these have been fully leveraged in my design to improve performance; in section 6.3.
- I'll then propose the use of partial run-time reconfiguration of the reconfigurable hardware to change the algorithm implementation intra-day as a method of increasing performance. This gives the hardware implementation greater flexibility over the software implementation; in section 6.4.
- The hardware software interface is an important part of the complete system, I will present my thoughts on how this could be implemented using either *direct memory access* or alternatively a network interface provided by the FPGA system board, evaluating both approaches; in section 6.5.
- I'll round up the contributions in this chapter and discuss the greater picture of the work that has been presented here and the other areas of research that may benefit from it; in section 6.6

6.1 Limitations

Handel-C affords the user a lot of benefits, such as providing an easy and intuitive language to learn, especially if the user has a background in C. Handel-C supports a subset of the functionality provided by C, adding its own facilitating its use on reconfigurable hardware. Some missing features are core pieces of functionality that are used regularly in software programming languages, providing challenges when programming in the language. They also make it nearly impossible to cross-compile C code into Handel-C, as these omissions are likely to have been used. I would be implementing my solution from scratch, using the specification described earlier. As C isn't an object oriented language, Handel-C isn't either. This provides limitations of how much abstraction can be achieved and code duplication is increases as code needs to be repeated with a different type. Handel-C does support `structs`, which can contain

information to pass around the program. This is why new models for equity orders, market data updates, etc were created using `structs`, simplifying passing large quantities of information around the program and reducing code size.

Abstract data types are a difficult problem to solve in Handel-C. There are none provided by any libraries because creating one is either difficult or has little point in existence. You could create a list by using `structs` as entries and each entry having pointer to the next `struct`. However you cannot have a variable length as there is no notion of creating or destroying `structs`. They either exist in the hardware or they don't. This is why a better solution would have been to maintain an array of `structs`. Handel-C arrays need to have their size determined at compile time, preventing variable length arrays. This limits the flexibility of implementations and restricts solutions when dealing with sets of orders or level 2 market data update entries. This is because those abstract data types can have a varying number of entries, where now we must decide at compile time their sizes. In hardware, arrays are built up of sequentially arranged registers, they are heavily optimised for parallel access. As we want out design to be as parallel as possible, this makes arrays the obvious choice of data structure to use.

My solution to the problem of processing a varying amount of orders or entries is to process them in batches. Both order and entry batches have a size associated with them that must be determined at compile time. These sizes specified as a definition allowing them to be changed quickly. A current limitation of the design is that we can only process a single batch of market data entries to calculate the market favourability from. Extending this to process multiple batches is left as further work. We must now choose how many entries from the market data update that we will process, which may result in some information being lost. It might be the case that our market favourability measure will only be interested in the top ten entries, in which case choosing the batch size is straight forward. However, if we want the most accurate measure, which requires all possible information then there is going to be a difficult decision to make. The design does support multiple batches of orders but with each batch, all of the other information must be provided again. Each batch is treated independently and the market favourability must be recalculated. This only happens if there are more orders than the current order batch size. These limitations where due to making the implementation simple to allow greater analysis and lack of time. The last batch may not use all of the entries supported if there are less items than the batch size. In this case, the final batch will not operate at full performance, as some computation paths will have nothing to process. The entires that require processing are marked with a flag, if the flag is down then the implementation knows not to carry out any calculations on that data. This is a necessary compromise that has to be made because of the compile time limitations of array sizes.

One annoyance of using Handel-C, although it does have its advantage, is that you can only make a single function call per statement. For example, if you defined a function `int add(int a, int b)` which returns the sum of the two integers, you cannot use it again or any other defined method in the same statement. This makes the statement `d = add(a, add(b, c));` illegal and you would be required to write `d = add(b, c); d = add(a, d);`. The built in operators, such as $+$, $-$, $/$ and $\times$ don't have this limitation. The advantage of this limitation is that you can count the number of cycles a function or process requires. Only a number of items consume a cycle, such as assignment ("="). For example, the statement `a = (b + c) * ((d / e) + f);` only requires a single cycle, although it might result in a undesirably long pipeline. The statement `a = function(a, b, c, d, e, f)` would require one cycle plus the number of cycles required by the function. Using this information with the final design placed and routed on the device (the process of making the design ready for use on the device), we can derive the performance of the implementation. This is possible because when the implementation is placed and routed, the critical path delay of the circuit can be calculted, from which the maximum clock speed of the device can be derived.

The final challenge to overcome arises from function use in reconfigurable hardware. In a multi-threaded software design, as long as the processing is completely independent, all of the threads can pass through the same code at the same time without any problems. For example, if you abstracted a complex equation away from the main control thread to simplify making changes to the code, any number of threads could use that function at the same time. In hardware, a function is a discrete computation resource, of which there are a limited number. If the design only specifies one, then there is only one to use at any given time. In a single threaded design this isn't a problem as the program can only be in one place at a time. However, reconfigurable computing is designed to be massively parallel and we would be sacrificing nearly all performance advantage if we did this. Therefore, we need to specify how many function instances there should exist at compile time, depending on the level of parallelisation used. For

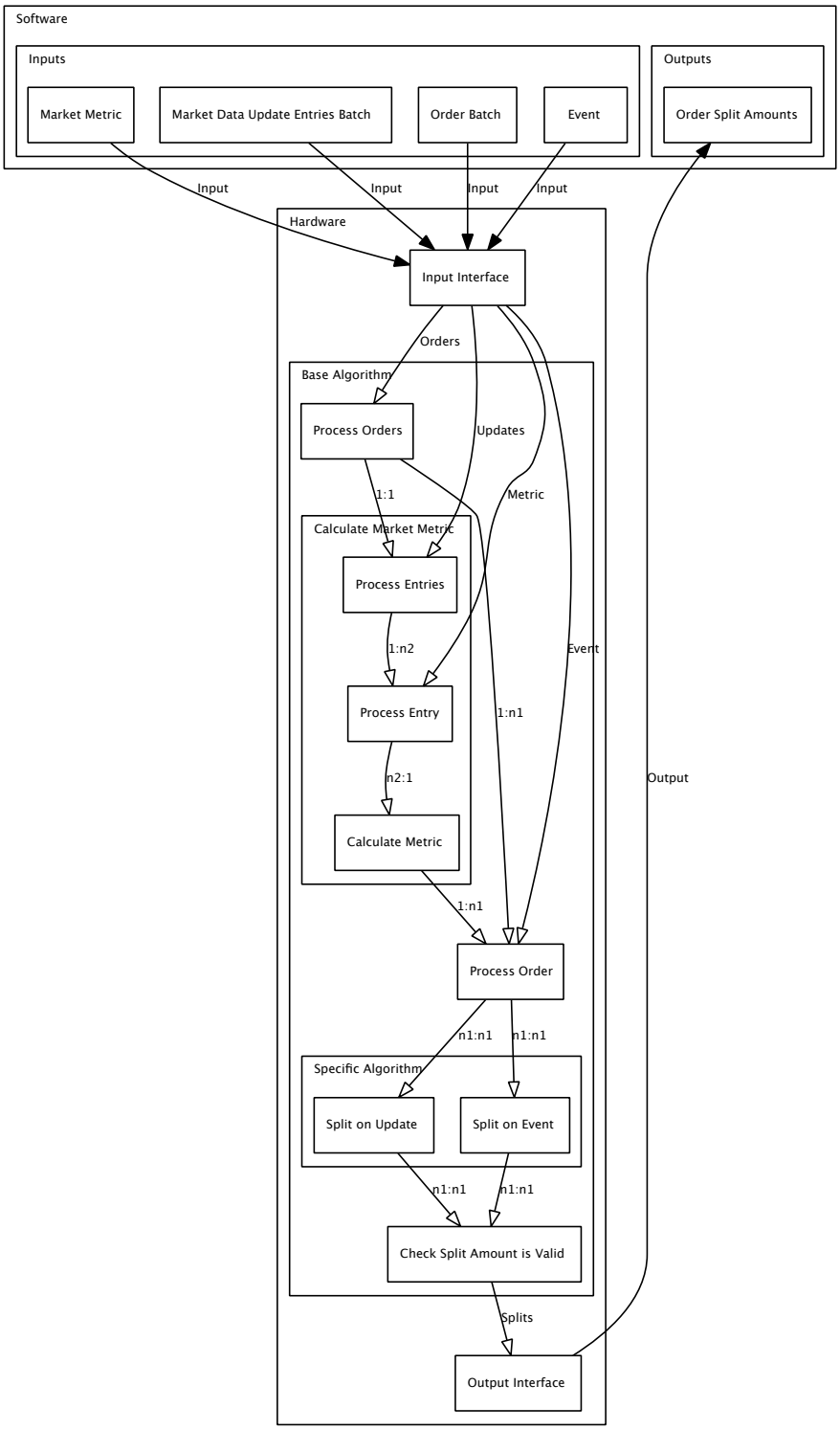


Figure 6.1: Hardware architecture from input to output highlighting important computational blocks.

example, if we have an array of four elements and in parallel we would like each element to be processed by the same function, then we require four instances of that function. Handel-C provides two solutions to this problem, either through `inline` functions or *arrays* of functions. An `inline` function works much like that of the software version, where the contents of the function are copied to where they are required, removing the method call. This duplicates the code of that function where it is used and it can then be used multiple times simultaneously. The second option is to have an *array* of functions, they aren't arranged in an array, but are accessed like an array element. If we carry on the example above, on processing an array of four elements, when we declare the function we must specify that there should be four instances of it. Now when we call that function, from a parallelised `for` loop, where all iterations of the loop are executed in parallel. We must also specify which of the functions it uses, normally the same as the position in the array of the data in question. I make use of both of these features depending the size of the function and how it is accessed.

Testing the hardware implementation was difficult. Without a device to place the hardware implementation on and run, the testing that was achievable was limited. This was because it was impossible to connect the hardware implementation to any of the software test frameworks. Testing within the Handel-C environment is limited to writing methods which will compare the output of a function with an expected result. However there is no built in framework to help aid this process, such as the unit testing framework discussed earlier. There is however a useful *simulator* built within the development environment. This allows code to be run at a reasonable speed and the contents of execution can be inspected. Additionally a limited *logger* is provided for the simulator environment. This is only available within the simulator and not on any FPGA chips. Using these I could perform limited testing of the implementation however further testing would be required when the implementation is placed on a device for use. This is also because all FPGA chips have slightly different interfaces and specifications, this means that it is non-trivial to take an implementation an place in on a device and use that same implementation on a different device. Part of this limitation was also the reason why the hardware was placed on a device, because of the difficulties in finding a suitable device to use. Although some devices were available, those that were had APIs that had never been used with the Handel-C environment. That would have resulted in a large amount time being devoted to placing the implementation on a device and getting it working. This would have reduced the amount of time that was available for investigating more interesting areas of the application of reconfigurable computing, such as partial run-time reconfiguration for swapping algorithms during the trading day, which will be discussed later in depth.

6.2 Architecture

I have created an architecture for the hardware implmentation of the trading algorithms, much like the design pattern for the software implementation. The idea is to form a base representation of an algorithm, which the specific algorithms can then utilise different parts of or supply different information to, depending on their own objectives. The control flow is set out by the base algorithm, providing the functionality for threshold trading. On top of this the specific algorithms implement their equations defining their behaviour described their specificaiton. Figure 6.1 shows this architecture design and over the next few subsections I will cover the important computational units. The inputs are shown coming from the software layer at the top of the diagram. Filled arrow heads signify a change in the abstraction layer. Information reaches the hardware layer, where it is processed in the algorithm implementations before returning to the software layer. This is where the orders are still managed, the child orders are created and sent to market.

The edge labels through the hardware section describe either the information coming from the `Input Interface` or parallelism used throughout the system. For example, $1 : 1$ signifies a single threaded operation, $1 : n$ shows a *fan-out* or creation of parallelism. Where $n : 1$ is the converse, the collapse of parallelism back to a single-threaded operation and finally $n : n$ maintains parallelism at the same degree as before. There are two different values of n for describing the parallelism amount, the values are dependent on the batch sizes. Where $n1$ is equal to the batch size for equity orders and $n2$ is equal to the batch size for the number of entires in a level 2 market data update.

6.2.1 Base and Specific Algorithms

The main purpose base algorithm is to implement the common functionality and set out the control flow. This starts from the `Process Orders` block which uses the `Calculate Market Metric` module to find the market favourability. This is then duplicated for every order as the order batch is fanned out into the `Process Order` block. Given the type of event information, it will determine the type of trading that will be carried out. The specific algorithms implement two types of trading which are `Split on Update` and `Split on Event`. This depends on whether it is a market data update or a trading event that the algorithm was supplied with. These two blocks are implemented by the specific algorithm. Finally the base algorithm performs a series of checks on the split amount to make sure that it is within the boundaries set out by the parameters on the order.

6.2.2 Calculating the Market Metric

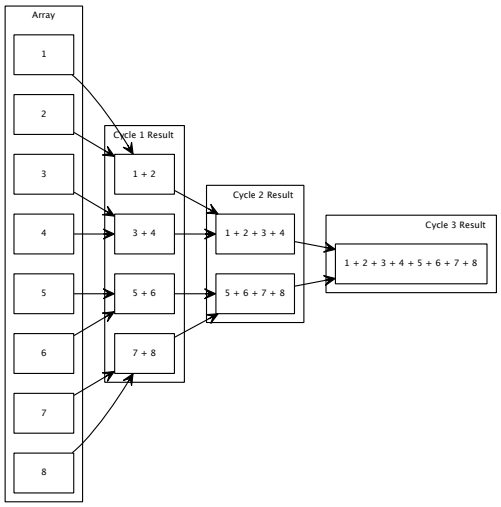
The market favourability is calculated using the same equations as the software implementation. The benefit afforded by the hardware implementation is that where the software has to iterate through all of the entries, performing a calculation on all of them before producing the answer, the hardware can do much of the calculation at the same time. Each entry can have its comparison value computed against the market metric individually. The difficulty comes from optimising the summation of the values. Although it could be hard coded into a single cycle, which would sum all values of the array, this would require the code to be modified before compilation if the size was changed. Singular summation of an array is shown in figure 6.2b. However, the advantages of parallelising the computation would be lost of we simply iterated through the array and summed all the values as this would require a cycle for every entry in the array, shown by figure 6.2c.

My solution is *tree* summation, where instead of summing the linearly which is expensive and takes $O(n-1)$ time complexity, the entries form leaves of a binary tree which are then summed back to find a single root node containing the value, the time complexity is $O(log(n))$. This is best shown in figure 6.2a. Singularly summing all of the entries requires $O(1)$ complexity, but as stated, breaks flexibility. The figures in 6.2 show the array being summed and the operations with their results occurring at each cycle of operation. Tree summation is implemented in Handel-C using the parallelised `for` loop. This has the same conventions of a regular `for` loop except the whole loop is carried out within the same cycle. This is wrapped in a conventional `for` loop which executes iteratively. The inner loop carries out the sums on the entries and places the result in the first entries position in the array. In the first iteration the entries are consecutive, in successive iterations the gap between the entries grows by one entry. If $log(n)$ where n is the number of entries in the array isn't a whole number then the resulting addition is with zero for non-existent entries. After completion, the result can be found in the first entry of the array. The disadvantage of this solution is that the information within the array is destroyed and cannot be used else where. If this is undesirable then the array could be copied to another for this process.

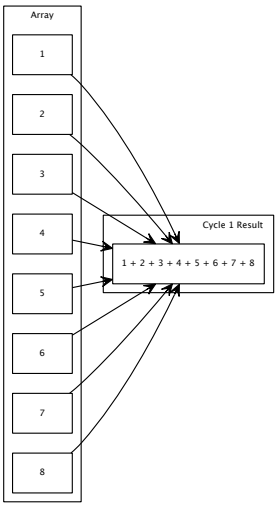
As we want to ignore negative values when we carry out the summation, when the pairs are summed, if the first value is positive and the second value is negative, the first value is summed with zero. In addition, we would need to know the total number of positive entries. This can be calculated at the same time as the summation, with the result being placed in the second entires location of the summation. In future iteration, these are just summed like the first entries. The final result ends up in the middle entry in the array. With these two final values we can find the average of the array. Figure 6.3 shows an example of this procedure, in cycle 3,the first value is the sum of the positive integers and in the 5^{th} as the number of positive values in the array. The average can be found by dividing the first number by the 5^{th} number, in this case $20/5 = 4$, which is clearly the average of the set $\{7, 5, 4, 3, 1\}$.

6.2.3 Split on Update and Split on Event

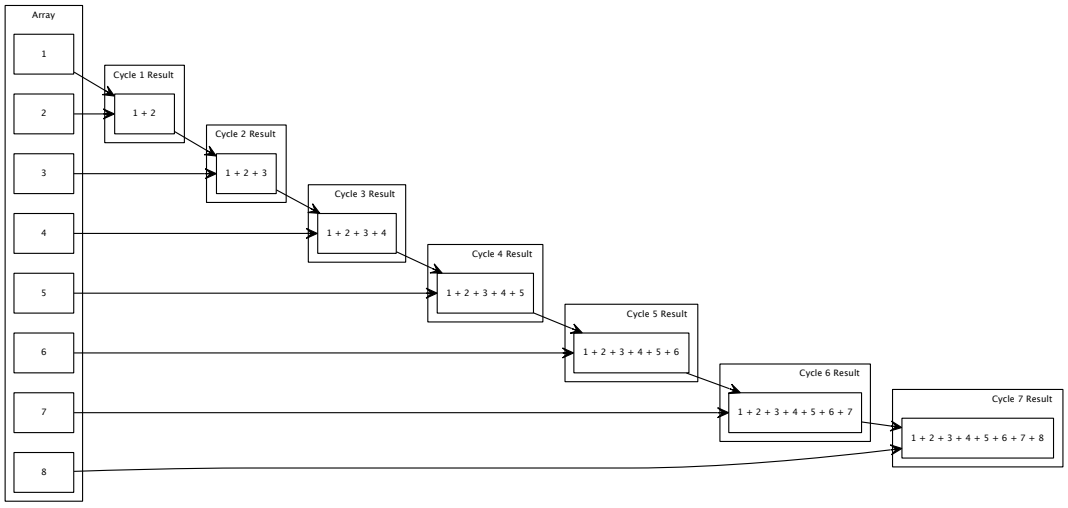
`Split on Update` is the hardware implementation of the threshold trading feature. When a level 2 market data update is received, every order is given the chance to trade ahead of its target trading volume, iff the market is favourable at that time. `Split on Event` is the hardware implementation of the core trading practice of the specific algorithms. For the participation algorithm it is on an execution data update from the market and both TWAP and VWAP algorithms are interested in timer events. Both `Split on Update` and `Split on Event` are implemented by the specific algorithms and the



(a) Tree summation of an array.



(b) Singular summation of an array.



(c) Linear summation of an array.

Figure 6.2: Three different methods of summing an array of values.

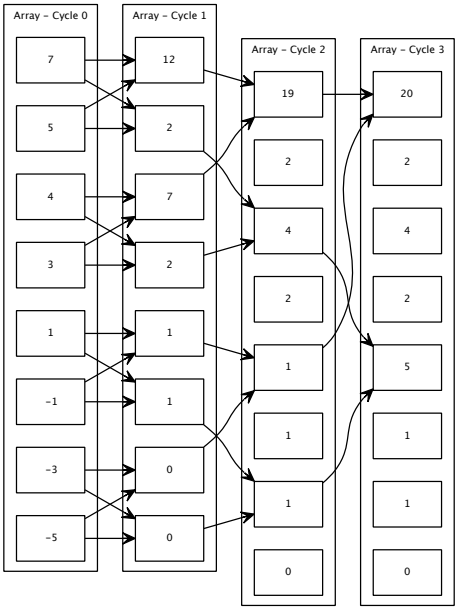


Figure 6.3: An example of the algorithm using tree summation and positive number detection to find the average of the positive numbers of an array.

base algorithm calls into to their functionality. The lack of classes limits how much code reusability is available and a hardware software interface for each algorithm is required. This is necessary because of the different amounts of information each algorithm requires. As each interface has to transfer different information, adding a small amount of extra code to specify which `structs` are being passed through does not increase the over head significantly.

The control flow and equations implemented in hardware replicate those that were described in software. As each order is completely independent of each other, they can be evaluated simultaneously. In addition, many of the equations used within these two modules are also independent of each others results, this allows further parallelism within the design. The two modules both output an array of split values, one for each order in the same arrangement as the input array. These values are then processed by the `Check Split Amount is Valid` module which prevents these split values from exiting trading boundaries. In addition to the array, a flag is also raised, notifying the module of which form of trading has occurred. This makes sure the correct parameters are checked against the split amounts.

You may ask yourself have the base algorithm can select between the two different modules. The obvious choice would have been to use an `if-else` statement, but this would have been ugly and have required more work to extend at a later date. Handel-C supports function pointers thankfully which allow for limited abstraction. This require the two modules to show the shame interface but this wasn't a problem as the same information could be supplied to both. This does have some overhead though as both modules may receive slightly more information than they require. This overhead though will only be in the implementation size of the algorithm and not in performance though. This allows the `Input Interface` to specify the module to use through selecting the correct function pointer for the `Process Order` method to invoke. Therefore in the future if the user of the architecture split on another input, they can program this functionality to have the same interface and then update the `Input Interface` to handle receiving this flag. Now the architecture will support a totally new trading functionality and no internal architecture code needs changing.

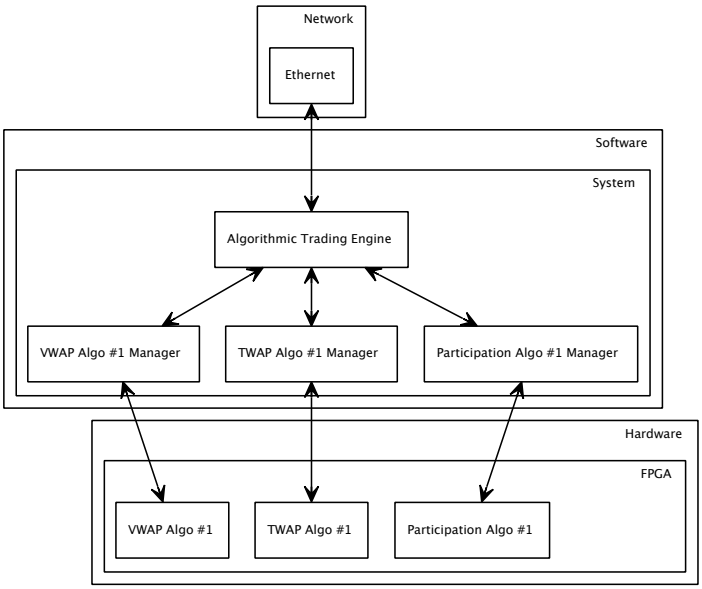


Figure 6.4: Task level parallelism within the hardware architecture.

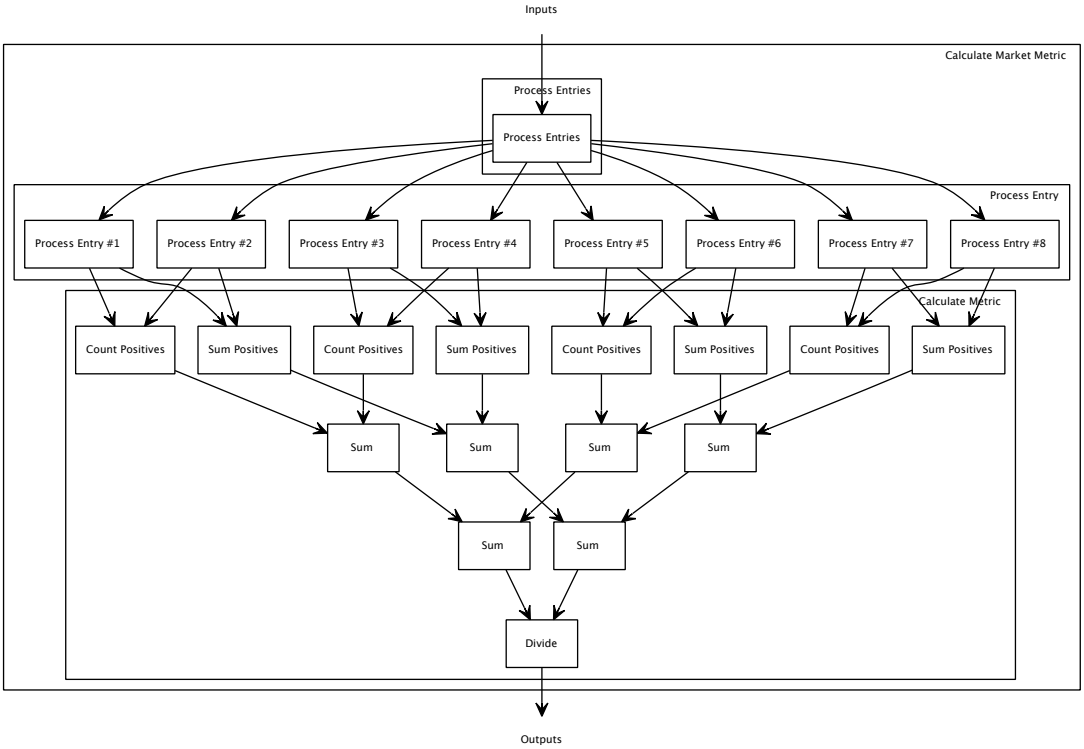


Figure 6.5: Pipeline level parallelism within the hardware architecture.

6.3 Parallel Processing

There are three different forms of parallelism supported within the hardware architecture. They are *task-level*, *pipeline parallelism* and *expression evaluation* parallelism. I will overview all three forms and give concrete examples of their use within my design. These forms of parallelism have very little overhead because of the characteristics of reconfigurable hardware. The development tools that are available for reconfigurable hardware are fully featured in helping utilities massively parallel designs. This simplifies and speeds up the development process. The highest level of parallelism is *task-level* parallelism, this can be represented as the placement of multiple algorithms on a single device. A FPGA is built up of configurable logic blocks which when connected together reproduce complex behaviour. There are varying amounts of different types of logic block available which provide specialised operations. An algorithm implementation requires a certain number of these logic blocks to be placed on the device, if sufficient blocks do not exist then the implementation cannot be placed. As care has been taken to reduce an unnecessary bulk within the code and the optimisations provided by the compiler have been used, I envision that multiple algorithms will be able to be placed on a single FPGA. This is not only important from a performance perspective, as each algorithm on the device can scale overall performance by a factor of the number of algorithms available. It is important to ensure the device has a number of algorithms to select from, as no one algorithm will be used all of the time by the clients. Figure 6.4 shows task-level based parallelism in the system as a whole. The reconfigurable hardware is represented at the bottom of the diagram with the three algorithms placed on the device. I have included some additional space which has been left free, this represents that it may be possible to fit another algorithm on the device. For example, if the VWAP algorithm was very popular, there could be a performance benefit from placing a second instance on the device which should double the amount of orders or updates that can be processed. In the diagram I have included the surrounding system, showing how each algorithm is tied to a manager which provides the interface between it an the software.

Pipeline parallelism is used to allow multiple tasks to be carried out at the same time. As most of the data within the process is independent from each other, it means that calculations can be carried out at the same time. Figure 6.5 best shows this principle, items within the same row of the diagram are carried out at the same time. The limitation to pipeline parallelism is that before the process can move onto the next set of calculations, all calculations in the proceeding set need to be completed. For example, if two modules were executing in parallel, if one module simply added two inputs together and assigned the result to a variable, this would take only a single cycle. However if the other module had to carry out intermediate work before completing its calculation, such as summing an array, then both modules would only finish when the slowest module completed. This means that there could be a lot of time may be wasted and care has to be taken to prevent huge discrepancies between module pipeline length to prevent wasted time and resources. One trade-off of producing a long parallel pipeline is the that it would increase clock speed and power consumption [52] of the implementation. An increased clock speed may be desirable and the increase in power consumption may be acceptable. However the parallel pipeline would also increase latency, this may be more difficult to accept given the nature of the algorithmic trading problem. We are not only interested in processing large amounts of information but also with as low as possible average and standard deviation on latency. This results in another trade-off that has to be made when designing the implementation. This trade-off difficult to make though and would require experimental analysis to discover the best results. However these experiments would be time-consuming and difficult to produce, therefore it would probably satisfactory to use programmer intuition on making the best possible decisions.

The final form of parallelism exhibited is *expression evaluation*. The three code blocks 6.1, 6.2 and 6.3 show the three methods of initialising two variables. Code blocks 6.1 and 6.2 are effectively the same, 6.2 is more explicit. Handel-C by default will evaluate all expressions sequentially. Code block 6.3 explicitly states that these two variables can be initialised at the same time. Code block 6.3 will take half the number of cycles of that in either 6.1 or 6.2. This is based on the fact that using the assignment operator ("=") requires one cycle when implemented on the device. Using this technique requires a different frame of mind when programming, you constantly have to remind yourself and think if what you have just coded can be parallelised. Additionally you may want to consider calculating some equations before they are required, in order that they can be parallelised. This presents the question, if the later equation is not required in all branches of the control, should it be moved ahead and always calculated if there is sufficient information. The decision should be made on whether this additional

```
1 int 32 result;
2 unsigned int 4 j;
3 result = 0;
4 j = 0;
```

Code Block 6.1: Sequential (non-parallel) initialisation of two variables.

```
1 int 32 result;
2 unsigned int 4 j;
3 seq {
4   result = 0;
5   j = 0;
6 }
```

Code Block 6.2: Explicit sequential (non-parallel) initialisation of two variables.

```
1 int 32 result;
2 unsigned int 4 j;
3 par {
4   result = 0;
5   j = 0;
6 }
```

Code Block 6.3: Parallel initialisation of two variables.

equation will require a longer pipeline than the other equation, if it does then it may slow down the calculation of the other equation. This might be offset by the fact that 90% of the time we require this additional calculation, making it worth while to move forward. All of these decision have to be made on implementation, making the process longer and requiring more planning. Finally, sequential and parallel blocks can be nested within each other, a parallel block is always nested in at least one sequential block which is the main program control flow.

6.4 Reconfigurability

Run-time reconfiguration would be an important part of any production quality system. This is because the resources of a FPGA are limited, meaning that supporting additional algorithms after an core set might be difficult. There would be tough decisions to make about which algorithms would be supported at a given time and changing this decision throughout the trading day would not be possible without fast partial run-time reconfiguration. FPGA chips are expensive and cost grows with the size of the chip (the amount of resources that it contains), therefore using additional or larger chips may not be cost effective. We are interested in making the chips as performance effective as possible, with partial run-time reconfiguration we have the option of tailoring our designs on the chip to market conditions. This would hopefully have a noticeable increase in performance.

Compilation is an expensive process (it can be considerably more time-consuming than compiling conventional software) and these implementations would require testing. My suggestion would be to create a pre-compiled library of acceptable implementation that could be quickly deployed to the device by the operator. These different implementations could be completely different algorithms or they may be just different configurations of existing algorithms, using different order and entry batch sizes. These different batch sizes may be able to cope with different market conditions better than others. For example, if a stock was particularly liquid it would have a lot of orders on the market, this would make the level 2 market data updates contain a lot of entries, therefore we may wish to increase the entry batch size so that more of these could be processed when calculating the market favourability. The size and granularity of the library can be determined by the user, based on a number of factors, including the scenarios that they expect to encounter and the capacity of their target device. Figure 6.6 shows a possible set of configurations for an algorithm to be pre-generated for use during a trading day. The variables m and n are only bound by the requirements of the use and the ability for the implementation

$$\begin{pmatrix} A_{10,10} & \dots & A_{10,n} \\ \vdots & \ddots & \vdots \\ A_{m,10} & \dots & A_{m,n} \end{pmatrix}$$

Figure 6.6: A matrix of different configurations for a single algorithm varying configuration by order and update batch size. Where m is the order batch size, n is the update batch size and A is the algorithm. Groups of matrices for differently algorithms could form a library of implementations to deploy.

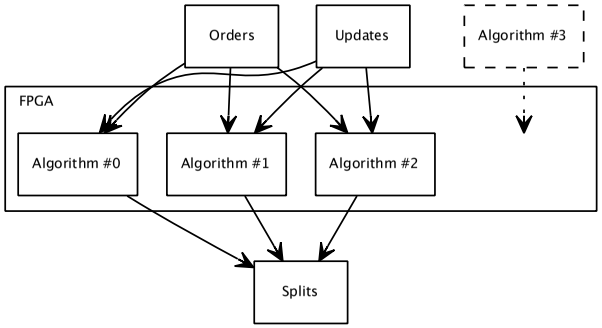
to be placed on the target device. The variables m and n represent the two different batch sizes.

It is to be expected that during times of increased market activity, fewer algorithms will be used but they will provide better utilisation. During times of less activity a wider range of algorithms could be selected to provide better trading functionality.This fine-grained approach to selecting which algorithms to use gives the user a lot of control, leading to performance benefits in both high and low activity times. Predicting before hand when to change algorithm implementation is more difficult however, it is not impossible from a limited point of view. Trading activity throughout the trading day can be coarsely predicted by expecting higher volumes of orders and activity at the start and end of the trading period. In addition there is a notable bump in load during the afternoon when the USA comes online. Future activity increases may have fore warning if a client informs the operator that they are about to place a significant number of orders. Load can be expected to rise after a particular piece of news is announced which may affected the markets, these are sometimes known to be happening before the announcement, leaving time to prepare.

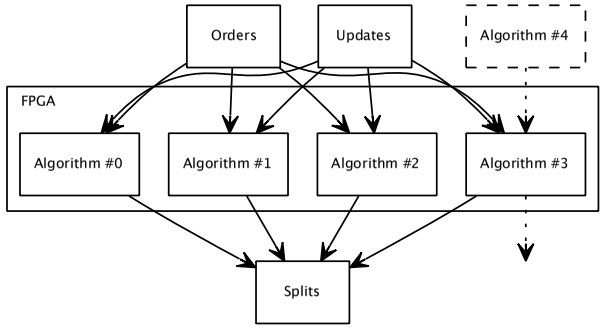
Figures 6.7 explains and shows the process of partial reconfiguration from a high-level perspective. The first scenario (on the left) has the enough capacity on the device to place an additional algorithm but the second scenario (on the right) doesn't have this luxury. After placement of the algorithms, the data feeds and output are connected up so that they can start processing data. The process doesn't intrude on the operation of any algorithms that aren't being modified and they are able to continue their work at full performance. On high market data update throughput, it would be appropriate to change the order batch size to that which was just larger than the current number of orders being processed. In the case where it cannot be increased enough, then increasing it to where wastage in processing the final batch is minimised by making the batch size close but above a factor of the total number of orders to process would be beneficial. With an increased depth of market, the update batch size can be increased to improve accuracy of decision making, utilising all of the information available. There may be occasions where an algorithm could be modified to make it more effective in the short term. If one algorithm is proving popular, then it would improve performance to place more instances on the device.

All of these situations lend themselves to using the run-time reconfigurable features provided by many FPGAs, allowing implementations to be modified or fully changed quickly. The disadvantage is that although relatively quickly it does still take a notable amount of time, especially when dealing with latencies in the millie and micro-second realms. During this time, performance is reduced and trading opportunities may be lost. For this reason, it would be useful to know how long example reconfigurations would take. During which, no processing can be completed. In algorithmic trading, updates can be dropped and left unprocessed or buffered for processing later. Generally only storing the last good update is required as it contains all the current information for the state of the market. If the algorithm tried to trade on updates that had expired then it may trade incorrectly causing a detrimental effect on its price performance. This is because the liquidity it was targeting may no longer be available when the trade is sent. However some algorithms may require this information for other purposes such as statistics or internal state. For the time being, the algorithms that we are interested in only require the last good update when resuming processing.

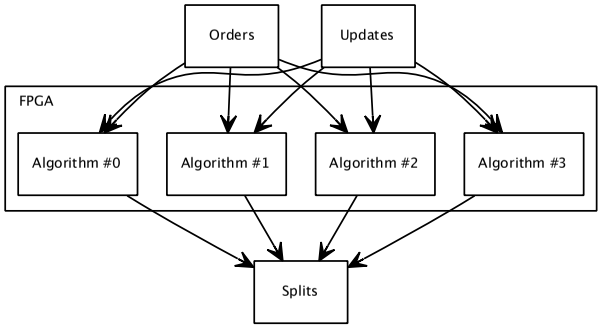
A software implementation that is designed to be able to load algorithms, has similarities to that of the hardware implementation. It would suffer from many of the same disadvantages of compiling and testing new algorithms as the hardware version. It lacks one major disadvantage of limited resources. Unlike our hardware version, there is no need to compile a library of specific variations of an implementation and store them separately. This is because an algorithm's memory footprint is small compared to the data



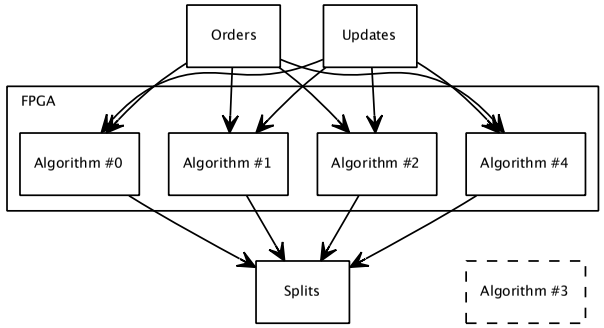
(a) Before partial reconfiguration, the FPGA already has space for the new algorithm to be placed on the target device.



(c) Before partial reconfiguration, the FPGA has no available space and an existing algorithm, #3 in this case must be removed.



(b) After partial reconfiguration, the FPGA includes Algorithm #3, connected and processing data.



(d) After partial reconfiguration, Algorithm #3 has been removed and replaced with Algorithm #4.

Figure 6.7: The two scenarios that may arise when performing partial run-time reconfiguration.

that it stores and so having a large number of algorithms ready to accept information in a software based environment is possible. Memory is finite but I assume that the system running this application has sufficient memory. As each algorithm uses a certain amount of time on the processing unit to execute, there is no concern over the implementation size of the algorithm and data representation size is coarsely grained. This relieves many of the problems faced by the hardware implementation. The software system can be thought of as having all the implementations ready and having to load them every time it wants to use them, much like how a reconfigurable hardware device would have to be reconfigured to use another algorithm. However for software, loading the code from memory is a natural overhead of the system and extremely quick compared to reconfiguring a hardware device. Therefore there is little benefit in designing a software system in this way, in addition the reconfigurable architecture affords many methods of increasing performance by device usage compared to software.

6.5 Bidirectional Hardware Software Interface

There are two possible implementations for the hardware software interface, both having advantages and disadvantages. The first option I will look at is using *direct memory access* (DMA), the second is communicating via *ethernet*. DMA is supported between the CPU and the FPGA by having shared memory owned by the FPGA mapped so that the CPU can read and write it additionally. Communications via ethernet is supported by the system board that the FPGA is mounted on, which are more commonly supplying an ethernet controller and interface which the FPGA can directly access.

The DMA option links the hardware and the software very closely. The first problem to overcome is the mutual exclusion of the critical shared part of memory which will be used to transfer the information on. For this I would suggest the use of the Peterson 2P algorithm [53], shown by code block 6.4. This provides weak fairness and as only a single lock will be required we are free from deadlock. The variable `turn` and the array `flag` would have to exist in the shared memory so that both layers could access it freely. The software layer requires to write down the information for the calculation and then would leave the critical region. Now the FPGA would take over, perform its calculation and write back into the critical region the result of this calculation, the FPGA then leaves. The software takes over again, after leaving it would immediate register its interest to enter the critical region again. This is because when the FPGA leaves its calculation will have been completed. The software can read the results out and remain in the critical region until there is more information to be processed, so it can write this information down immediately.The FPGA can keep requesting the critical region again immediately after exiting, knowing that when it gains access again it will have a calculation to perform. There would be a Peterson 2P algorithm for every trading algorithm placed on the device. This is because they are all completely independent, allowing easy partial run-time reconfiguration explained previously. Grouping of the same algorithms would be carried out by the software side of the interface, which would choose what information is sent to specific algorithms. This would allow work load to be evenly distributed between the multiple resources. The DMA solution is simpler than the ethernet solution, however it doesn't provide the freedom that the ethernet solution does. In the short term, the DMA solution is a better option, with the proposed further work and development on moving the whole algorithmic trading engine down to the hardware level, the ethernet solution would be the better option in the long term.

The ethernet system would require the trading algorithm to take information off the network interface, process this information before sending it back out again. The network interface on the hardware side would be considerably more complex than that of using DMA. This is because it would have to handle constructing the data from the individual packets before processing could begin. We would also be more reliant on other services, such as the network layer and the operating system, failure or delay in these would result in performance degradation of the trading algorithms. The software side of the

```
1 flag[i] = 1;
2 turn = i;
3 while(flag[(i % 2) + 1] && turn == i);
4 // enter/use/exit critical region
5 flag[i] = 0;
```

Code Block 6.4: Peterson 2P lock provides safe and fair mutual exclusion for two processes.

interface would also be more complex, it would have to implement further functionality to track what information had been sent and what had been received in response. Furthering the difficulties would be handling reconfiguration of the devices, knowing what algorithms were currently on the device and servicing information. This means that the ethernet hardware software interface is only really and option when implementing the rest of the algorithmic trading engine in hardware. This would remove these problems of discovering algorithms and transferring information. This is because the system would no longer be split and would be more unified than again. The communication of updates and orders is designed to be between remote system, where the communication between the trading engine and the trading algorithms isn't.

6.6 Summary

In this chapter I have described some of the limitations and problems associated with developing for a reconfigurable platform. I have presented my solutions to these problems and how they are applied in my designs. These solutions are generic so that they could be made use of in any number of other projects using reconfigurable hardware. Such as how to model data and effective use of function instances to avoid clashes. I have helped set out the state of mind that is required to effectively program for reconfigurable hardware, showing the extra thought that is required to make the implementation as effective as possible.

I have presented my hardware architecture which is used to abstract the common functionality of the algorithms being implemented in this project to reduce code duplication and speed up further algorithm development. This architecture follows many of the same principles of the software trading algorithm design pattern already presented in this report. I also showed in detail the optimisations that I used, particularly focusing on carrying out operations on arrays in a flexible yet time effective manner. My suggested solution is using tree summing of the array implemented using conventional and parallelised `for` loops. This results in operations completing in $log(n)$ time complexity and requires no code alterations when changing the array size. On top of this I have shown how multiple array operations can be carried out at the same time, on the same array, to find the average of the positive numbers in an array in the same time complexity. These are general solutions and could be applied to a large number of applications.

I have discussed the three forms of parallel processing, *task-level*, *pipeline parallelism* and *expression evaluation* parallelism and how they are used effectively within my implementation. I have shown through the use of task-level parallelism, a single device can support a large number of different algorithm implementation simultaneously, increasing both performance and flexibility. I have discussed some of the issues around pipeline and expression evaluation parallelism, that must be considered by the programmer when utilising these features. These thought should provide insight to those that are wishing to use reconfigurable hardware in their own projects and research.

Using task-level parallelism and partial run-time reconfiguration, I have suggested processes of being able to tailor a device, during the trading day to improve performance and flexibility. These processes are applicable to many other problems using reconfigurable hardware which would benefit from being able to modify themselves during operation to cater for different events, data or amounts of work load. Direct benefits from these features including providing better and more algorithms for clients to use. This in-turn makes the system more attractive for clients to use, with more clients and work load directly correlating to revenue for the operator of the system.

Finally I have discussed two methods of communication with the hardware layer from the software layer. Both *direct memory access* (DMA) and ethernet communication have benefits. In the short term, DMA is a much more attractive and simple solution, but in the long term the advantages of using network communication outweigh the difficulties of implementation. Both of these methods of communication are relevant to any system using reconfigurable hardware that requires communication between multiple systems, particularly between different abstraction layers. The network communication solution provides a road map to moving the whole of the algorithmic trading engine into reconfigurable hardware, bypassing the software implementation completely. This would result in a highly attractive solution, although controlling partial run-time reconfiguration from the hardware level would need to be investigated.

Chapter 7

Testing Harness

In this chapter I will outline the important work that I have carried out in creating a test harness which can be used to evaluate any number of electronic trading systems. Over the sections in this chapter I will cover the libraries that have been utilised how they be been integrated into the code base. This meant that the interesting structure of creating random but statistically correct event data which can be feed into the system easier and quicker to develop. Specifically this chapter will cover the following:

- Explanation and overview of the event logger used within the system. Description of the library integrated with the system and the extensions that I had to create it to complete its functionality required by this project; in section 7.1.
- Description of the unit testing framework built into the project, providing low-level functionality testing. Additionally how it is integrated into the main binary, allowing any user to complete a full self test. Finally its limitations and functionality lacking from a complete system; in section 7.2.
- The creation of a highly accurate and synchronous but flexible event timings. These can be used to call a method at a specified frequency until an end point. This is critical for testing the system at different data rates to be able to exam where performance degradation takes place; in section 7.3.
- Generating random event data to be processed by the application. Built up on a set of statistics which are adjustable depending on the testing needs that can create a number of different test profiles to stress the system; in section 7.4.
- A summary, describing the benefits of such a testing harness and its application in this project and other projects testing electronic trading systems. Also the study of trading events and market conditions; in section 7.5.

7.1 Event Logging

One of the first things required for a production ready system and testing is a good event logging system. This is so critical because the system has no GUI, so this provides the only form of feedback on the systems operation. As it is treated as a self-contained black-box, the only way of knowing what the internal state of the system is through the use of strategic logging output which has been incorporated. Although the use of a logging manager isn't particularly relevant to the aims of this project, it was a required part of work as no logging library provided the functionality required out of the box. Eventually I settled on using *log4cpp* for providing the logging functionality. This was because it seemed to be the most active and well documented logging project for C++. Unfortunately, it doesn't come with a very user-friendly front-end for incorporating into your code base. The first item I had to create was a `LoggingManager`, which would provide the interface to logging. This makes use of the *singleton* design pattern to ensure that only one instance exists at any one time. During setup it makes sure there is a valid log file available, which is named by the date they it was created. Methods exists for each of the logging levels and for each level there is a method which streams the output and another which accepts a string to log. The singleton implementation which I made use of was the one created by Scott Meyers, shown by code block

```
1 static ClassA& ClassA::getClassA() {
2   static ClassA instance;
3   return instance;
4 }
```

Code Block 7.1: Effective implementation of the singleton design pattern in C++.

```
1 [2010/02/22 00:09:22.144944] [INFO] [LoggingManager started up]
2 [2010/02/22 00:09:22.145022] [INFO] [Starting tests]
3 [2010/02/22 00:09:22.145537] [INFO] [TestParticipateAlgorithm::testOnLevel2Update(): Start]
4 [2010/02/22 00:09:29.331971] [INFO] [TestParticipateAlgorithm::testOnLevel2Update(): Finish]
5 [2010/02/22 00:09:29.332472] [INFO] [Tests completed]
```

Code Block 7.2: Example log output from the layout manager.

```
1  ostream& ClassB::stream(ostream& out) {
2    out << "<ClassB>";
3    ClassA::stream(out);
4    return out << "</ClassB>";
5  }
6
7  ostream& operator<<(ostream &out, shared_ptr<ClassA> command) {
8    return command->stream(out);
9  }
10
11 ostream& operator<<(ostream &out, shared_ptr<ClassB> command) {
12   return command->stream(out);
13 }
```

Code Block 7.3: Streaming all relevant information with class hierarchies in C++

7.1. This implementation of the singleton is very simple yet incredibly robust, ensuring the object is only created once during the program execution and destroyed at the end of execution. This implementation is far more concise than a similar object oriented languages, such as Java, which require static variables to be held in the class, rather than the static method.

The second item that I had to create to complete the logger implementation was the layout manager. This is a class that the library will use and provides with a large quantity of information, which it will return as a formatted string. Code block 7.2 shows some example logging from the layout manager, the format is the date followed by the time to microsecond precision. Next the logging level is output followed by the logging message. I wanted to add some additional information which would always be output in the logging as well. This was the class and the method which utilised the logging, this would improve clarity and debugging of the code, while reducing the amount of unnecessary hard coding. Currently the only method to do this is to hardcode the string for each method and class when it is output. This is less than desirable as it leads itself to being forgotten. Sadly I was unable to add this functionality, this was partly because of the abstractions that the library uses don't make it possible to backtrack to the caller. Secondly it is actually very difficult to find out the caller of a method at runtime. With further time, the best solution is probably to use a set of macros to solve this problem, but without previous experience and lack of time I choose not to pursue this further. After all this project is about exploring algorithmic trading in reconfigurable hardware, not implementing a fully-featured logger.

One small problem that had to be over come when supporting streaming of objects to the logger was how to deal with the class hierarchies. Thankfully the best solution was to carry out the streaming in a method owned by the object being output. The streaming operator would be a friend method to the class and take the object to output. This would call the output method on that object before returning the result. The big advantage came from declaring the output method as pure virtual methods (abstract) in the abstract base class. The result is that with on any type, the output method called was on the concrete class, not the abstract class. Another benefit is that in C++ we can define the functionality of pure virtual methods still, this means that if the abstract class has information that needs to be output, then the concrete class can call its base classes implementation to output its information two. This is

shown in code block 7.3 and no matter what streaming operator is called, the output will be the same if the smart pointer holds a `ClassB` object, where `ClassB` extends `ClassA`.

7.2 Unit Testing

For low-level unit testing I have utilised *cppunit* library to simplify the process of creating and running unit tests. Unit tests are associated with classes and are designed to test all of the functionality supported by that class. Code block 7.4 shows how to run automatically the tests after they have been registered. A `TestRunner` is created, all of the registered tests are added to the runner after being read from the `TestFactoryRegistery`, the tests are then run and the result output as the result of the programs completion. This can be integrated into the main program options, allowing any user to carry out a full self test of the application. This can provide trust in the system that the code being executed is correct and works. Code block 7.5 shows how the tests are set out, a test class is associated with the class it is testing. The test class sets out all of the tests that it will run and registers them through the use of macros, in lines 3 through 6. The class extends the `TestFixture` class supplied by the unit testing library. Additionally the `setUp()` and `tearDown()` are overridden which are executed before and after any test methods respectively. This allows a set of objects to be created for testing and then deleted after their use, so fresh objects can be created for the next test to be run. Finally, for the tests to be run, the test suite needs to be registered so that it is called when testing is run, code block 7.6 shows the macro which completes this which must exist in the source file of the test class.

All test methods utilise `assert` methods provided by the unit testing library. The simplest provided is `CPPUNIT_ASSERT(bool);` which passes the test if the bool value is `true`, otherwise the test fails. Others exist, such as comparators, reducing the amount of code required to create a test for an item. I wanted to include the use of a mocking library for C++, such as *gMock* or *mockpp*. This would have prevented the test classes from requiring use of any classes other than the one they were testing. This would have provided a better testing environment with less dependencies, unfortunately I didn't have sufficient time to perform this integration and get up to speed on using a mocking library. The lack of use of a mocking library shouldn't hinder the testing abilities outlined however.

7.3 Event Timer

Before I discuss the creation of the realistic test generations, I required a way of injecting communication into the system at a specified frequency. As the main processing thread is separate from the communication generation thread, we can sleep the generation thread between creating and sending events. Using three *boost* libraries, *date_time*, *thread* and *function* a very simple yet highly accurate and flexible solution can be created.

Code block 7.7 shows my solution for the event timer problem. It is implemented in its own class, but this isn't absolutely necessary and the method could be static as it requires no data from its class. The method takes a frequency to call a method as a double, an end time represented using boost's time class and a function call, using boost's method binding functionality, this is all shown in the method header on line 1. The time duration between the method calls is the first item calculated, this is in microsecond precision and the reason why 1,000,000 is divided by the frequency, on line 2. Next a timer is calculated which is the time until the next method call, this can be provided to the sleep function to sleep the thread until the time specified, on lines 3 through 5. Now the method enters a `while` loop, which terminates upon the current time becoming greater than the expiry time, this means that if the frequency is very small, the method can return without ever calling the supplied method, on line 7. Finally, within the loop, the method is called, the timer has the interval duration added to it and it becomes the time for the next method call and the thread sleeps until this time. At which point the loop will complete an iteration and enter the next, on lines 8 through 12. The method returns control of the thread after when the loop is exited.

The method accepts a bound method with either no arguments, or the arguments already bound into the bound function and the function must return nothing. Therefore it can accept a large number of already existing functions without needing to be changed. The advantage to this solution is that if the method call that it is performing is particularly expensive in time, as long as the final amount of

```
1  int main(int argc, char** argv) {
2    LoggingManager::infoStream() << "Starting tests";
3
4    TestRunner runner;
5    runner.addTest(TestFactoryRegistry::getRegistry().makeTest());
6    int result = runner.run("", false);
7
8    LoggingManager::infoStream() << "Tests completed";
9
10   return result;
11 }
```

Code Block 7.4: Initiating and running all of the unit tests to complete a self-test after automatic registration.

```
1  class TestClassA : public CppUnit::TestFixture {
2
3  CPPUNIT_TEST_SUITE( TestClassA );
4      CPPUNIT_TEST( testMethodA );
5      CPPUNIT_TEST( testMethodB );
6    CPPUNIT_TEST_SUITE_END();
7
8  public:
9    TestClassA();
10   virtual ~TestClassA();
11
12   void setUp();
13
14   void tearDown();
15
16   void testMethodA();
17
18   void testMethodB();
19 };
```

Code Block 7.5: Class layout for an example unit test class `TestClassA`.

```
1  CPPUNIT_TEST_SUITE_REGISTRATION( TestClassA );
```

Code Block 7.6: Test registration for test class `TestClassA` in `TestClassA.cpp`.

```
1  void EventTimer::sleepInterval(double frequency, boost::posix_time::ptime end, boost::function
       <void()> method) {
2    boost::posix_time::time_duration interval(boost::posix_time::microseconds(1000000 /
         frequency));
3    boost::posix_time::ptime timer = boost::posix_time::microsec_clock::local_time() + interval;
4
5    boost::this_thread::sleep(timer - boost::posix_time::microsec_clock::local_time());
6
7    while (boost::posix_time::microsec_clock::local_time() < end) {
8      method();
9
10     timer = timer + interval;
11     boost::this_thread::sleep(timer - boost::posix_time::microsec_clock::local_time());
12   }
13 }
```

Code Block 7.7: A simple but highly accurate and flexible event timer.

time is less than the frequency interval then the method will remain in time. This is because the time to sleep till next is iteration is calculated after the method has executed. In the event that the method takes longer than the frequency interval then the method will try to sleep, but the time will have already elapsed so it won't sleep at all and the next iteration will take place. At which point the frequency of calls is equivalent to how many times the method can be executed in a second. This problem could be eliminated with careful use of additional threads to execute the method calls independently of the sleeping thread, but this solution is sufficient for this projects needs. The accuracy of this solution was measured to be around the 50 microsecond which is more than sufficient for the purposes of this project.

7.4 Realistic Test Generators

The realistic test generators produce the different parts of communication package. They work in a hierarchical waterfall fashion, `CommunicationGenerator` is the top controller which will be called and return the next communication event, this can be seen in figure 7.1. The `CommunicationGenerator` uses four of the generators, with the exception of the `ResponseGenerator`, one is chosen from the remaining three depending on its internal state. The `counter` holds the state and is incremented after every `generate()` call. Every `frequency` calls, the `CommunicationGenerator` returns a `TimerEvent` generated by the `TimerEventGenerator`. The `TimerEventGenerator` creates a new `TimerEvent` and returns it. It is the simplest of all generators as `TimerEvents` contain no other information. On all other calls, the generator makes the decision to return a `MarketData` or issuing a `Command`. This decision is made by generating a pseudo-random number which is in the range $[0, 100)$. If the number generated is below the ratio, then a `Command` is generated and returned, otherwise a `MarketData` update is created and returned.

The only exception to this is the `ResponseGenerator` which requires a `Command` to generate its response output. This generator is designed as a reflector of commands issued by the system and provides feedback on whether the order was accepted or rejected in our virtual environment. Currently it doesn't support generating any update responses such as `Execution` or `ForceCancel`. To add this functionality would have required significantly more work, it would be required to keep track of all orders that it had accepted. This would have meant it would have had similar functionality to the order manager, you could use an order manager slightly modified but then you would essentially be testing something with itself which isn't a good practice. The `ResponseGenerator` uses ratios to decide if a command is accepted or rejected, held by the variable `acceptanceRate`.

The use of pseudo random number generators (PRNG) to decided which event is created and sent is important. This is because a PRNG seeded with the same number will produce the same series of numbers every time it is executed. Additionally, the numbers produce in the range will all have equal probability of being generated. These two features allow tests to be exactly the same every time they are run. We also want the ratio of choices to be maintained, so if we want 90% of events to be market data updates, then out of 100 events generated we should see very nearly 90 updates.

Next, the `CommandGenerator` generates a new command. Again this uses a PRNG to decide what type of command to create. This time we also have boundaries to keep the number of active orders within. These are `minimumNumberOfPreviousOrders` and `maximumNumberOfPreviousOrder`. When started, there will be zero active orders which may be less than the minimum, so until the minimum is reached new orders will be sent. If the maximum is reached then cancel orders will be sent. If the total is between the boundaries then a ratio decides whether to increase the number of active orders or cancel an existing one. The amend command isn't utilised at the moment as its functionality isn't properly supported throughout the system. When an order is created it is added to a map to store it, this is so that when an order is to be cancelled, one can be selected at random from this map and then cancelled. The `CommandGenerator` doesn't create orders however, that is left to the `OrderGenerator`. The disadvantage to the current solution at the moment is that all orders are grouped together. Further extension could include boundaries for all different stocks being tested. This would make it easier to create a large number of orders on a single or multiple stocks and small numbers on others.

The `OrderGenerator` produces parent orders for the `CommandGenerators`, it decides the order attributes that it is going to use in creating the new order. To do this though it requires some information about the stock that the order is to be traded on. For this, there is a class called `SymbolData` which stores this information. This includes information such as the price of that stock. The act of choosing

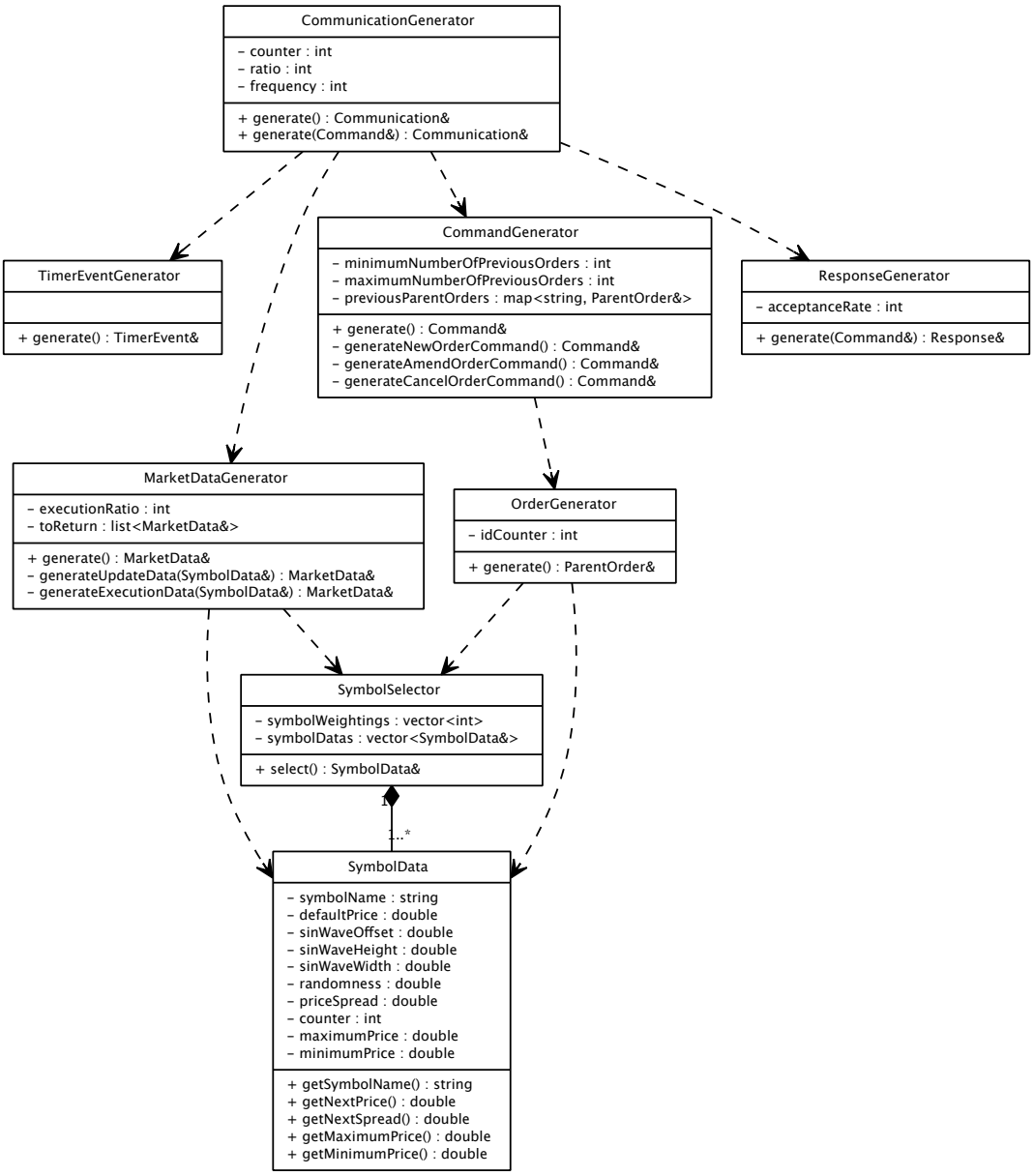


Figure 7.1: Class structure of the generators package showing interactions and waterfall hierarchy.

what stock to trade is carried out by the `SymbolSelector`. This maintains all of the `SymbolData`s, each with their own weight. This weight determines how often that symbol is selected for trading. When asked, it will provide the next `SymbolData`, which can be used to create an order. In addition there are a number of other order attributes that need to be decided, such as the order type, order time in force and order trading details. The most important one is the order trading details, as this will decide which algorithm trades the generated order. Again, these are all decided by ratios on the individual attributes, this allows them to be changed depending on the test.

The final generator which I haven't covered is the `MarketDataGenerator`. This generator differs slightly again from the core style. It contains a ratio (`executionRatio`) which determines the number of execution updates compared to other updates. What is required in addition, is because a level 1 market data update is a subset of the information in a level 2 market data update, we must form a level 2 update, then create a level 1 update from this update. These must be sent separately, so the unsent updates must be stored within the generator and upon being called again to generate an update, it returns the first update in this list of pre-calculated updates. Like the `OrderGenerator`, the `MarketDataGenerator` requires `SymbolData` and the symbol chosen for this update is done by the `SymbolSelector`. The biggest downside from this current implementation is there is a direct correlation between the number of orders that a symbol receives and how active that symbol is for updates. This is unfortunate because it would be interesting to vary this relationship, however it is more realistic that if a symbol was receiving a large number of orders then there would be a large number of market data updates. The `SymbolData` class calculates the next price point and the spread between the buy and sell prices. This is used by the `MarketDataGenerator` to create level 2 market data updates. Firstly, the number of entries on each side is chosen by random number, again this can be bounded by parameters. Next each of those updates is move slightly further away from the previous price point on that side. The quantity of the entry is also randomly generated. The price spread determines the start point of selling side, which is the sum of the spread and the price generated. The `SymbolData` must generate the next price and spread point for a symbol. This is dependent on its internal parameters and state. The greatest problem is in creating a function which varies the price of a symbol over time in a fashion which is realistic enough and also simple enough to provide an accurate representation. For this I chose to use the *sine* function, manipulated with the parameters to distort its curve into a price of a symbol over time. I chose this over a more complex function of trying to vary the price over time depending on randomly moving the price, however this could have fallen into the trap of ending up being very noisy. With the sine wave solution we have a clean price change over time which can be made noisier by including an amount of randomness into the price.

$$Price = P_{Default-Price} + (R_{Amount} \times R_{Factor}) + (P_{Height} \times sin(P_{Offset} + (P_{Width} \times S_{Time}))) \quad (7.1)$$

Equation 7.1 describes how the price for a symbol is calculated. $P_{Default-Price}$ is the median and average of the sine wave. R_{Amount} is the amount of randomness, setting this to zero will mean that the price is a perfect sine wave, R_{Factor} is the generated random number to use. The randomness is used to slightly fluctuate the price away from the sine wave. P_{Height} is the height of the sine wave, from which the maximum and minimum prices can be calculated. P_{Offset} is used to move the start of the sine wave away from time zero, this means some symbols prices will be increasing while others are decreasing at the same time. P_{Width} is used to stretch or shrink the sine wave, changing the amount of time to complete a full cycle. Finally the S_{Time} is the state attribute describing the time of the price. The random number generated has a range of $[-1, 1)$, meaning the randomness can move the price higher or lower.

$$Spread = P_{Default-Spread} + abs(R_{Amount} \times R_{Factor}) \quad (7.2)$$

Equation 7.2 shows the calculation for the price spread between the best buy and best sell price. The only additional variable is $P_{Default-Spread}$ which is the default spread to which an amount of randomness is added. Also note that this time the randomness value is always positive by the absolute being taken. This is because if the spread was negative then the market would be crossed, the best selling price would be cheaper than the best buying price. This would mean executions should have taken place that haven't and would likely indicate that there was a problem somewhere in the system. You'll notice that the spread doesn't depend on any internal state, simply the randomness and the parameter provided,

Stock	$P_{Default-Price}$	$P_{Default-Spread}$	R_{Amount}	P_{Height}	P_{Offset}	P_{Width}
1	250	1	5	20	0	1
2	40	3	3	5	30	3
3	60	2	4	10	40	2
4	120	1	2	30	100	4
5	200	2	1	15	270	2

Table 7.1: The parameters used to generate graphs 7.2 and 7.3 with equations 7.1 and 7.2.

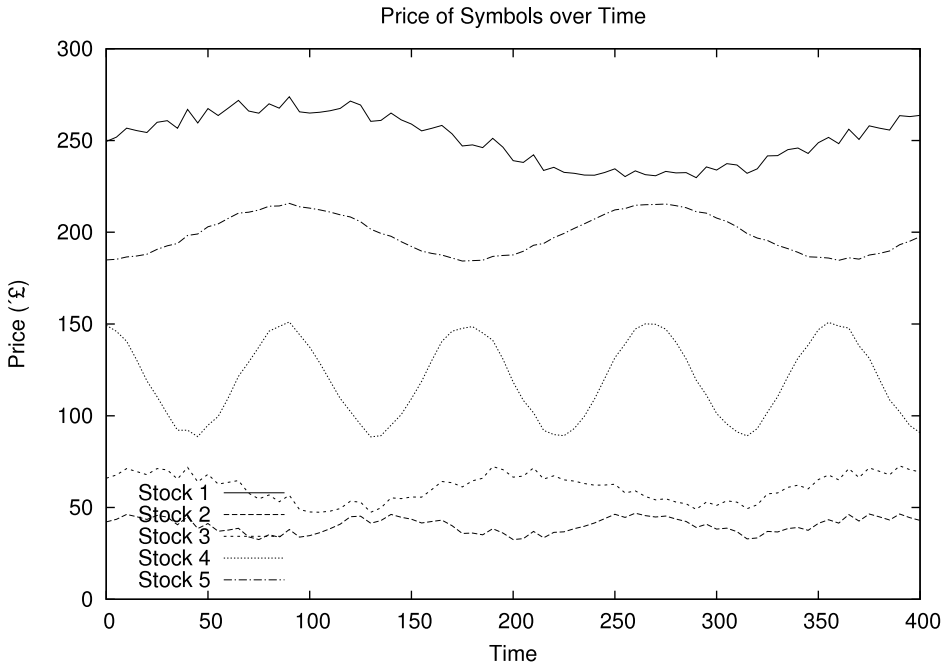


Figure 7.2: An example of 5 stocks prices varying over time, each with different parameters.

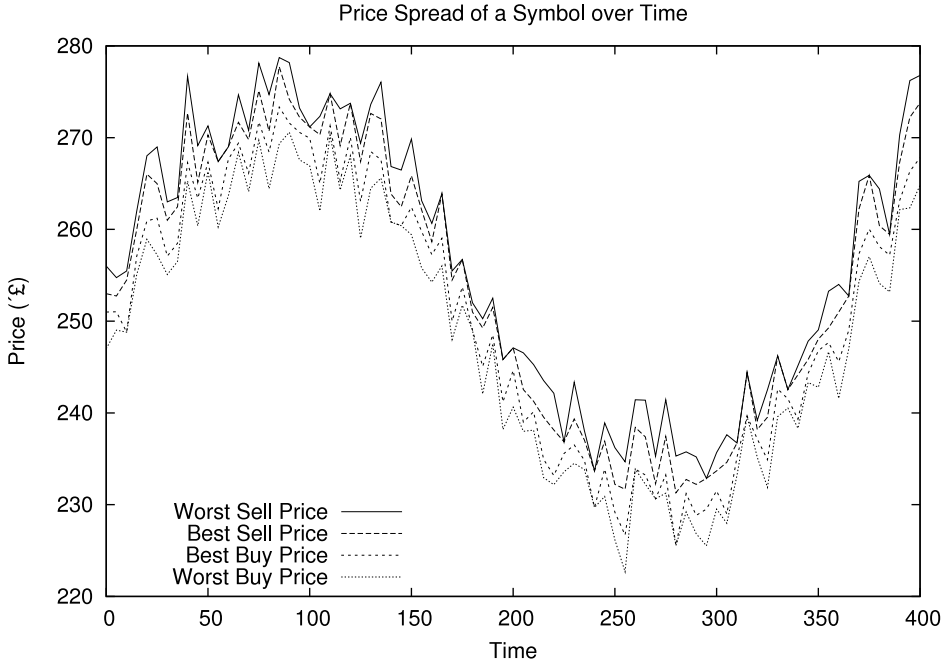


Figure 7.3: An example of the price spread of a single stock varying over time.

meaning the spread can be generated multiple times without furthering the state of the market data. From the spread, the full market depth can be built up. This is done first by generating random numbers which will determine the depth of each side of the market, these are limited by default to a range of $[0, 10)$. Next for each entry, the price is moved away from its best price (increased if selling, decreased if buying). The amount it is moved away is a random number in the range $[0, 1)$. Additionally the quantity of the entry is generated randomly, with a default range of $[100, 1000)$.

To demonstrate the power of these two functions, here is an example of their use. Table 7.1 shows the parameters used for five stocks. Then using equation 7.1 we can calculate the price over time for these symbols, this is shown in figure 7.2. It is clear to see that *Stock 4* has a very small interval, meaning that it cycles very quickly, as opposed to *Stock 1* which has a very large interval and a cycle takes much longer. You can see that both *Stock 4* and *Stock 5* have very low randomness, represented by their near perfect sine waves. This is opposed to *Stock 1* which has a high randomness, couple with its low price height and long interval, it makes it more difficult to see that it is a sine wave. Using equation 7.1 and table 7.1 we can also plot the spread of symbol with is generated market depth, this is shown in figure 7.3. In this case I have chosen only to plot *Stock 1* on the graph to simplify it and allow for contraction of the y-axis range. It is clear to see the four lines stack on top of each other, the top line is the *Worst Sell Price*, the most expensive, followed by the *Best Sell Price* which is the cheapest selling price. The converse is true for the buy side with the *Best Buy Price* being first as it is the most expensive price and the *Worst Buy Price* being the cheapest buying price. If you look carefully, at no point do any of the four lines cross, the two lines for each side may overlap a little where they are the same. This is when the best price is equal to the worst price, a single depth market. The two best prices however will never cross or overlap as this would be a crossed market.

The big advantage to using this design of generating random event data for testing is that apart from the `ResponseGenerator` which requires input to respond to, the event stream can be precomputed. This means that before the test is run, the stream of events can be generated and stored, ready for replaying when desired. The downside to this is that the storage requirement required may be considerable, especially when requiring a large number of event updates to replay quickly. To counteract this though, the events could be cycled, this shouldn't perform any differently, but the system output may also loop as well, the only thing that would need to be updated would be new order identifiers to make sure that they don't cycle as well.

7.5 Summary

In this chapter I have presented my approach to integrated event logging in the system, utilising the *log4cpp* logging library to provide the core functionality. I have shown how a `LoggingManager` provides the interface to the logging functionality and the manager sets up the log files to ensure absolute clarity. Further more I have produced a logging format which is clear to understand and uses microsecond precision for event timing. This greatly enhances debugging and testing of the system. These principles could be applied to any number of other C++ based projects. I have also discussed the failures of attempts to include the class and method name producing the logging but have shone light on how this could still be achieved.

To provide low-level testing and validation of the system, I have integrated a unit testing framework within the application through the *cppunit* library. This provides a simple process to creating and executing tests, tests are automatically registered to a test running, reducing the amount of boiler-plate code required which could be forgotten. I have proposed how the tests are integrated with the main binary, allowing any operator to run the system in a self test mode, executing the tests and receiving confirmation that the system has passed. This provides confidence and trust in the system by assuring the user that the binary they are about to deploy is correct. I feel that this is a worth practice of all applications and would recommend that it is used in other projects. There is some overhead in the size of the binary, but it is not considerable and with the cheapness of storage space it can be mitigated.

Enabling the ability to test the system at different data rates is an important part of the test architecture. To do so I have created a highly accurate and flexible event timer, utilising the power of the *boost* libraries. The timer is accurate to roughly 50 microseconds and will remain this accurate even when provided with a multitude of different methods to call when the timer fires. This is due to how it monitors how longer the method takes to complete before decided how long it must sleep until it must

carry out the process again. The event time is abstract by using bound functions it can be provided with any number of functions without requiring any code changes. This means that this functionality could be used in a number of different scenarios, projects and research.

Finally, I have presented my design for creating statistically random test events in predetermined ratios. This allows for realistic test data to be generated before testing or just in time, to be injected into the system. The generators of this data are arranged in a hierarchical waterfall structure but can be removed from the complete package and used independently. Such as if only market data updates were require, then the market data generator could be lifted with its dependencies and used in another application or piece of research. Alternatively, the operator of the generators can arrange the ratios of generation so that the complete generated will only generate market data updates, removing the need for the package to be cut up. This provides the user with a large amount of power and use for such a test generator has wide reaching effects. It would be extremely useful when building and testing any number of electronic trading systems and it isn't dependent on how they are implemented, whether it is in software or it is in hardware. I have also presented in detail the equations that I have invented to produce a moving price and depth of market for any number of stocks, provided with a set of parameters. This produces a realistic fluctuation in price data which should enable all of the behaviour of an electronic trading system to be observed.

Chapter 8

Evaluation and Results

With any new research, a strong emphasis should be placed on evaluating the work that has been carried out. I have been looking at is a applying existing ideas to new technologies in this project, therefore we are interested in how effective these new solutions are compared to the original. I will not only look at the computational performance of the system, which is obviously very important, but also a number of other areas of performance. Specifically this chapter will cover the following:

- Introduction to the details of the implementations and the testing methods. I will also discuss how I carried out the testing to avoid any skewness in results where required; in section 8.1.
- Performance analysis will hopefully confirm my hypothesis that reconfigurable hardware provides sufficient acceleration over software to warrant its use within the algorithmic trading setting. Including a technology independent analysis with wide reaching applications; in section 8.2.
- Partial run-time reconfiguration provides flexibility at the expensive of time, we need to know how much time to decide if it is an acceptable trade-off. Again I will use technology independent analysis methods to further understanding; in section 8.3.
- Power consumption is an important consideration for companies with a large computing footprint, given the current environmental climate. Therefore it is useful to know of any power efficiency gains over the previous solution; in section 8.4
- Additionally cost and space efficiency are important when considering a new technological solutions. Therefore I present analysis of the current options and comparison taking into account performance achieved; in section 8.5.
- I will look at the effectiveness of the software produced in this project to add some qualitative analysis to the successes of this project. This will cover how the framework meets its objectives; in section 8.6.
- Summary of the information and work presented in this chapter, covering the other applications of the technology independent analysis techniques presented. This will allow for conclusions on these results to be draw in the next chapter; in section 8.7.

8.1 Implementation and Testing Details

The hardware implementation was developed in Handel-C [54]. The development environment used was the Mentor Graphics DK Design Suite 5.2 which compiles the Handel-C implementation and exports it as an Electronic Design Interchange Format (EDIF). Then Xilinx ISE Webpack 11.1 was used to place and route the design on the target chip, in this case the Xilinx Vertex 5 xc5vlx30. This chip was selected because it is one of the smaller chips available from Xilinx, therefore making it one of the most affordable and common chips in industry. Additionally, it was the largest chip supported by the Xilinx ISE Webpack 11.1.

The software implementation has the same specification as the hardware architecture, allowing for a better comparison between the two implementations. The software was implemented in C++ and the reason for this were previously discussed in chapter 2. The software implementation doesn't carry out batch processing as this was unneeded due to variable sized abstract data types available. In addition the implementation wasn't heavily multi-thread. Further work includes introducing multiple worker threads to process collections of orders and entries in parallel. For the time being will can assume that multiple instances of the application are running, one per processor available, to discover the full performance of the software implementation. The development environment used was Eclipse with the CDT plug-in to aid C++ development on Mac OS X 10.6.2 using g++ 4.2.1 for compilation with full optimisation enabled. The host machine was a MacBook Pro with 2.33GHz Core 2 Duo, 3GB of 667MHz CL5 DDR2 and 500GB 7,200rpm hard drive with 32MB of cache formatted in Journaled HFS+. As Mac OS X isn't a real-time operating system, all software implementation tests where executed multiple times. This reduced any skew in the results from the operating system prioritising other processes on some test.

8.2 Performance Testing

Arguably the most important testing is on the performance of the two systems. With this project though this is easier said then done. I have been dealing with very complex procedures and even with clever event generators discussed, producing meaningful tests and results is difficult. This was not aided by the limited time available and the inability to be able to put the hardware implementation onto an FPGA chip because of the lack of a suitable device. Never the less, because of the features of reconfigurable hardware, they lend themselves heavily to theoretical analysis easier than software does. This is how much of the analysis is carried out initially in research and production. Therefore I will first present my technology independent performance analysis which will model the performance of the hardware system mathematically. Then to produce results I can use the data from the hardware implementation and with the software results draw preliminary comparison and evaluation of the two systems. For the initial testing results, because I wanted to be able to compare the two implementations directly, the order manager was bypassed. This was because if it was used in the software testing then we would have its overhead, where with the analytical analysis of the hardware implementation this couldn't be taken into consideration. This was the downside of not having any available devices to place the hardware implementations on before the project was due. For the test data, the test generators described earlier where utilised to generate a set of orders that were repeatedly used. Then a set of updates where generated with the ten entries on each side, to max out the hardware entry batch size. The calculation where timed over these two sets of data.

8.2.1 Performance Technology Independent Analysis

Using the work in chapter 6 we can perform technological analysis independent of any specific technological implementation. In this section I will outline analysis of such a model in preparation for any implementation. If we assume that we have L_{total} logic cells available and each algorithm requires L_{used} logic cells then we can calculate the total number of algorithms that can fit on a single device, $N_{engines}$ defined as equation 8.1.

$$N_{engines} = \lfloor L_{total}/L_{used} \rfloor \tag{8.1}$$

Defining B_{orders} to be the batch size used for orders and $B_{entries}$ for the batch size of the number of market date entries per update supported. The market favourability calculation is defined by equation 8.2. Where C_{metric} is the number of cycles taken to perform the calculation, $C_{entries-const}$ is the number cycles required calculate the metric free of the number of entries. Likewise we can calculate the number of cycles to process a batch of orders, C_{orders} and $C_{orders-const}$ being the constant number of cycles required for each order, by equation 8.3.

$$C_{metric} = 2 \times (C_{entries-const} + B_{entries}) \tag{8.2}$$

$$C_{orders} = C_{orders-const} \times B_{orders} \tag{8.3}$$

The total number of cycles being equal to the sum of the number of cycles to calculate the metric and process the orders. If F_{max} is the maximum operating frequency of the system, then we can calculate the maximum update throughput, $T_{max-updates}$, in updates per second by equation 8.4. The expression will process up to B_{orders} per update because of the batch processing. Conversely the maximum number throughput, $T_{max-orders}$ by equation 8.5.

$$T_{max-updates} = \frac{N_{engines} \times F_{max}}{C_{metric} + C_{orders}} \tag{8.4}$$

$$T_{max-orders} = \frac{N_{engines} \times B_{orders} \times (F_{max} - C_{metric})}{C_{orders}} \tag{8.5}$$

Although these are maximum throughput for processing updates or orders, in most situations the system will have to deal with multiple order batches and multiple updates per second. When the system has more than B_{orders} but does not have so many that it must spend a whole second processing them all, allowing it to handle multiple updates per second. This is expressed in equation 8.6, where $N_{order-batches}$ is defined in equation 8.7 and N_{orders} is the number of orders to be processed. $N_{order-batches}$ is the number of batches are required to process N_{orders}.

$$T_{updates} = \frac{N_{engines} \times F_{max}}{C_{metric} + (N_{order-batches} \times C_{orders})} \tag{8.6}$$

$$N_{order-batches} = \lceil N_{orders}/B_{orders} \rceil \tag{8.7}$$

We also care about the latency of processing, not only the overall maximum throughput. For example, if we have a very large throughput, but processing takes a long time then the implementation would be useless, by the time the decision was made, the result would be out of date. Again we can form an mathematical model for this property, shown by equations 8.8, 8.9 and 8.10. Where $L_{avg-decision}$ is calculated from the average from the minimum latency $L_{min-decision}$ and the maximum latency $L_{max-decision}$. $L_{avg-decision}$ is the average latency of the decision being made and the rest of the variables are as the same defined previously.

$$L_{min-decision} = \frac{C_{metric} + C_{orders}}{F_{max}} \tag{8.8}$$

$$L_{max-decision} = N_{order-batches} \times \frac{C_{metric} + C_{orders}}{F_{max}} \tag{8.9}$$

$$L_{avg-decision} = \frac{L_{min-decision} + L_{max-decision}}{2} \tag{8.10}$$

8.2.2 Performance Analysis Results

Using the work in subection 8.2.1 with the implementation data from chapter 6 we can produce performance results. Setting both B_{orders} and $B_{entries}$ to 10 in our implementation, we can calculate the number of cycles required to perform certain operations. This is possible because only a small set of actions in Handel-C require a cycle. The results can be seen in top half of table 8.1. Exporting the implementation into Xilinx's ISE we can place and route the design to find out the resources that are used by the implementation and calculate $N_{engines}$. Table 8.2 shows how many implementations of an algorithm the Xilinx Vertex 5 xc5vlx30 chip can support with its total 4800 logic slices available. Algorithms can be used multiple times on a single chip, so a realistic number of algorithms available is 6, which is possible unless all algorithms were to be the TWAP algorithm. Therefore we shall use 6 algorithms for all further calculations, it is advisable not to use 100% of available system resources. This is because as system resource usages gets closer to 100% there is performance degradation seen in the system. The number of engines supported is calculated by taking the floor of the amount of each resource available divided by the amount used, then the minimum number from these calculations for each algorithm is selected as the maximum number of engines possible on the device.

Xilinx ISE produces the Critical Path Delay (CPD) of the system, from which the theoretical maximum clock speed of the device can be derived. Although this can only be used as an upper limit, using the same chip as mentioned above, ISE calculated a CPD of 1.765ns which equates to 566.6MHz. With F_{max} set to 566.6MHz we can calculate the value of the throughput equations from subsection 8.2.1, this estimates maximum performance. The mixed throughput equation results are best shown as a graphs because $N_{order-batches}$ varies as well. Figure 8.1 and 8.2 show both the order and update throughput with the number of order batches varying between 1 and 100000. The number of orders being processed remains roughly the same no matter the number of batches being processed. As expected the number of updates being processed drops steadily as the number of batches increases. This is due to large batches of orders taking more time to process a single update. The figures show the average performance across the three algorithms. The bottom half of table 8.1 shows the maximum throughput of each of the algorithm independently. It is clear to see that the participate algorithm is the quickest, followed by the VWAP and finally the TWAP algorithm. The update throughput is always an order of magnitude less than the order throughput, this is because orders are processed in batches of ten, where updates are processed individually. The worst case scenario is that there is only a single order being managed and so nine computational slots of the order batch are unused and wasted.

The software implementation was executed on the development machine which is accepted as having reasonable performance. Tests were executed as described in section 8.1, they were also executed on a number of other machines to compare performance and consistency between the different processors. The development machine performed the best and all machines produced consistent results. One of the main reasons for this was probably because the system had been developed and optimised for that environment, secondly all other machines tested were shared with other users and their activities could have had serious effects on performance. The tests were executed multiple times with an average execution time of 30ms for when processing for the maximum number of updates. As the processor used has two physical cores it is possible to run two executions simultaneously together, much like running multiple algorithms together on a single FPGA chip, this enables us to double the number orders and updates processed in the amount of time. The software results for the worst case scenario found the maximum number of updates that could be processed per second was 1,390,000 per process executing. On the dual-core test machines the result was double, at 2,790,000 updates per second.

The frequency that the hardware runs at must equal or improve upon performance than that of the software implementation. This frequency or F_{equal} must also be less than 566.6MHz which is the theoretical maximum calculated from the CPD. Using equation 8.4 rearranged to find F_{max} given $T_{max-updates}$ as the maximum performance of the software implementation, to equal the software's performance the hardware would have to run at 2.5MHz. Even after having to factor in other performance degrading operations that would be required in a production system, this is an obtainable number. It should also be possible to run in excess of this frequency to increase performance. Figures 8.1 and 8.2 graphically show the difference in performance between the software and hardware implementations. It also shows that the hardware version tracks consistently against the software, showing no difference in scaling. Our results show that the hardware implementation is 377 times faster than the software version which is a promising increase in performance. Figure 8.3 shows performance scaling of the two systems as clock speed is increased. Additionally I have plotted both the software and hardware maximum performances on the x-axis. The important point is the intersection of the *Hardware* curve and the *Software Maximum* line. This crosses at the previously declared 2.5MHz, it can be seen that that this is very early on in the graph. Its worth noting that although the *Hardware* curve carries on past the *Hardware Maximum* line, this performance cannot be reached as this device is unable to reach those clocks speeds.

I will now look at the life expectancy of each systems. Obviously any system created should last a sufficiently long period of time, this will offset any initial costs. The initial cost of the hardware (omitting any failures which require replacement parts) and the cost of development are spread over a much longer period of time, making the system more attractive. We have two different directions which we have compared the two implementations, the number of orders processed and the number of updates processed per second. Statistics on either of these figures from current markets is very limited, especially with order quantities and no metrics on the number of parent orders being managed by systems. Thankfully there are some metrics for market data update throughput, I have taken those used here from *Market Data Peaks* [55]. They consolidate updates from 15 American markets, so they present a rather extreme, but valid view of work load. Taking some of their data on market peak message rates, we can extrapolate out behind and in front of the data to provide a roadmap of how peak update throughput will look in

Calculation		Algorithm Name		
		Participate	TWAP	VWAP
Cycles	Update	31	31	31
	Event	41	41	41
	Metric	13	13	13
# Engines		7	5	6
Throughput	Updates/s	10500000	10500000	10500000
	Orders/s	138000000	138000000	138000000
	Max Updates/s	73400000	52400000	62900000
	Max Orders/s	967000000	690000000	829000000

Table 8.1: The number of cycles each activity and maximum processing throughput.

Resources	Algorithm Name			xc5vlx30 FPGA
	Participate	TWAP	VWAP	
Slice Logic Utilization	Used	Used	Used	Available
Number of Slice Registers	544	560	562	19200
Number of Slice LUTs	1732	2473	2256	19200
Number of Occupied Slices	619	811	770	4800
Number of Bonded IOBs	4	4	4	220
Number of BUFG/BUFGCTRLs	1	1	1	32
Number of DSP48Es	4	5	5	32
Number of Route-Thrus	88	114	102	-
Number of LUT Flip Flop Pairs	2004	2652	2456	-
Number of Engines	7	5	6	-

Table 8.2: Resource usage for the implementations on the Xilinx Vertex 5 xc5vlx30 FPGA.

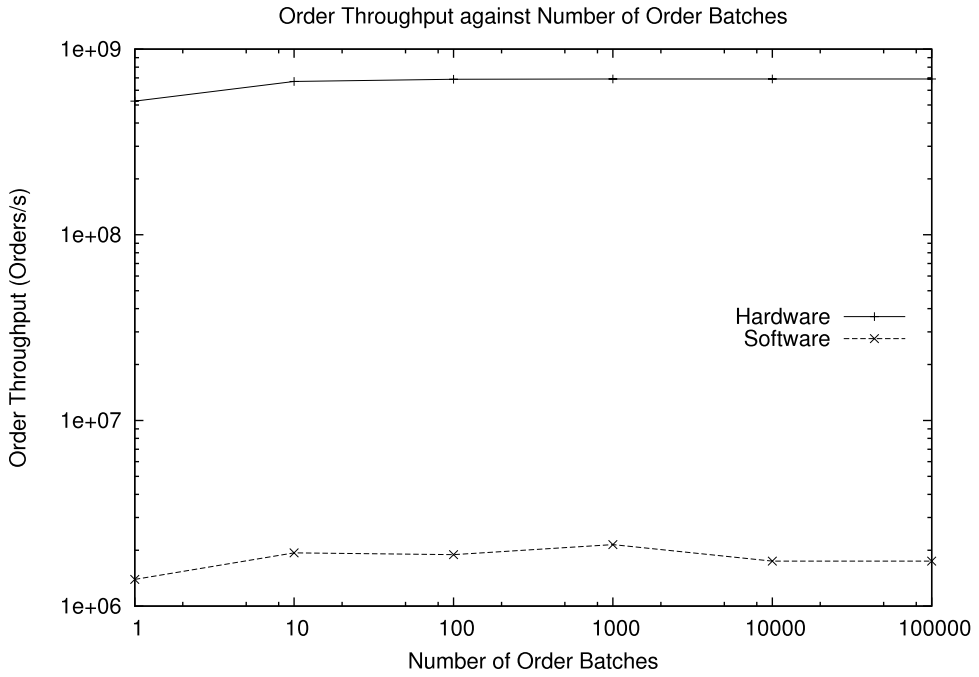


Figure 8.1: Order throughput while the number of order batches increases.

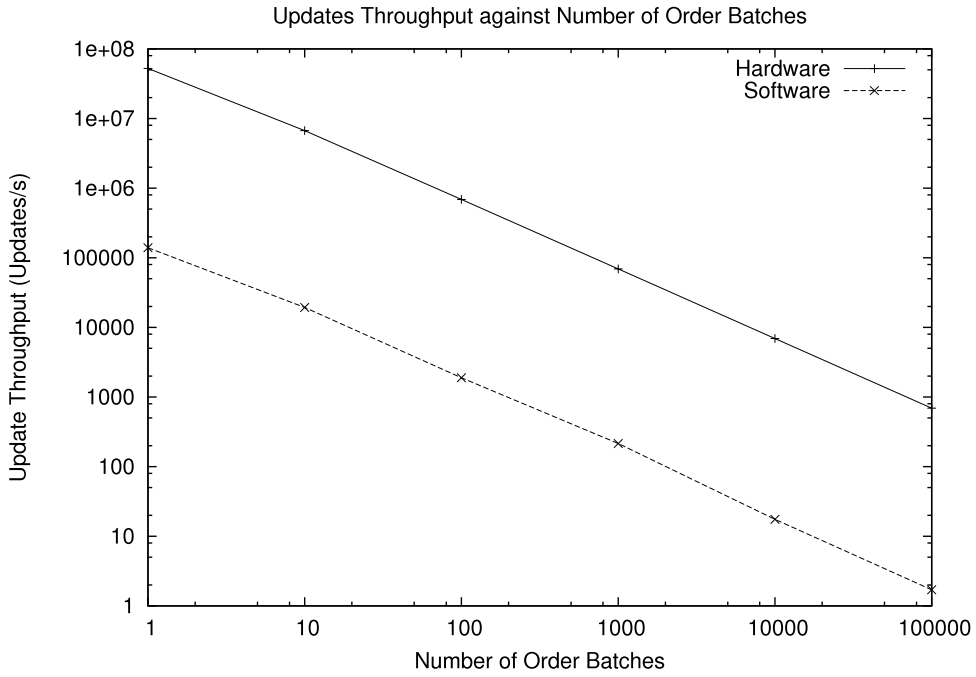


Figure 8.2: Update throughput while the number of order batches increases.

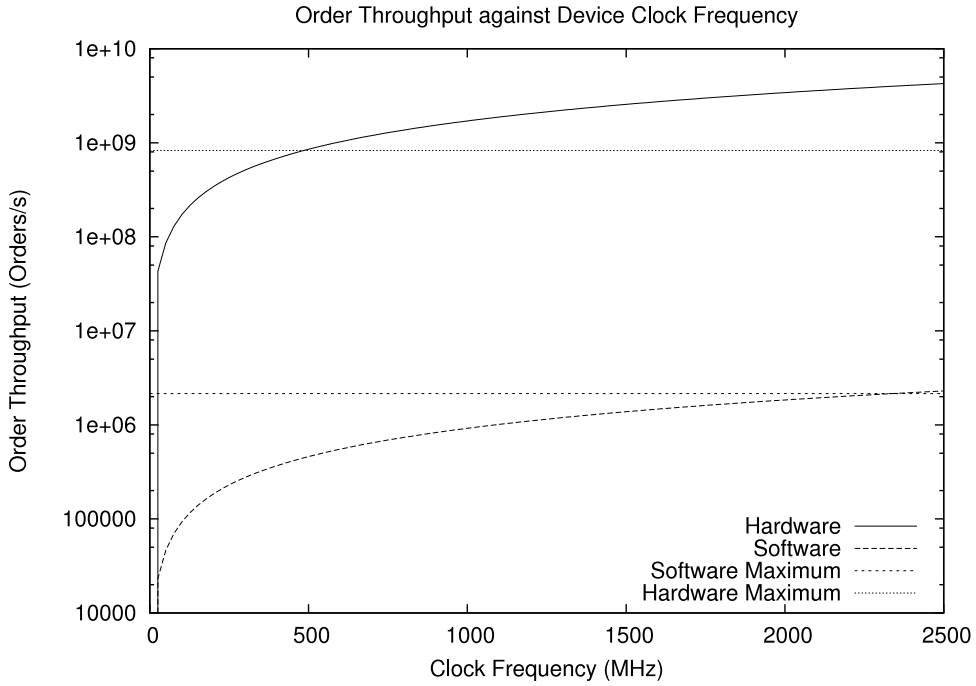


Figure 8.3: Update throughput while the number of order batches increases.

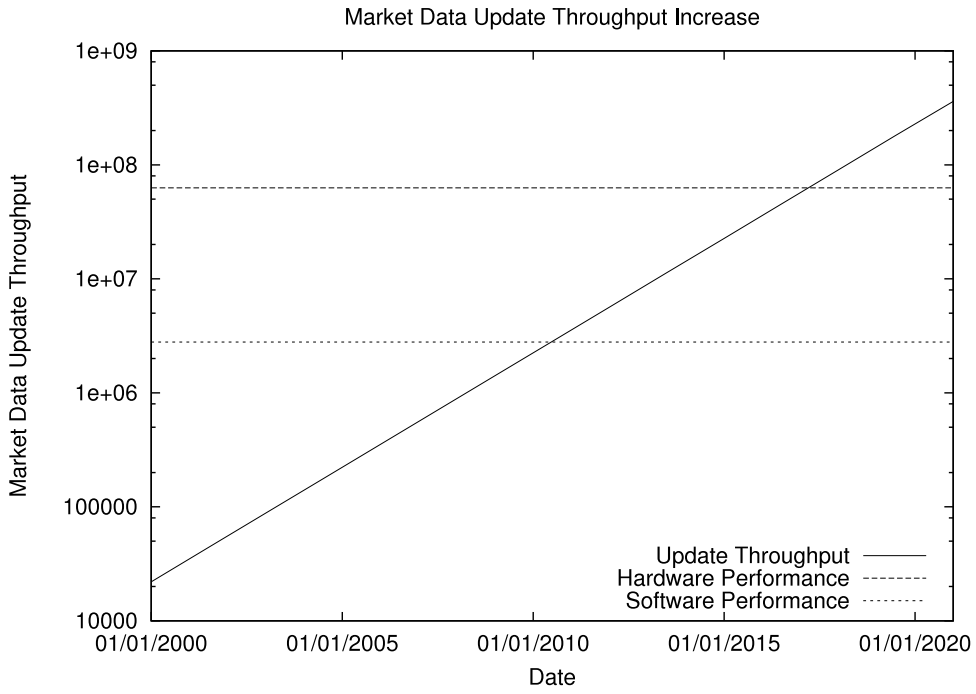


Figure 8.4: System performance against market data throughput increase over time.

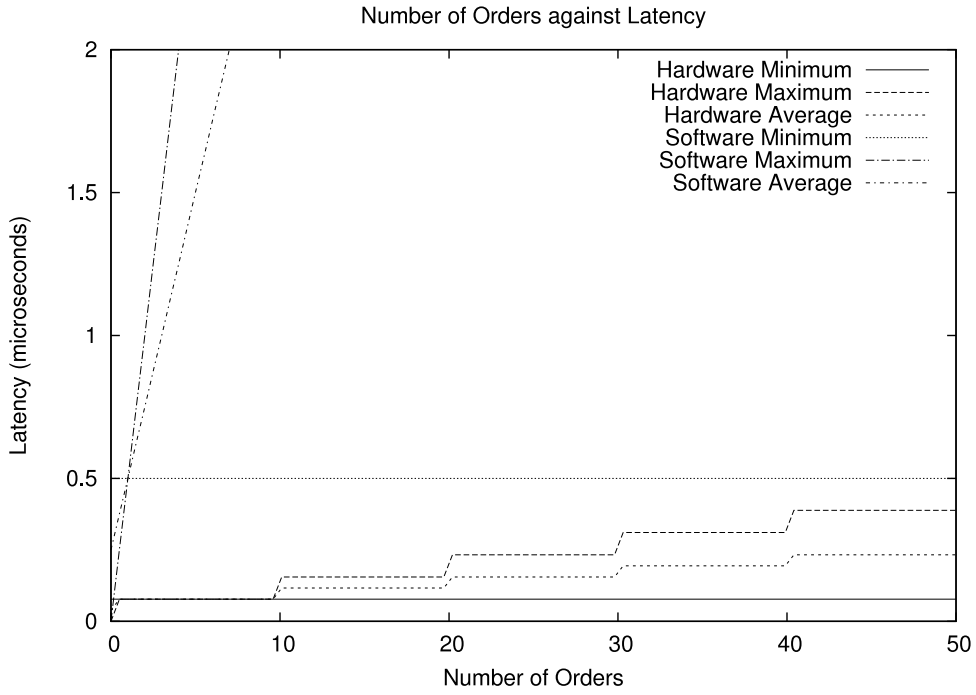


Figure 8.5: Order processing latency against number of orders being processed.

the future. This is shown as the diagonal line in figure 8.4. On this graph I have also plotted the current hardware and software performance as straight lines on the y-axis. The intersection point shows where expected market data peaks will eclipse the performance of the system. It is clear that the software implementation is already running into this problem with the expected intersection towards the end of this year (2010). The hardware implementation on the other hand looks to be able to cope until 2017, which gives the system a life expectance of 5 to 6 years conservatively. The assumption of market data throughput growth is that it doubles every 18 months, much like Moore's Law on silicon performance. You could argue then that the software performance would remain just under that of the market data throughput, however this would require the host for the software to be replaced frequently with the latest processors, increasing the cost of the system. Additionally, with new systems, the software would have to be transferred across and tested to ensure that it still worked correctly. Finally, there would be nearly no headroom available, if there was a sudden market explosion in activity then it would be quite possible for the system to be overwhelmed at a critical time. We can deduce from this that not only is the hardware implementation faster, it also offers much better life expectancy both are equally important in an industrial setting. The closeness of the performance to the current market data throughput of the software could explain why some companies are now evaluating the effectiveness of FPGAs and the media coverage it is now attracting.

Finally we can investigate the latency of processing orders. Figure 8.5 shows the latency difference between the two systems. I have plotted the minimum, maximum and average for both of the systems. The minimum should be the amount of time it takes to make a decision for a single order, the maximum is the amount of time until the last order has a decision. The average is taken between them as an expected value for the latency of the system. What we can see in the graph is two horizontal lines, these are the minimum latencies for the systems. Next you'll notice the stepped lines, these are a characteristic of the hardware implementation with its batch processing functionality. The final point is the two lines to the left of the graph with very steep gradients. These are the average and maximum for the software implementation, it shows why there is such a performance difference between hardware and software, with the hardware possessing a significantly shallower gradient. What is also interesting is that we can see that the performance difference between the minimum latencies is much less than the overall throughput performance. The hardware's minimum latency is 0.08 compared to the softwares 0.5 microseconds, therefore the hardware is 6.25× faster. Considerably less than overall performance, but still very welcome and should allow the algorithm better price performance.

8.3 Partial Run-Time Reconfiguration Testing

I will now take this opportunity now to evaluate the possibility of using using partial run-time reconfiguration to increase performance further. Like with performance testing, I will first present an technology independent mathematical model for analysing any number of hardware implementations. This ensures that the work presented here is as general as possible, allowing many other pieces of research or projects to make use of it to analyse their own implementations. I will then use these models to analyse my own implementation and evaluate its own effectiveness. This will be in a number of different scenarios and finally I will draw a conclusion based on looking at how much market data throughput there is during the reconfiguration time to determine if this is a viable option. These statistics will be draw from real-world examples and look at a worse case situation.

8.3.1 Partial Run-Time Reconfiguration Technology Independent Analysis

I now present a simple analytical model for estimating the time to complete a reconfiguration of the target device, exploring both full and partial reconfiguration. The total time of a reconfiguration of the device or T_{full}, can be calculated by equation 8.11. Where S_{config} is the size of the device configuration in bytes and S_{update} is the number of bytes that can be updated every reconfiguration cycle with F_{config} as the reconfiguration frequency in hertz.

$$T_{full} = \frac{S_{config}}{S_{update} \times F_{config}} \tag{8.11}$$

If the device doesn't require to be fully reconfigured, time and performance can be saved by only reconfiguring part of the device while leaving the remaining section of the device to carrying on processing information. If the device supports partial reconfiguration, we can calculate $T_{partial}$ using equation 8.12. Where $N_{columns}$ is the number of columns the implementation requires and N_{frames} is the number of frames in a single column. S_{frame} is the size of a frame in bytes, S_{update} and F_{config} are the same as before. A frame is the minimum amount of configuration information that can be accessed.

$$T_{partial} = \frac{N_{columns} \times N_{frames} \times S_{frame}}{S_{update} \times F_{config}} \tag{8.12}$$

Now we are able to calculate the time that it would take to reconfigure the device given the changes to be made and the capabilities of the device. We can use this information to calculate how many updates would be seen during this time using equation 8.13. This is dependent on the update throughput, which in reality varies constantly depending on market activity, but for the purpose of this project I shall use a selected number of throughput levels. Where $N_{updates}$ is the number of updates seen over the time $T_{reconfig}$ with a throughput of updates of $TP_{updates}$ in updates per second. Each update not processed could signify a missed opportunities to trade or loss of accuracy, therefore it is important that this number is low. Using a column based reconfiguration system we can estimate the number of messages, $N_{messages}$, that would be received using equation 8.14. Where $N_{columns}$ is the number of columns in the reconfiguration, $T_{reconfig-col}$ is the time to reconfigure a column and $TP_{messages}$ is the message throughput.

$$N_{updates} = TP_{updates} \times T_{reconfig} \tag{8.13}$$

$$N_{messages} = N_{columns} \times T_{reconfig-col} \times TP_{messages} \tag{8.14}$$

If the algorithm is required to process all updates, then the user could choose to buffer these on chip to maintain performance. This would be particularly effective if the solution was using a hardware-based market data handler, such as the one presented in [40]. For this we require to know how much storage, $S_{total-updates}$ in bytes, would be required. This can be calculated using equation 8.15, where $N_{updates}$ is the maximum number of updates that we are required to store over the reconfiguration period and S_{update} is the size of a single update in bytes. S_{update} is defined by equation 8.16, where $N_{update-entries}$ is the number of price quantity entries looked at in the update. In batch processing this is equal to the batch processing size of market data updates. $S_{data-representation}$ is the size of the internal representation of data used in bytes, for example in a 32 bit integer representation, 4 bytes are used. The factor of two signifies that each entry is a pair, representing the quantity and price of the entry.

$$S_{total-updates} = N_{updates} \times S_{update} \tag{8.15}$$

$$S_{update} = 2 \times N_{update-entries} \times S_{data-representation} \tag{8.16}$$

8.3.2 Partial Run-Time Reconfiguration Analysis Results

The design and implementation process was the same as before, compiling into EDIF before exporting into Xilinx ISE to place and route the design. Remaining consistent, I will carry on using the Xilinx Vertex 5 xc5vlx30 FPGA as the target device to produce the results using the independent analysis in subsection 8.3.1. The xc5vlx30 is made up as an array of 80 rows and 30 columns, there are a total of 4,800 slices available [56]. There are a total of 8,374,016 configuration bits and there are 6,376 configuration frames and each frame has 1,312 bits [57]. On average a single column contains $4800/30 = 160$ slices or 212.5 frames. The xc5vlx30 can have its reconfiguration clock clocked up to 50MHz and the reconfiguration data path is 16 bits, providing up to 800 Mbit/s transfer [57]. Using this information, the analysis and the information from our design, we can calculate the time to reconfigure the target device under a number of scenarios.

Table 8.3 describes the three different scenarios and links them to their associated graphs with the results of the scenario. Table 8.4 shows the results from the variations created from the existing algorithms,

Scenario	Details
#1 (Figure 8.6)	Varying the order batch size, increases the order throughput processed. This is useful in times of high market activity, when there are a large number of orders. Decreasing the batch size will allow the algorithm to cope with a higher number of updates.
#2 (Figure 8.7)	Varying the update batch size, increases the accuracy of decision making by processing more of the update. This increases resource usage on the device, unused space is wasted space. Therefore it is beneficial to keep the batch size as close to the actual number of entires arriving in market data updates.
#3 (Figure 8.8)	Varying both order and update batch size, this delivers the advantages and disadvantages of both the previous scenarios, depending on the size of the variation.

Table 8.3: The three different scenarios available when reconfiguring the device.

	Participate			TWAP			VWAP		
	N_{slices}	N_{cols}	$T_{partial}$	N_{slices}	N_{cols}	$T_{partial}$	N_{slices}	N_{cols}	$T_{partial}$
$A_{5,5}$	512	4	0.001394	823	6	0.002091	726	5	0.001743
$A_{5,10}$	538	4	0.001394	791	5	0.001743	672	5	0.001743
$A_{5,15}$	462	3	0.001046	846	6	0.002091	762	5	0.001743
$A_{5,20}$	558	4	0.001394	710	5	0.001743	659	5	0.001743
$A_{10,5}$	665	5	0.001743	868	6	0.002091	826	6	0.002091
$A_{15,5}$	641	5	0.001743	1,006	7	0.002440	870	6	0.002091
$A_{20,5}$	770	5	0.001743	896	6	0.002091	1,135	8	0.002788
$A_{10,10}$	662	5	0.001743	802	6	0.002091	736	5	0.001743
$A_{15,15}$	718	5	0.001743	820	6	0.002091	980	7	0.002440
$A_{20,20}$	737	5	0.001743	1,112	7	0.002440	1,062	7	0.002440

Table 8.4: Algorithms size and reconfiguration time, varying by order and update batch sizes.

their size and time to complete a partial reconfiguration of the device by placing that implementation. The graphs linked by table 8.3 are figures 8.6, 8.7 and 8.8. These show the difference between how the algorithms scale against each other in the different scenarios. The graphs show the implementation space requirements when placed on the target device. In nearly all of the graphs the resource utilisation increases as the batch size for the order and or update batch increases. This increase is not linear though and in fact can sometimes decrease with larger parameters. The only explanation that I can provide to this is that compliers optimisation procedures are more effective when presented with a larger implementation. The worst offender appears to be the TWAP algorithm, which is also the largest algorithm. When increasing the order and update batch size the increases seem to be more apparent. This leads to me to assume that when increasing both parameters, the implementation size is rising significantly enough even with optimisation it cannot be made smaller than a previous implementation. What is also interesting is that the increases of the algorithms do not scale similar when compared against each other. This possess the question of whether some implementations are more efficient than others.

Figure 8.9 shows the stepping in the reconfiguration time because the reconfiguration takes place column by column and the size of the reconfiguration determines the number of columns that need reconfiguring. From this information we can deduce how many messages would be missed or need buffering. A historic maximum peak in market data information can be take as 2,526,103 messages per second recored on February 11, 2010 [55]. This peak is the aggregate of 15 american markets and it would be extremely unlikely for a single system to require data from all of these venues. We can deduce the number of messages missed per column by omitting $N_{columns}$ from equation 8.14. In this case we can infer that with a reconfiguration time of 0.00035 seconds be column, ˜880 messages would be missed per column needing reconfiguration. It is worth noting again that this is for 15 venues and thousands of symbols. In reality a system would only be processing a small fraction of these updates and not receiving the rest. Slow stocks (those that do not receive much activity) would normally have no updates within this reconfiguration time. Busy stocks may receive a couple of updates, using a 32 bit data representation and an update entry batch size of 10, each update requires 640 bits to store. This would allow 2MB of

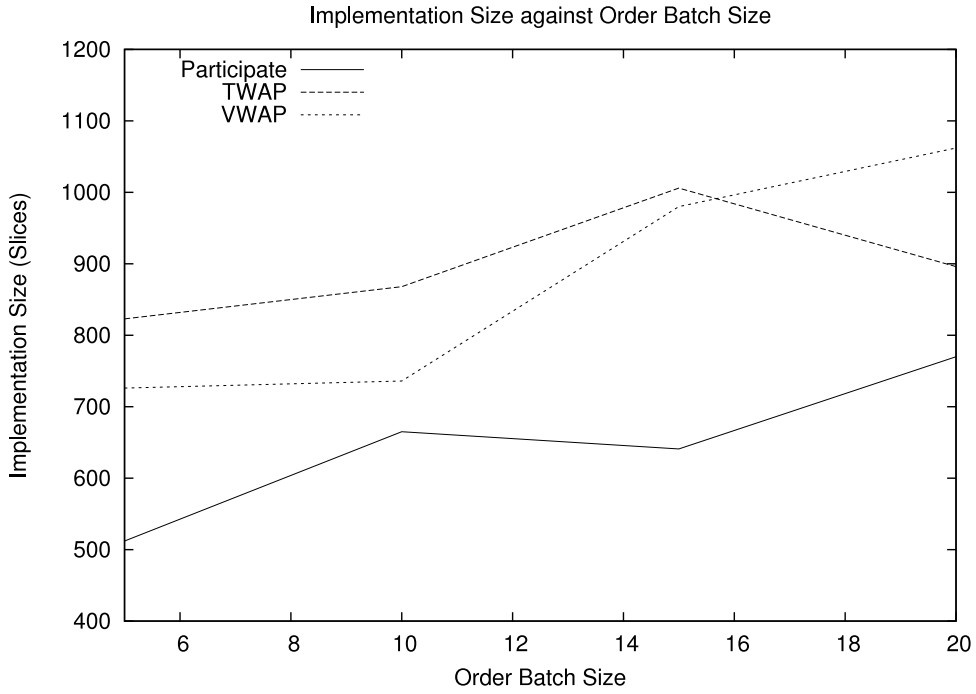


Figure 8.6: Implementation size against order batch size increasing.

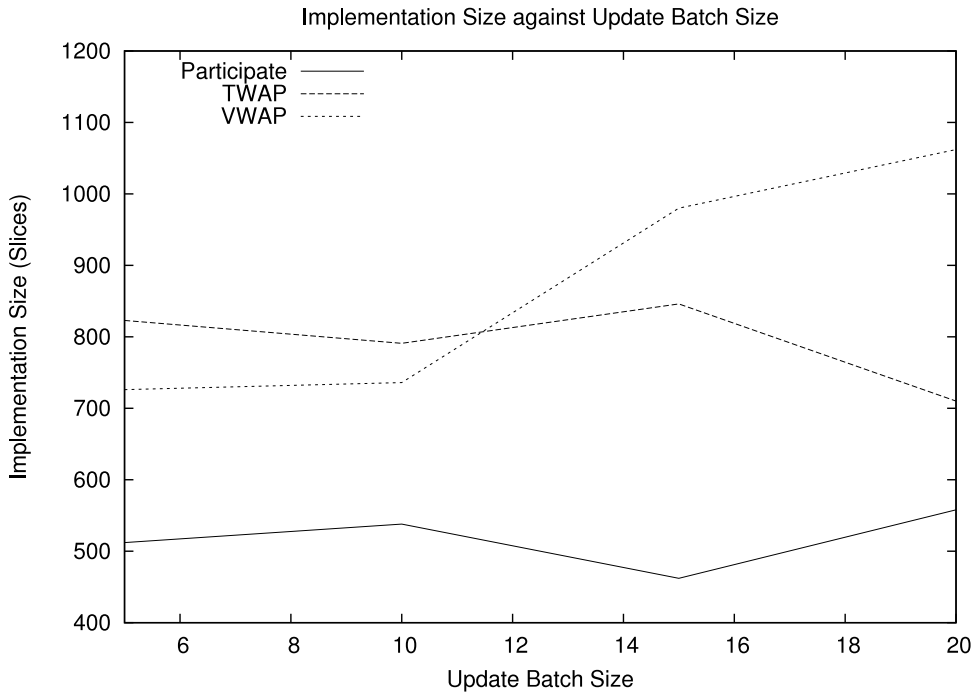


Figure 8.7: Implementation size against update batch size increasing.

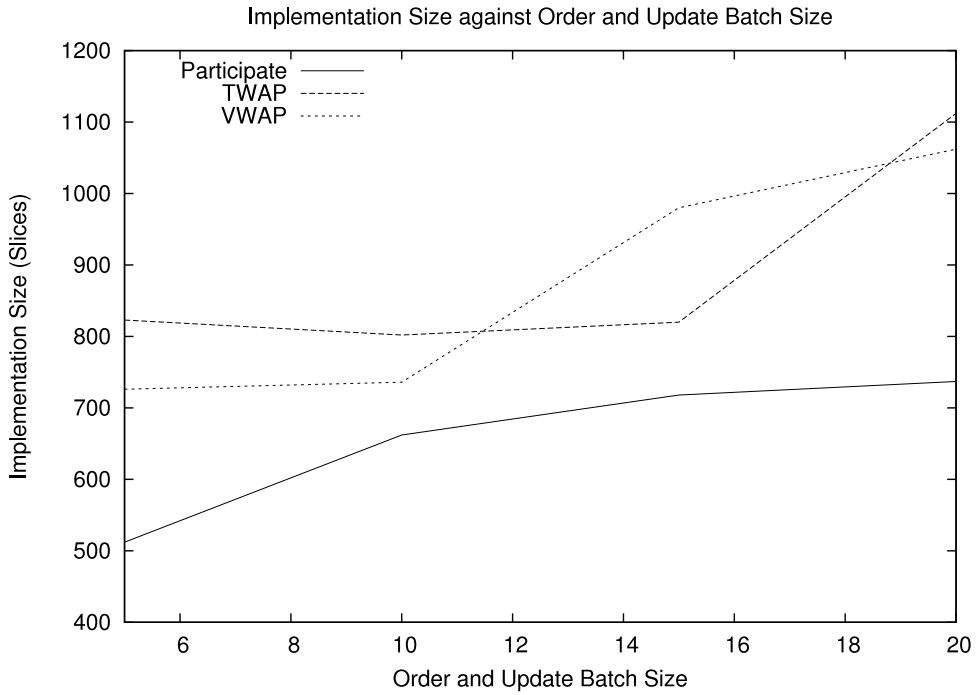


Figure 8.8: Implementation size against order and update batch size increasing together.

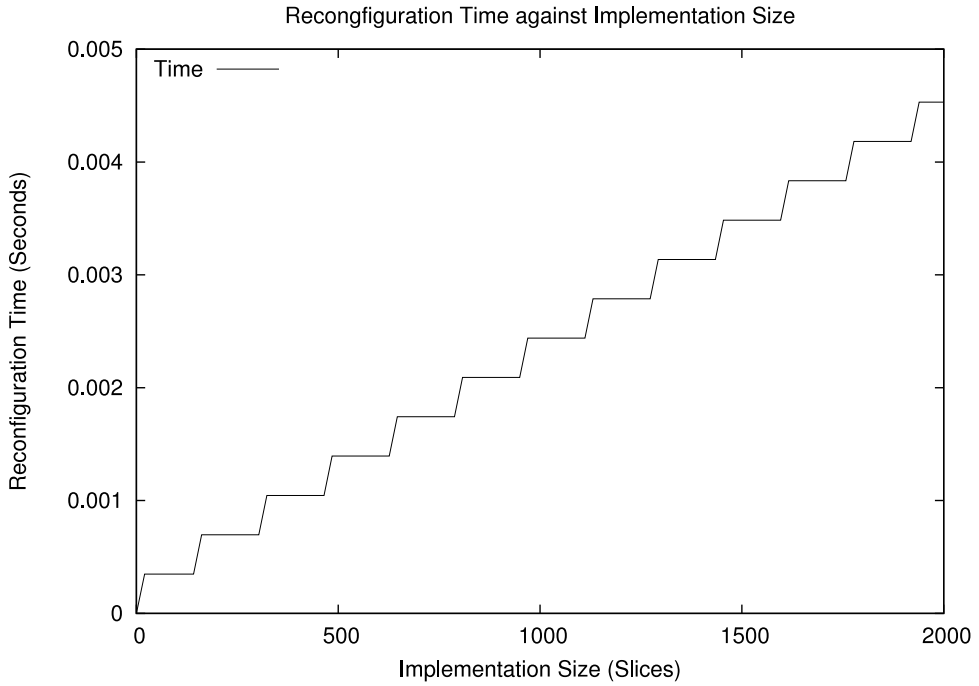


Figure 8.9: Reconfiguration time based against the number of slices being reconfigured.

Manufacturer	Name	Code	Cores	Frequency	V_{min} (V)	V_{max} (V)	Power (W)
Intel	Core 2 Duo	T7600	2	2.33GHz	1.0375	1.300	34
Intel	Xeon	3065	2	2.33GHz	0.8500	1.5	65

Table 8.5: Power consumption of the Intel processor used for testing with comparison.

		Algorithm Power Consumption (W)		
		Participate	TWAP	VWAP
Configuration	$A_{5,5}$	0.603	0.626	0.617
	$A_{5,10}$	0.600	0.624	0.615
	$A_{5,15}$	0.600	0.623	0.616
	$A_{5,20}$	0.603	0.627	0.619
	$A_{10,5}$	0.612	0.634	0.628
	$A_{15,5}$	0.623	0.644	0.648
	$A_{20,5}$	0.633	0.654	0.648
	$A_{10,10}$	0.610	0.632	0.625
	$A_{15,15}$	0.621	0.643	0.636
	$A_{20,20}$	0.635	0.656	0.650

Table 8.6: Power consumption of algorithms when varying their order and update batch sizes.

storage to store 25,000 updates. This doesn't include the information required to categorise updates such as stock name. As this would only be required once per category I have omitted it here as it is small.

8.4 Power Consumption Comparison

Power consumption is of growing importance because of change of attitudes to the environment. No longer can companies afford to pay large electricity bills supporting huge compute farms or the public image of appearing to be wasteful with electricity. Therefore it would be advantageous for our new solution in reconfigurable hardware to be more power efficient than the software solution. We already know that this is likely to be the case as FPGAs are known to be extremely power efficient and from the previous analysis, we know that it holds a sizeable performance benefit over the software implementation.

Some of you may question the use of a mobile processor in the software testing. This has advantages as processors designed for laptops are normally far more power efficient than their desktop or server brothers. This is because of the limited space for cooling, ensuring that the laptop does not weigh too much or is too large. Secondly as laptops are designed to run off battery power, a more power efficient processor will be able to run longer on a battery or for a shorter period of time on a smaller battery. Which in turn would make the laptop smaller and lighter. To show this effective I have compared the processor used in testing with an equivalent server process also supplied by Intel, in table 8.5. The information form this table is taken directly form Intel [58]. The processor used is on the top row, with the server processor below it. You can see that they are very similar in specification, the same number of cores, clock speed, amount of cache and lithography size. Additionally they were released at the same period of time, Q3 '07 and Q4 '07. It would be expected that the performance would be very similar, but that the server processor will probably have the edge on a number of activities because it will have a few more features designed for high performance computing. Crucially though, the power consumption of the sever processor is nearly double (1.9×) that of the mobile processor. As we are certainly not expecting the server processor to have that kind of performance advantage over the mobile processor, the mobile processor will give us a better performance per watt than the server processor could. The use of mobile processor in high-performance computers is not a new idea, research in this area has been performed by a number of parties, including CERN [59]. Many commercial server companies are now also offering their own solutions in this space as well, such as Kontron [60]. Both of these use Intel's Atom processor which is primarily designed as a cheap and lower powered processor for very small laptops. However they have been found to be very effective in the area of power efficient computing.

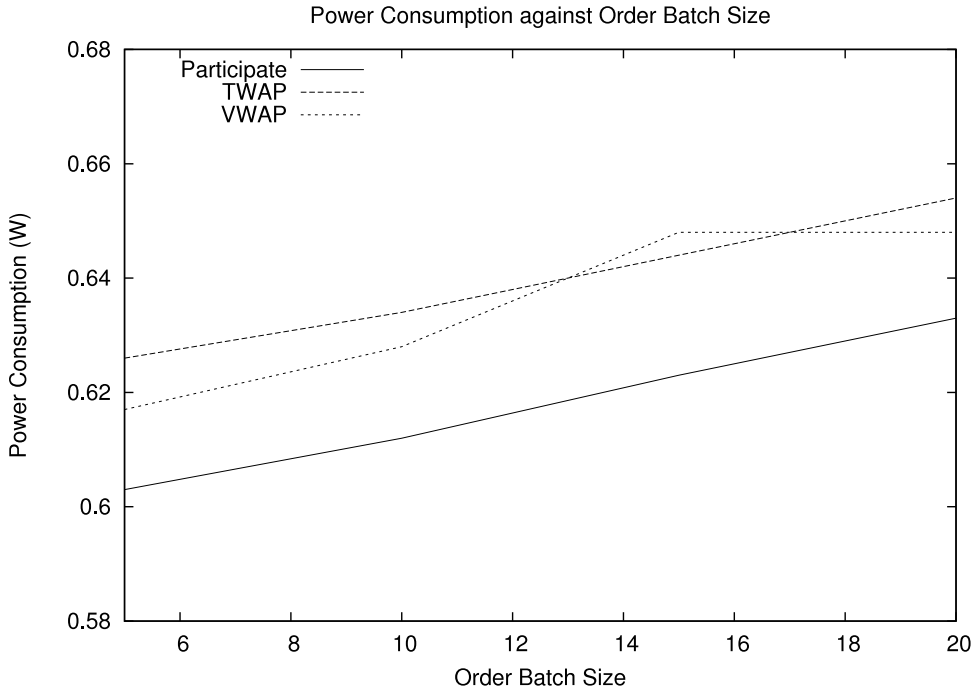


Figure 8.10: Power consumption size against order batch size increasing.

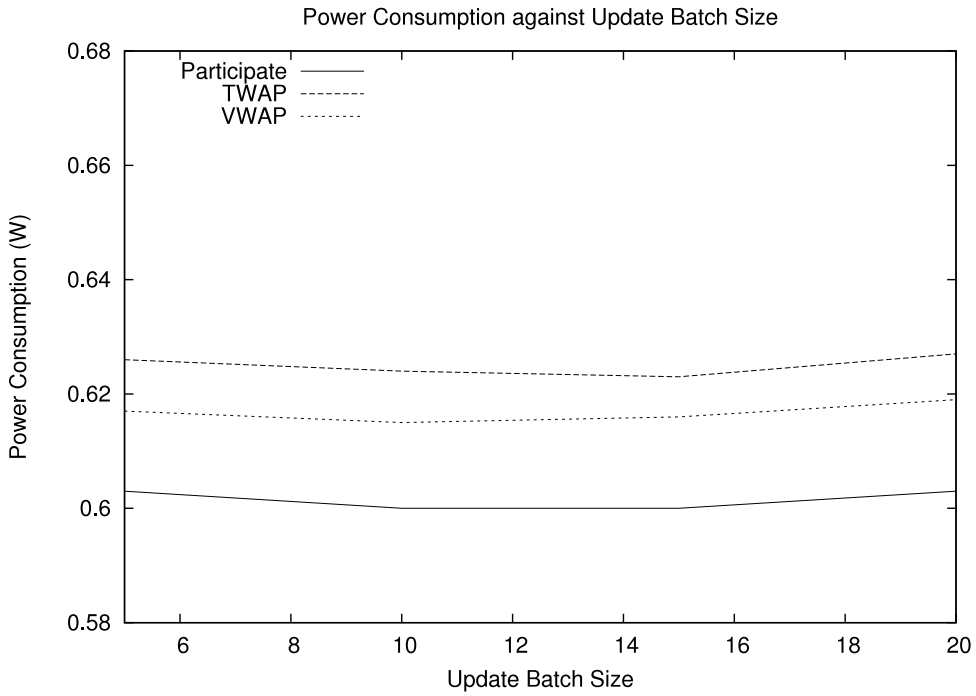


Figure 8.11: Power consumption against update batch size increasing.

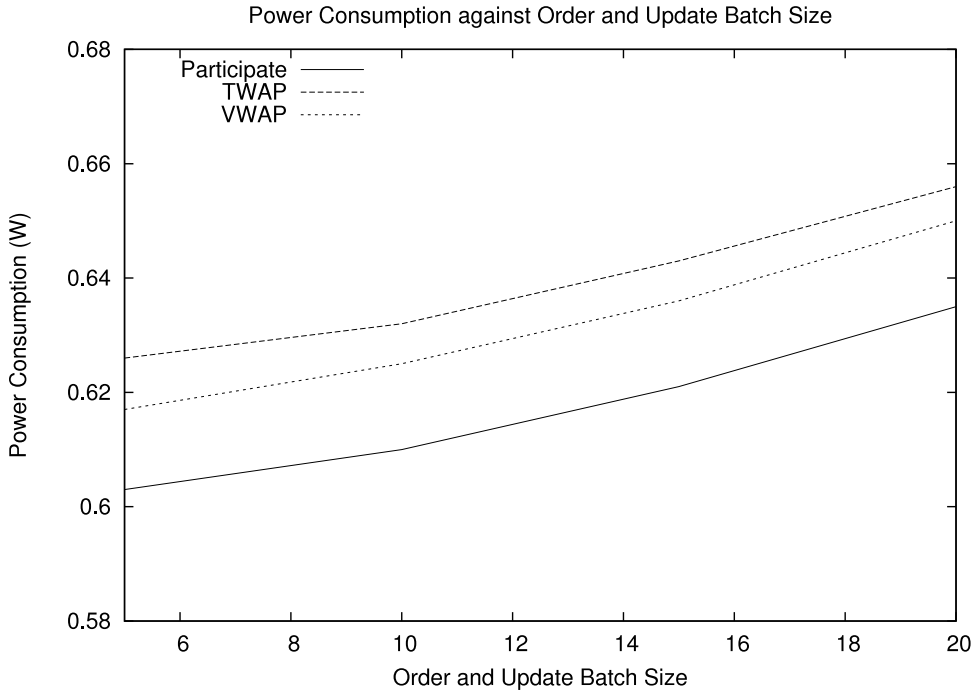


Figure 8.12: Power consumption against order and update batch size increasing together.

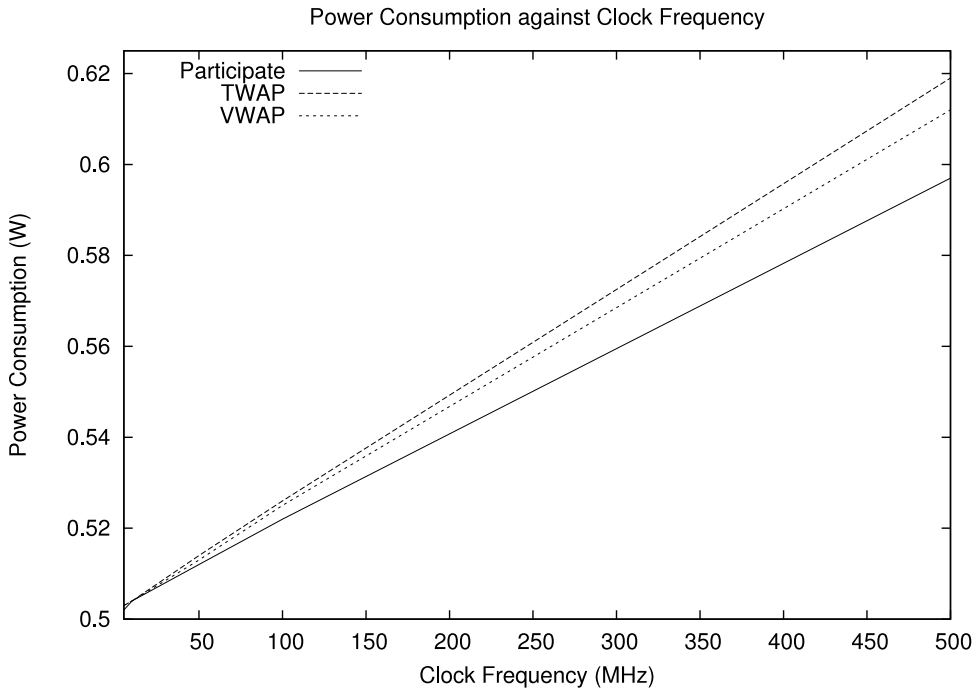


Figure 8.13: Power consumption against clock frequency

Knowing the maximum power consumption of the software implementation, we now need to look at the power consumption of the hardware version. For this we can make use of my original designs after they have been placed and routed by Xilinx ISE. This produces a map report of the implementation which contains large amounts of information about how it is expected to perform. Xilinx provides a spread sheet [61] for calculating the power consumption of implementations by loading in the map report. You can also modify a number of parameters, such as the device clock speed which affects the power consumption. The results are at the maximum clock speed previously calculated to keep results consistent. The results from this experiment can be seen in table 8.6 and I have looked at all of the algorithm configurations which were studied in the partial run-time reconfiguration analysis. Additionally I have plotted this information in three graphs found in figures 8.10, 8.11 and 8.12. The first interesting result is that increasing the update entry batch size does not appear to increase power consumption. This ties in with the results in implementation size which show that increasing the update batch size doesn't appear to have a trend of increasing implementation size. It can be seen that when increasing the order batch size though we get noticeable increases in power consumption. It is scaling linearly and at a rate which is acceptable. Another interesting point is that the power consumption usage graphs are much smoother than their associated implementation size graphs. This adds weight to the idea that the compiler can carry out better optimisations on implementations of larger parameters. The amount of logic that is being used is increasing, but the computational density is better.

With these performance figures, we can look at the performance per watt for each of the implementations. Using the throughput metric of updates processed per second we can find out how many updates can be processed per joule of energy used. We can use the maximum figure of power consumption for the CPU as testing was at maximising performance, preventing the CPU from performing any power saving activities, such as reducing its operating voltage. For the software solutions we have a figure of $2790000/34 = 82000 updates/joule$ and for the hardware solution $62900000/((0.610 + 0.632 + 0.625)/3 \times 6) = 62900000/3.734 = 16800000 updates/joule$. This points to an improvement per watt of $204\times$, where the value of 3.734 was calculated as the average power consumption for an algorithm with batch sizes set to 10 and 6 algorithms placed on the device. These result are within the same order of magnitude as the previous performance related results, showing just how much of an effect that performance has on the other attractiveness features of the hardware implementation. As a final experiment, I have looked at how the algorithms scale with a varying clock speed. The results are shown in figure 8.13 which shows that power consumed grows linearly with clock speed as well. This means that we don't expect to find a better performance per joule figure as performance also linearly scales with clock speed. This means we can be safe in the knowledge of knowing that we should run our devices at the maximum clock speed supported.

8.5 Cost and Space Effectiveness Comparison

Even though we have seen that reconfigurable hardware looks like an attractive option for accelerating trading algorithms, there is still at least one criteria they must meet before they are to gain momentum in industry. That is their cost effectiveness. The first disadvantage is that we require a host machine to support the FPGA board and this will execute the software layer of the application. This means that our cost will already incur the same cost as not using the FPGA plus the additional cost of the FPGA. With further work, moving the whole application down into hardware, a FPGA board that is independent of a host could be used. This is a board which provides its own network interface and then is run separately. This may not be the most attractive solution though as it would limit the amount of insight into a running system. Probably the best solution is use a host machine which would then support multiple FPGA boards internal, this is available because most FPGA boards connect through the PCI or PCI-Express bus on a system board. Many system boards provide up to seven slots available. However there may be limitations to using all of these, such as bandwidth and cooling issues. In the most powerful systems utilising a large number of *graphic processing units* (GPUs) which are very power hungry and hot, a maximum of four can be placed within a system, using a single slot each but the space of two. For this reason we will assume that this is also the maximum number of FPGA boards that could be placed within a system reliably.

The second problem that we have to address is that of initial cost. What I will present is the most readily available data on cost. These are for development boards which most likely wouldn't be used for a

production ready installation. They would be required while developing and testing the implementation, but they have extra features which aren't required by my design that increase their price. To put this in perspective, the Xilinx Vertex-5 xc5vlx50t FPGA chips, which is 50% larger than the xc5vlx30 that we have previously been studying is available $550 to $600 each. This doesn't include the support board which could cost the same again, especially if specially developed for this application. The xc5vlx50t is also available from Xilinx as a development board, with built in ethernet controller, this however costs $2,000. Which is approximately double the cost of the specialised board. This raises the cost significantly of the system. The other factor we can't take into account is buying the components in bulk, which a company setting up an electronic trading system in reconfigurable hardware would surely do to safe further money.

We can build an abstract representation of this information though which can be manipulated to evaluate the cost and space effectiveness of the these implementations. I propose that cost value of the two systems can be calculated by the performance of the respective system over its initial cost, disregarding any additional running costs or assuming they are identical between the systems. As the performance and cost scales differently between the different implementation, I present an equality to allow comparison between the different systems shown by equation 8.17. Where $P_{software}$ and $P_{hardware}$ is the raw performance of the two implementations respectively, $N_{processors}$ and N_{fpga} is the number of computation units available in each of the systems. S_{fpga} is the size of the hardware computational unit and S_{engine} is the size of the algorithm placed on it, taking the floor of this division allows us to find how many algorithms can be placed on a single FPGA. C_{host} is the cost of the host system and C_{fpga} is the cost of each FPGA board used. We need this equality to be true to make the cost of the FPGA effective against using a conventional software based solutions.

$$\frac{P_{software} \times N_{processors}}{C_{host}} \quad < \quad \frac{P_{hardware} \times N_{fpga} \times \lfloor S_{fpga}/S_{engine} \rfloor}{C_{host} + (N_{fpga} \times C_{fpga})} \tag{8.17}$$

Given the numerator of the equation finds absolute performance, we can tell that when comparing these two values that the cost provides a scaling factor. As the hardware cost value shares the cost value of the software implementation, it can be see that the additional cost of the FPGA provides the scaling factor. The worst case scenario is with the fewest possible number of FPGAs available, which is of course 1. This results in the smallest difference between the cost of the host and the cost of the FPGAs, meaning the cost of the host has more of an effect on the calculation. Using our previous results we can now calculate the maximum value compared to the cost of the host that can be spent on FPGAs while keeping them cost effective for their performance. As we will be comparing against the price of the xc5vlx50t board which is 50% larger than the xc5vlx30, the number of engines deployable now increases to 9 from 6. Given that the majority of servers are dual-processor systems and the prevalences of quad-core processors currently, it is safe to assume that the software implementation will have 8 processing cores available, taken as being roughly the same performance each as a single core on the test machine. Applying this information to our equations and with a little rearrangement, we end up with equation 8.18.

$$C_{fpga} \quad < \quad 332.33 \times C_{host} \tag{8.18}$$

This result shows that the cost of the FPGA used must not eclipse 332.33 times the value of the host system. If it were to, then it would be more cost effective to purchase hosts to run the software until the same performance was reached. This isn't taking into consideration the space required by those systems and the cost of running them though. We know previously that we can expect to pay about $2,000 for the FPGA board, this means to remain cost effective, the host server would need to cost less than $6, this is an impossible target and shows how cost effectiveness of reconfigurable hardware.

We can now carry out the same process but looking at the space utilised by the systems running these implementations. Space is becoming more of an issue for companies with a large computing footprint. Data-centre space is often very expensive, additionally many of these servers will be located as close as possible to the exchange where they are sending orders. This is to reduce WAN costs and latency between the server and the exchange, in fact some of the small exchanges now offer co-location facilities next to their servers. This means that the space required by the implementation is more important than ever. Again we can use an equality to determine at which point either of the implementations becomes

more attractive, this is shown in equation 8.19. The new variables $U_{software-host}$ and $U_{hardware-host}$ represent the rack space in rack units required by each of the implementations. The equality looks very similar to that used for cost effectiveness, this is because we are again comparing it against performance. The main difference this time is that the hardware implementation isn't dependent on the software space used. A rack unit is a measure of space in a standard server rack. A full-height rack is normally 42 units, which is limited by the standard height of a room ceiling.

$$\frac{P_{software} \times N_{processors}}{U_{software-host}} < \frac{P_{hardware} \times N_{fpga} \times \lfloor S_{fpga}/S_{engine} \rfloor}{U_{hardware-host}} \tag{8.19}$$

We can use the same information for performance as before, taking the worst case scenario of only a single FPGA board being utilised. This results in the equality shown by equation 8.20. Knowing the height required by the FPGA implementation we can calculate the space required by the software implementation to be able to reach a greater computation density. As the FPGA boards available are full-height PCI-Express board, they require at least a 4 unit height server, the result is that the software implementation would have to occupy only 0.012 units. A single rack unit is 44.45mm (1.75 inches), this would mean the software server would be required to be less than 0.53mm high, an impossibility. Currently the greatest density server option for dual processor server is a blade configuration which supports 10 servers in 7 rack units, or each occupying 31.1mm, which is nearly 600% larger than required to beat the computational density of the hardware implementation.

$$U_{hardware-host} < 333.33 \times C_{software-host} \tag{8.20}$$

We have looked and discovered that currently the software implementation cannot compete on either the cost or the computational density scales. What are the current comparisons available at the moment. We can expect a host server to use in the cost calculates as roughly \$3000, based on an average server supplied from a manufacturer. As previously mentioned, a suitable development board costs roughly \$2000, this gives a combined cost of \$5000. Opposed to the software implementation of \$3000. Using absolute update throughput performance, the software implementation processes $930 updates/\$$ where the hardware implementation offers $165,800 updates/\$$. This results in an improvement per \$ spent of 178 times, giving the hardware implementation incredible cost effectiveness. Looking at the computational density performance, we can fit a software server in 31.1mm where the hardware implementation will need a full 177.8mm of rack space. This gives us the results of $4,660,000 updates/mm$ for the hardware and $89,000 updates/mm$. This only gives an improvement of 52 times the software implementation, but it is still a sizeable improvement. Both of these figures would be increased when using multiple FPGAs per hardware host.

8.6 Software Analysis

The electronic trading framework and the order manager provide the bulk of the software backbone of this project, although the order manager wasn't used in the final evaluation quite as much as I had hoped. Both of these modules have been developed with the strictest guidelines for software engineering best practises. They exhibit zero cyclic dependencies, allowing different components of them to be taken and used in a large number of other projects. This also aids in reducing maintenance and compilation time, giving the developer a better environment in which to work. The modules are arranged as layers, although they do not adhere to strict layering where a module can only use the contents of the module immediately before it. They practise loose layering where modules can utilise any module that come before them in the chain.

The heavy use of abstract classes and interfaces within the code base provides plenty of anchors for the project to become collaborative. The project has so far solely been developed by me, but this is a highly unrealistic arrangement for development of such a large and wide ranging application. These interfaces would allow other developers to be quickly on-boarded with the project, speeding up development time. Examples include implementing the low level communication methods in the bottom of the framework and developing other electronic trading systems with the package interfaces presented.

These interfaces also provide extension points for use when the framework is picked up by another development time. I have shown through C++'s multiple inheritance feature how simple it is to effectively extend the concrete models provided by the framework so that they contain more specific information and logic. This is also furthered by the design patterns and architectures presented for the trading algorithms. They provide general functionality which can be made use of by the developers or ignored. The developer can pass as much control as they like over to the abstract algorithm, only dealing with the few interesting pieces of behaviour which are critical to their algorithm. The open closed principle states *A module should be open for extension but closed for modification*, this is shown throughout the framework and abstract design patterns. In particularly, it is shown within the hardware implementation with the operation performed depending on the function pointer provided to the architecture. This means that increasing functionality of the framework doesn't require modification, just provided with a different input.

I believe the frameworks and software presented here are easy and intuitive to learn. Some understanding of the business principles being modelled is required but otherwise their layout and structure is very natural. Naming principles are consistent and explicit, the code is also supplied with a reasonable number of unit tests and documentation. This is only the first iteration of the frameworks so I have not had to deal with problems of existing users when making modifications to the framework. Many people do not think that a framework is reusable until it has three clients, this framework doesn't currently have three clients but there are certainly more applications that could benefit from using the framework. Such as *order routers*, *high frequency trading engine* and *post-trade systems*, all of which would gain benefits collectively from using the same framework. Another advantage is that once a developer has learnt the framework they would find it easy to transfer between projects using the framework and get up to speed. This would reduce the amount of time wasted when a developer swaps projects and has to learn new frameworks and libraries. This helps the company using the framework to save money on the salaries of their developers as they are more productive.

In short the software provided in this project projects a great background for being used in the future, adds a design methodology and code structure to projects it is used in and provides all of the initial functionality that is required by electronic trading systems.

8.7 Summary

In this chapter I have evaluated the performance of much of the work presented in this report. I have not limited these metrics to just computational performance but also looked at power, cost and space efficiency gains made by the new implementation. I have presented the implementation and testing details of the experiments that have I have performed here that give credibility to the conclusions that I will make in the final chapter of this report. Starting with performance testing, my two focuses have been on throughput and latency. To start with I have presented a technology independent analysis of performance for a solution based in reconfigurable hardware. This is abstract and provides grounds for many other projects to utilise when performing their preliminary performance analysis of their applications. Using this and the implementations described in previous chapters I have been able to evaluated the performance of the two systems. Given that the systems follow identical specifications, this allows for direct comparison between them without any doubt in the results. I have looked at results under a number of scenarios which all provide correlating evidence. Additionally I have looked at how the new implementation will scale with the increases expected in work load over the next decade.

I have studied the plausibility of using partial run-time reconfiguration within the system which was proposed as a means of improving performance further and increasing the flexibility of the system. Like with the performance analysis, I have provided a technology independent analysis of this functionality which has then been used with my implementation to evaluate the effectiveness of the available technology. Again this mathematical analysis is abstract, allowing it to be applied to a number of different scenarios, technologies and projects. Using these models and known information about current reconfigurable hardware, I have evaluated my implementation, looking at three different scenarios for the three algorithms provided in this project. This has looked at the implementation size of each of the algorithms and how it changes with their computational batch size varied.

To complete the performance comparison, I have looked at the power, cost and space efficiency of both of the implementation. Using the algorithm library created for the partial run-time reconfiguration analysis section, I have studied the power profiles of the algorithms and how it varies with different

parameters. I have also looked at how power usage scales with clock frequency of the algorithms to ensure that the most power efficiency design to performance is used. Using similar techniques of analysis, I have looked at the cost of the implementation compared to the software equivalent and the space that they both consume. Both of these are gaining importance within companies as they look to make their operations as efficient as possible. Without an improvement in performance per dollar, their would be little attractiveness to using such a solution in industry as alternatives would produce better performance for the investment made.

Chapter 9

Conclusion and Further Work

I began this project with the view of evaluating the possibility of using a reconfigurable hardware solution for accelerating an algorithmic trading engine, specifically the trading algorithms. The evaluation was based mainly on performance but also looked at a number of other important characteristics which are fundamental when choosing one platform over another. In this chapter I will review the project and address any possible further work enabled during this process.

9.1 Project Review

At the beginning of this project I outlined the background required to understand the work carried out and presented here. I explored a number of different industry standard algorithm trading styles based on mathematical models that would later go on to formalise my concrete examples of the design patterns and architectures that I would present with my system for evaluation. The reasons on why a reconfigurable hardware platform had been chosen for investigation were outlined with the requirements for an algorithmic trading engine. Additionally I laid out an overview of the currently available reconfigurable hardware technologies and the state of the art in research within the area.

Secondly I presented my implementation of an abstract electronic trading framework which is designed to facilitate the development of any number of electronic trading systems. This would increase compatibility between different systems, aiding their communication. I also laid out the design principles underlying the software parts of this project, the problems that they presented and how they were solved. The framework provided the starting point for the rest of the work presented in this report.

Thirdly I created a bespoke order manager with an algorithmic trading engine extension on top to increase functionality. I discussed how the architecture of the system presented itself to easily being coarsely threaded to improve performance. Additionally I covered in detail the process of separating communication into its constituent components through the application of the acyclic visitor design pattern and the abstractions built on that to provide components to construct a more sophisticated electronic trading system above them.

Fourthly I presented my proposed design pattern for creating trading algorithms which complied with the algorithm interface set out by the algorithmic trading engine. This provides a method of standardising the trading algorithms, affording them important low-level functionality in threshold trading to help improve their price performance while reducing the amount of time required to develop a trading algorithm. This approach provides a powerful solution to the complex task of designing algorithms and would enable less technically savvy individuals to write specifications describing complex behaviour that could be quickly translated into an implementation. Demonstrating the power of this pattern, I presented three algorithms in mathematical models which I have then implemented using these procedures to provide concrete examples.

Fifthly I have carried over these specifications and the idea of creating an abstract design pattern for trading algorithms into reconfigurable hardware. Instead of a design pattern I have presented an architecture which achieves the same criteria as the design pattern. Decreasing development time and complexity of creating new algorithms while affording important base functionality. I have explored the possibility of using partial run-time reconfiguration of the devices to further increase performance

and allow the algorithms to be specialised to current market conditions. All the while not significantly reducing the price performance of the algorithm or the amount of time lost during reconfiguration. I have proposed two methods of implementing the hardware software interface to enable communication between the different systems, comparing and contrasting the advantages and disadvantages of both.

Sixthly I outlined the testing methods that were utilised within this project to ensure the code developed was of the highest quality and some of the additional work that was required to enable the system to be complete. I have presented my solution for the difficult task of creating realistic testing event data for the complex environment of electronic trading. This included generating market data prices for a number of desired symbols which fluctuated and varied over an infinite time span. This provides a useful test harness for all other electronic trading systems and an improvement in many existing testing solutions using static data captured from a production source.

Finally I have evaluated the two implementations, comparing them against each other to determine the benefits that they each hold over the other. This included the performance acceleration provided by the use of reconfigurable hardware. As well as investigating the power consumption, space and cost implications of such a system, it was discovered that such a system does require marginally more power, space and initial cost. However because of the large increases in performance it offers considerably better usage statistics than the amount required by the software solution to equal the hardware solutions performance. This leads to an extremely attractive solutions that definitely warrants further investigations and work to place into production. I also looked at the long term viability of such a system with current estimates proposing that such a system would have a life expectancy of 5 to 6 years without requiring significant updates. This is opposed to the softwares solution which has less than a years life expectancy.

9.2 Project Remarks

In this report I have proposed an implementation of algorithms for trading equity orders in reconfigurable hardware and compared it to an equivalent software implementation. I have evaluated the performance and found it to be superior, increasing maximum throughput of both orders and updates. In addition I have looked at the size of the implementations and how many can be placed on a single device, making the implementation more cost effective, flexible and increasing performance. The hardware is approximately 377 times faster than the equivalent software implementation and I have shown that it is possible to place on average 6 hardware engines on a single FPGA chip to reach this speed up. With more complex algorithms we can expect both figures to decrease, however it is safe to assume that there would remain a similar speed up. Additionally I have shown a reduction in minimum decision making latency from 0.5 microseconds for the software solution to 0.08 microsecond with reconfigurable hardware. Coupled with the much shallower growth curve for the hardware solution, this explains why there is a significant performance acceleration afforded by the reconfigurable hardware. Finally on the performance analysis topic, I have looked at the life expectancy of the implementations. Using current statistics on work load available, I have extrapolated out into the future, predicting the growth over time. Using the performance results I have been able to come to the conclusion that the current software implementation is already reaching the point where it may no longer be able to cope with any spikes in activity. The hardware solution however appears to have another 5 to 6 years conservatively before it will require replacement.

I have also looked at using partial run-time reconfiguration to allow for changes to be made to a device during the trading day, making it more effective and flexible so that it can cope better with varying market conditions. We have shown that the device can maintain a high utilisation percentage making it highly cost effective while loosing none of its ability to implement multiple complex implementations. Based on the column reconfiguration technique, I have shown that t takes 0.00035 seconds to reconfigure a single column. The concrete examples provided consume between 3 and 8 columns each. To show that the loss of time is minimal I have looked at what the current peak market data update rate has been recently and how many messages would have been missed or required buffering during that time. These figures were taken from a large number of stock exchanges concurrently where normally we would only be interested in one or two venues and a selection of stocks on each of those venues. With current expected market data update rates, we can expect to miss or need to buffer about 880 messages per column reconfigured. However as each message only requires 640 bits of storage, in a single 2MB buffer we can hold 25,000 updates. Giving plenty of head room for further expansion. I have presented a solution for reducing the time to produce a set of alternative algorithms that can be deployed by suggesting that

the user builds a library of valid implementations. These can be deployed quickly and simply given the appropriate situation. The time required to compile and test the implementation can be completed before the algorithms are required. I have also discovered that optimisations at compilation vary greatly depending on the size of the implementation. This shows in our results where an increase in parameters does not necessarily result in increased resource usage from a high level perspective. Therefore when generating a library it would be worth using a benchmark to discover which algorithm parameters are the most effective and efficient. This allows for inefficient designs to be removed from the library to prevent their use.

Looking at power, space and cost efficiency gains. I have shown that the reconfigurable hardware solution is 204 times more power efficient in terms of the number of updates processed per joule of energy consumed. Additionally I have shown that running a FPGA chip will only consume 3.7 watts more power when utilised as suggested in this report. This is a negligible considering most servers consume over 100 times this amount alone. I have also shown that the software solution cannot beat the reconfigurable hardware solution on either cost or space efficiency. For it to do so, the server running the software would have to cost less than $6 and occupy less than 0.53mm of height in a standard server rack. Both are impossible targets. The increase in performance per $ is 178 times, but one system hosting a FPGA is likely to cost about double that of just the server for the software solution. It is also likely to take up more than 4 times the amount of space in a server rack. However these are worthy trade-offs given the longevity expected of the system and large performance increase it affords. All of these improvements combined offer an extremely attractive solution to the problem of ever increasing work load and competition in the algorithmic trading space. The greatest problem remaining now seems to be the lack of programmers available for harnessing the power afforded by reconfigurable hardware. I feel that I have shown that this is not the case. A software engineer with a solid background in computing can quickly pick up these new technologies and paradigms, given that I had no previous experience working with reconfigurable hardware before starting this project. Therefore I find no reason why investment institutions shouldn't seriously consider using reconfigurable hardware for acceleration in their next electronic trading platform and systems.

9.3 Further Work

Here I will present possible avenues of further work and research that this project has opened up. Many of them came from the discoveries and thoughts that arose during completing this project, which also supplies a base for much of the work suggested here. This would allow much of the work to be carried out quickly with interesting results, but as always with projects like this, unfortunately there was insufficient time for me to carry out any of these points.

- **Remove market favourability recalculation between order batches on the same market data update.** Currently each order batch is treated independently from every other batch. Although this is not always the case if they share the same ancillary information in the market data update that is being processed. Two solutions present themselves, the software could provide notification that the update was the same as the previous batch. If the trading algorithm stored the previous market favourability calculated then this could be retrieved. The problem with this is that the manager for the algorithm would have to ensure that order batches for the same update were calculated consecutively. A better solution would be to provide the algorithm with the updates independently, it will then calculated the market favourability, storing the last value for each symbol. Then when presented with an order batch, it can always use the current value calculated.

- **Support multiple batches of market data update entries.** This current limitation on calculating the market favourability could be particularly problematic for some users and so should be high priority for correcting. The best solution would being implemented together with the above point, with market data updates being provided separately and calculated independently from when they are needed for order processing. This should allow for greater flexibility of the number of entries that are supplied to the system.

- **Investigation into the performance difference between different number representations.** Currently both software and hardware implementation utilise only 32-bit integer number

representations. This is limiting for accuracy and size efficiency on the hardware. Two investigation scenarios present themselves, an investigation into varying the integer size representation in the hardware implementation may result in a more size effective implementation. The benefits from this are that more algorithms will be able to be placed on a single chip or a smaller, more cost effective chip would be available to use and support the same number of algorithms. This should provide a further increase in performance over the software implementation. Secondly, investigation into using floating-point arithmetic to increase accuracy of calculations. This would be better suited to investigating the price performance of the algorithms to quantify any improvements. However, FPGAs are not optimised for floating-point arithmetic, so further performance analysis would be required to ensure that the hardware implementation still provided an acceleration over the software solution.

- **Implementation of the bidirectional hardware software interface.** Due to time constraints and lack of available suitable devices to place my designs on, implementation of the interface should take a high priority. Previously I have explored two methods of implementation, using either *direct memory access* (DMA) or a network interface. Both have their advantages and disadvantages, current I favour making use of the network interface as it supplies greater flexibility and it is more in keeping with the long-term goals of this project, which are moving the whole algorithmic trading engine into reconfigurable hardware. Although the initial work would be greater in supporting the network communication, it would provide longevity to the system and provides interesting areas of research, such as controlling partial run-time reconfiguration from within the device.

- **Investigation into using a more heterogeneous computing approach for computation.** In this project I have looked at accelerating the performance of trading algorithms with reconfigurable hardware. Instead of looking at these solution individually, all computational units should be combined to provide further performance increases. Further investigation into the use of *general purpose graphic processing units* (GPGPUs) may offer further performance benefits. As all three computational units (including *central processing units* (CPUs) provide difference characteristics, investigation should be carried out to discover if different algorithm implementations are more suited to one of the different computational units. Then the use of that algorithm can be prioritised to that device to receive the maximum speed up. Another scenario is that some algorithms may be more heavily utilised than others, it would be beneficial to make sure that these algorithms are executing on the fastest of all the computational units. Other less used algorithms could then be relegated back to slower computational units as the effect of them running slower would be less felt than heavily utilised algorithms.

- **Greater thread density within the software implementation.** Currently the software implementation is very coarsely threaded, this could be limiting its potential. Therefore it would be worth spending time increasing the level of threading within the implementation and reassessing its performance. The likelihood is that the hardware will still be considerably faster than the software, but the difference should be reduced. This is of obvious advantage if the heterogeneous computing approach described above is chosen as the direction of work and the software algorithms will still have a useful place within the design. It would also allow for fairer comparison.

- **Abstract trading algorithm definition language (ATADL).** With the use of a heterogeneous computing solution, all or at least some of the trading algorithms would need to be implemented for different computational units to make the system most effective. This would create a large amount of development work if it had to be done by hand every time. For this reason, I propose the creation of a technology abstract trading algorithm definition language which would generate the specified algorithm in multiple different implementations for the computational units. This would abstract the process away of designing the algorithms from creating the algorithms, making the process purer. The other advantage is that less technologically savvy individuals or people who lack the experience working with some of the specialised computational units such as the GPGPUs and FPGAs would no longer require knowledge in these areas. The disadvantage is that creating a definition language that could sufficiently specify all of the different functionality that may be desired by an algorithm would be difficult, large and complex. Additionally, new functionality may have to be introduced as it is invented by the business, requiring the language to be extended

frequently. There would be also considerable work in verification of the implementations generated and automated testing of the abstract implementations and the concrete implementations created.

- **Implementation of the whole algorithmic trading engine in reconfigurable hardware.** Rather than taking the approach of using heterogeneous computing solutions, the whole implementation could be moved to reconfigurable computing. This would provide an absolute speed improvement and a very attractive system. Cost may be a prohibitive factor as considerably large FPGA chips would be required which carry a premier. Another disadvantage is that more complex abstract data types are used within the engine than the algorithms. This would require solving first, my suggestion is that each algorithmic trading engine would support a set of orders, once this capacity is full then another engine would be required to accept any more orders. A decision would have to be made though on the ratio of parent to child orders, as a limit on the number of active child orders for each parent order would have to be put in place. The last problem would be designing the storage containers, there would have to be maintenance routines which would ensure that the orders were organised and any free space was discovered so that new orders could be accepted if there was available space. My solution to this would be to maintain an address books of unused space which could be kept in an array, acting as a list. It would consist of an array of the same size as the number of orders supported. Each entry would list a free entry in the main array with non-free entries marked. Then an integer position of the end of the list could be maintained. This would provide a method of accessing the end of the list, to add information onto. Finally, information would be taken from the front of the list (the first element of the array) and then all successive elements would be moved forward one place. The beauty of this solution is that it could be achieved in a single cycle in reconfigurable hardware, as each cell can take the value of the cell behind it, with the last cell taking a flag for in use.

- **Investigation into the placement size of algorithms while varying their parameters.** As we saw in the evaluation of the hardware implementation, the size of the implementation when placed on the device differs with the parameters for the algorithm compilation. This is to be expected, what wasn't was that when increasing some parameters, the size of the algorithm decreased slightly. This has been attributed to the compiler of the algorithm being able to carry out further optimisations on a slightly larger design. Therefore a finer-grained investigation into the parameters of the algorithm and its size could provide information on the optimum algorithms to be held in the placement library for deployment. That we any algorithms which do not provide a performance benefit over another but consume more resources on the device can be removed so that they are never deployed.

9.4 Final Remarks

Having completed this project I would like to review the opening scenario at the start. What difference would have been made to the work of our anonymous trader had he adopted my reconfigurable hardware solution? If we assume that the solution was adopted then the traded would have had greater faith in the technological solution for automating the trading of his smaller orders. This would be because of the increased performance benefits which show that the system could still perform optimally even with a significantly increased work load than expected. This is due to the 377 times increase in total capacity over the previous software based solution.

Additionally he would have greater faith in the price performance of the algorithms executing on the system because of their reduced latency in decision making, down from 0.5 microseconds to less than 0.08. This ensures that our algorithmic trading engine makes its decisions before the competitors system, meaning that our orders reach the attractive liquidity first and we get the price that we wanted. Denying our competitors that price in the process. All this has been achieved by less than double the cost of the initial hardware which was running the software solution and given the better price performance of the algorithm, a cost that should pay for itself multiple times over quickly.

List of Figures

List of Tables

List of Code Blocks

Bibliography

[1] James Ashton and Jenny Davey. Trading at the speed of light. *The Times*, May 2010.

[2] Richard Martin. Wall Street's quest to process data at the speed of light. *Information Week*, Apr 2007.

[3] Automatedtrader.net. FPGA's - Parallel perfection? *Automated Trader Magazine Issue 02*, July 2006.

[4] Ivy Schmerken. Credit Suisse hires Celoxica for low-latency trading and market data. *FinanceTech*, Nov 2009.

[5] Sam Mamudi. Lehman folds with record $613 billion debt. *Market Watch*, Sept 2008.

[6] Edward Krudy. Is the may 6 "flash crash" in u.s. stock markets the new normal? *Reuters*, Jun 2010.

[7] Jim O'Neill, Paulo Leme, Sandra Lawson, and Warren Pearson. Dreaming with BRICs: The path to 2050. *GS Global Exoncomics Website*, Oct 2003.

[8] Stephen Wray, Wayne Luk, and Peter Pietzuch. Exploring algorithmic trading in reconfigurable hardware. In *Application-specific Systems, Architectures and Processors*, July 2010.

[9] Stephen Wray, Wayne Luk, and Peter Pietzuch. Run-time reconfiguration for a reconfigurable algorithmic trading engine. In *Field Programmable Logic and Applications*, Aug 2010.

[10] Wikipedia. http://en.wikipedia.org/wiki/equity_finance. *wikipedia.org*, 2010.

[11] Wikipedia. http://en.wikipedia.org/wiki/stock. *wikipedia.org*, 2010.

[12] Google Finance. Vodafone group plc - google finance. *google.co.uk/finance*, Jan 2010.

[13] Wikipedia. http://en.wikipedia.org/wiki/electronic_trading. *wikipedia.org*, 2010.

[14] NYSE. http://www.nyse.com/. *nyse.com*, 2010.

[15] Council European Parliament. Directive 2004/39/ec. *European Union Law*, 21/04/2004.

[16] London Stock Exchange. Homepage. *http://www.londonstockexchange.com/*, Jun 2010.

[17] Martin Waller. Lse on the attack to safeguard its future. *The Times*, 02/01/2010.

[18] Chi-X. Homepage. *http://www.chi-x.com*, Jun 2010.

[19] Wikipedia. http://en.wikipedia.org/wiki/algorithmic_trading. *wikipedia.org*, 2010.

[20] Tellefsen Consulting Group. Algorithmic trading trends and drivers. *Tellefsen.com*, Jan 2005.

[21] Automatedtrader.net. Scalable participation algorithm. *Automated Trader Magazine Issue 10*, Q3 2008.

[22] Wikipedia. http://en.wikipedia.org/wiki/time_weighted_average_price. *wikipedia.org*, 2010.

[23] Investopedia. http://www.investopedia.com/. *investopedia.com*, 2010.

[24] Wikipedia. http://en.wikipedia.org/wiki/reconfigurable_computing. *wikipedia.org*, 2010.

[25] Gerald Estrin. Organization of computer systems: the fixed plus variable structure computer. In *IRE-AIEE-ACM '60 (Western): Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, pages 33–40, New York, NY, USA, 1960. ACM.

[26] G. Estrin. Reconfigurable computer origins: the ucla fixed-plus-variable (f+v) structure computer. *Annals of the History of Computing, IEEE*, 24(4):3–9, Oct-Dec 2002.

[27] Algotronix. Algotronix history. *algotronix.com*, Jun 1998.

[28] Alpha Data. http://www.alpha-data.com/. *alpha-data.com*, 2010.

[29] Xilinx. Homepage. *http://www.xilinx.com/*, Jun 2010.

[30] Altera. Homepage. *http://www.altera.com/*, Jun 2010.

[31] Wikipedia. http://en.wikipedia.org/wiki/fpga. *wikipedia.org*, 2010.

[32] Yan-Xiang Deng, Chao-Jang Hwang, and Der-Chyuan Lou. Two-stage reconfigurable computing system architecture. In *Systems Engineering, 2005. ICSEng 2005. 18th International Conference on*, pages 389–394, Aug. 2005.

[33] Kuen Hung Tsoi and Wayne Luk. Axel: A heterogeneous cluster with FPGAs and GPUs. In *FPGA'10*, Feb 2010.

[34] O. Mencer, Kuen Hung Tsoi, S. Craimer, T. Todman, Wayne Luk, Ming Yee Wong, and Philip Leong. Cube: A 512-fpga cluster. In *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pages 51–57, April 2009.

[35] Xilinx. *Spartan-3 FPGA Family data sheet.*

[36] Xilinx. *Virtex-5 Family Product Table.*

[37] I. Page. Hardware-software co-synthesis research at oxford. *Oxford University*, 1996.

[38] Zhi Guo, Walid Najjar, and Betul Buyukkurt. Efficient hardware code generation for fpgas. *ACM Trans. Archit. Code Optim.*, 5(1):1–26, 2008.

[39] G.W. Morris and M. Aubury. Design space exploration of the european option benchmark using hyperstreams. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 5–10, Aug. 2007.

[40] G.W. Morris, D.B. Thomas, and W. Luk. FPGA accelerated low-latency market data feed processing. In *High Performance Interconnects, 2009*, pages 83–89, Aug. 2009.

[41] T.K. Lee et al. Compiling policy descriptions into reconfigurable firewall processors. In *Field-Programmable Custom Computing Machines, 2003*, pages 39–48, April 2003.

[42] S. Yusuf et al. Reconfigurable architecture for network flow analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):57–65, Jan. 2008.

[43] A.H.T. Tse, D.B. Thomas, and W. Luk. Accelerating quadrature methods for option valuation. In *Field Programmable Custom Computing Machines, 2009*, pages 29–36, April 2009.

[44] D.B. Thomas and W. Luk. Credit risk modelling using hardware accelerated monte-carlo simulation. In *Field-Programmable Custom Computing Machines, 2008*, pages 229–238, April 2008.

[45] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *DEBS '09: Proceedings of the ACM Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[46] M. Showerman et al. QP: A heterogeneous multi-accelerator cluster. In *10th LCI International Conference on High-Performance Clustered Computing*, Mar 2009.

[47] Tobias Becker, Wayne Luk, and Peter Y. Cheung. Parametric design for reconfigurable software-defined radio. In *ARC '09: Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pages 15–26, 2009.

[48] Thomson Reuters. Homson reuters enterprise platform for real time. *http://thomsonreuters.com/products_services/financial/financial_products/real-time/enterprise_platform_for_real-time*, Jun 2010.

[49] Bloomberg. About bloomberg. *http://about.bloomberg.com/product_data.html*, Jun 2010.

[50] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison Wesley, 1995.

[51] Robert Martin. Acyclic visitor. *http://www.objectmentor.com/resources/articles/acv.pdf*, Jan 1996.

[52] S. Bard and N.I. Rafla. Reducing power consumption in FPGAs by pipelining. In *Circuits and Systems*, 2008.

[53] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[54] Mentor Graphics. *Handel-C Language Reference Manual*.

[55] Market Data Peaks. Homepage. *http://www.marketdatapeaks.com/*, Mar 2010.

[56] Xilinx. *Virtex-5 Family Overview*. Xilinx, 2100 Logic Drive, San Jose, CA, Feb 2009.

[57] Xilinx. *Virtex-5 FPGA Configuration User Guide*. Xilinx, 2100 Logic Drive, San Jose, CA, Aug 2009.

[58] Intel. Your source for information on intel products. *http://ark.intel.com/*, Jun 2010.

[59] G Balazs, S Jarp, and A Nowak. Experience with low-power x86 processors for hep usage. *17th International Conference on Computing in High Energy and Nuclear Physics*, Mar 2009.

[60] Kontron. Kontron brings intel atom record breaking performance per watt into compactpci systems. *http://kr.kontron.com/about-kontron/news-events/*, Sept 2009.

[61] Xilinx. Information and tools for conquering the key challenges of power consumption. *http://www.xilinx.com/products/design_resources/power_central/*, Jun 2010.