Imperial College London

Imperial College London

Department of Mathematics

Low Latency Finance

Author: Jan Jasper Eckstein (CID: 02096261)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2021-2022

Declaration

The work contained in this thesis is my own work unless otherwise stated.

Acknowledgements

I would like to thank my supervisor Paul Bilokon for the time and effort he invested into supporting my thesis by providing feedback and invaluable advice over several months.

Moreover, I want to thank the entire Mathematical Finance department at Imperial College to support us students and to impart us a huge amount of knowledge.

Lastly, I would like to express thanks to my family, for their constant encouragement and for enabling me the opportunity to undertake this M.Sc. program.

Abstract

This thesis investigates low latency strategies for high frequency trading. Starting from the introduction of low latency techniques used in industry today, we move towards statistical arbitrage strategies used in algorithmic trading, where we consider modeled and model-free approaches for choosing suitable pairs and subsequently executing trading signals. We then implement our own pairs trading strategy in C++ and show the impact of adding different micro optimization techniques by rigorous benchmarking. We also analyse the expected profitability gains for high frequency traders by decreasing execution time of their strategies.

Keywords: Low Latency, Algorithmic trading, High frequency trading, Code optimization, Cpp, Pairs Trading, Statistical Arbitrage

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Since the beginning of on-exchange stock trading in the early 17th century on the Amsterdam stock exchange, trading methods and techniques have come a long way. Initially, initial public offerings with only 1000 investors and small volumes were carried out in the marketplace, Over time, trading developed further, and some traders started to buy and sell different stocks while standing on designated busy floors for this purpose [7]. It took until the 1980s before the first computerized types of trading came to the market. It became increasingly famous when the first trading strategies between indices like the SP500 and its future counterpart became profitable. Back then, a program could automatically send an offer to the New York Stock Exchange if a certain threshold between the two markets was crossed. Another milestone was set with the development of Electronic Communication Networks, which allowed trading outside of stock markets and made it publicly available. In the early 2000s trading received another boost in popularity which emerged from a variety of reasons like technological innovations for algorithmic trading, but also the narrowing of spreads by enabling stock quotes to be priced in decimals and other regulatory changes which made it harder/impossible to make easy profit for human traders. All these innovations, along with higher computational capacity in general, paved the way for a new sector in trading: high frequency trading [8].

> *It reminds me of the old story of the two high-frequency traders on safari. Coming out of the jungle into a clearing, they are faced with a hungry lion, staring at them and licking his lips. One of the traders immediately starts taking off his boots and donning a pair of sneakers. "What are you doing?" says the other trader. "You'll never be able to outrun a hungry lion." "I don't need to outrun the lion," says the first trader. "I only need to outrun you."*
> *— HFT Review, April 2010*

In general, high frequency trading (HFT) means the process of buying or selling securities for which success is measured by how quickly the trader acts, and a delay of a millionth of a second will determine profit or loss. High frequency trading is not only bound to the stock market but also in the markets for futures, options and cryptocurrencies. While speedy execution is essential in nearly all fields of trading financial assets, it is not important for everyone trading these assets. Indeed, strategies that aim to buy a stock because the trader thinks that the company will perform well over the next several years are less dependent on speed than strategies that constantly scan all possible markets to spot a profitable trading opportunity before someone else sees and executes it. High frequency traders have their origins in the classic market makers or specialists [9] who primarily made a profit by earning the spread between the prices at which they bought and sold, which before 2001 was at least one-sixteenth of a dollar in the United States [10]. Nowadays, after decimalization and technological advances, traders have to settle for much narrower spreads - like margins of a penny or even less. Naturally, when the profit per trade is reduced, the high frequency trader has to scale up the volume at which she operates to achieve the same results. Signs of the rising popularity in high frequency trading and the increase in the number of firms operating in the market are not hard to find. Figure 1.1 shows the increase by more than a factor of 5 between 2002 and 2021 in daily average trading volume for U.S. equities. While over the years, the number of shares per trade reduced significantly, as shown in figure 1.2 - which is consistent with what you would expect when more and more firms compete against each other to make a profit; they try to seize every small opportunity. The speed required for these strategies is beyond anything a human

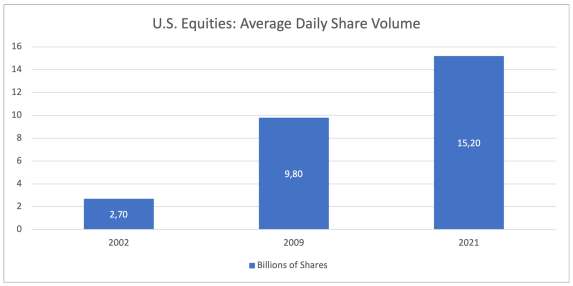


Figure 1.1: Source: Cboe Exchange, Inc.

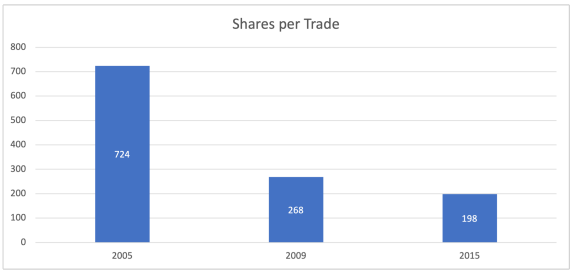


Figure 1.2: Source: Nasdaq, Inc.

could ever match, and the "winner takes it all" nature of trading, as outlined by Carl Cook [11] makes it no surprise that HFT firms try everything to reduce their execution times.

In this thesis, we will focus on the quantitative/computational aspect of high frequency trading strategies which includes latency characteristics for algorithms in C++ on the one hand, and the implementation of a financial algorithm on the other. Due to the black box nature of algorithmic trading, there are few publications and measurements regarding this topic. We will start by presenting a short literature review on pairs trading, low latency techniques and profitability in HFT before we outline in chapter 3 the benchmark procedure we implemented in C++ to measure and report our findings. The following section then provides an in-depth summary and benchmarks for low-latency techniques in C++ reported by professionals from the algorithmic trading industry. Chapter 4 is then used to educate the reader about recent development in statistical arbitrage/-pairs trading research and the different methods which are used in academia and industry at the moment. In Chapters 5 and 6, we then implement one of these strategies, optimize it by using techniques from chapter 4 ,and we conclude with reporting a profitability analysis. We dedicate the last chapter to general concerns around high frequency trading, and we provide incentives for future projects in this field.

Chapter 2

Lieterature review

In this section, we provide a brief overview of selected academic research. We split the review into three parts, one dedicated to statistical arbitrage / high-frequency trading and one to low latency techniques for which rarely any literature is published, but interesting talks from successful practitioners are mentioned as a reference. The third section is about profitability and its relation to speed in high frequency trading.

The origin of pairs trading and statistical arbitrage can be found in [12]. They used the distance approach, which we will explain in depth in chapter 5.1, to find suitable pairs on standardized historical prices. The authors then backtested their strategy on daily U.S. equity data from the past 40 years, dating back from 2002. The empirical approach includes two stages. They first chose a time window of 12 months to detect co-moving stocks (formation period), followed by the actual trading over a six months trading period. Choosing the formation period twice the length of the trading period developed to the academic standard nowadays. Similar to Bollinger bands, their trading signal was generated when the price deviated by more than twice the standard deviation from the mean observed in the formation period. This strategy achieves astonishingly an annualized average yield of around 11 percent. The authors then explain the high returns as compensation for making the markets efficient by enforcing the law of one price principle, which is the foundation for asset pricing theory and second, as a risk-reward for the exposure to market risks.

Another landmark result was published by Do et al.(2010) [13], in which they, on the one hand, replicated the results by Gatev et al. but also detected that the profits made from pairs trading strategies declined over the years, which they explain by increased competition and worsening arbitrage market risks. Their study shows that from 2003 onward, after taking execution prices and other profit influencing factors into account - the pairs trading strategy is barely profitable. Additionally, they correct the numbers presented by Gatev et al. for the time window before 2003 by incorporating trading costs etc. Another interesting founding of their study is that pairs trading performs better during recessions or periods of high volatility.

Noting that Gatev and Do used daily trading data for their strategies, the next step in pairs trading research was conducted by Bowen et al.(2010) [14] by performing a statistical arbitrage strategy on one year of FTSE100 equity intraday data. And their research is one of the first, which also shows that profits are highly correlated with the speed of execution, confirming the need for low latency trading systems. The study shows that a longer execution time completely nullifies profits and that the strategy is related to market and reversal risk factors. This leads to the idea of implementing a stop trading boundary if the prices between the pairs diverge too far.

We will also discuss the co-integration approach for pairs trading in chapter 5.1, which was introduced by Kishore in 2012 [15]. He states that optimal boundaries in statistical arbitrage are non-static, but his study cannot show the profitability of his proposed method.
To conclude the literature review for pairs trading, we mention the recent work by Stübinger and Bredthauer [16]. They achieve profitable results on high frequency U.S. equity data by increasing the standard deviation multiplier needed to enter into trade positions. They also suggest Kishore's result of using dynamic trading signals for better profitability.

For low latency, we would like to briefly mention interesting talks of professional C++ meetings that can be classified as scientifically relevant. Kevin Goldstein(2018) [17] talks about the top concerns for a trading system: first, ultra-low latency; second, consistent performance without outliers and reliability across every market circumstance. He then also introduces in-memory computing (IMC) for trading systems.

Several presentations from Carl Cook [11] in recent years also stressed the importance of setting up a rigorous input/output time measuring system which as he states, is the only way to consistently measure the real performance of a trading system without adding unnecessary overhead or altering the code. He also introduces interesting ideas for tuning the kernel and optimizing the code in a low latency setting.

Nimrod Sapir (2019) [18] and Mateusz Pusz (2017) [2] both align and expand the optimization techniques suggested by Cook in separate talks. Chandler Carruth (2014) [19] shows that besides proven mathematical complexity advantages, the underlying structure of the programming language and how computational processing units are built can alter the actual result and make theoretically slower algorithms faster in practice.

Yaniv Yardi (2018) [20] performed live optimizations on an algorithm to reduce latency by using techniques ranging from presorting data over compile-time calculations of critical values to utilizing low-level system architecture by using bit-wise operations to speed up his code.

We conclude our literature review by referencing interesting work related to profitability in high frequency trading in general and attempts to capture the sensitivity of profits with regards to latency of trading firms.

Elaine Wah(2016) [21] analyzes U.S. exchange and regulatory data to show that many latency related profitability opportunities arise given the fragmentation of trading across several stock exchanges. More opportunities are available in cross-market arbitrage for larger stocks with high liquidity and not on all exchanges. She estimates the total potential profit for the SP500 to be around three billion dollars in 2014. How much of this profit is realized by HFT firms and if dark pools also play a role in latency arbitrage could not be shown.

Weller (2013) [22], on the other hand, focuses on the interaction between market makers. Analyzing commodity futures transaction data, he shows that market makers with different latency times build a chain of intermediaries for trade execution sorted by speed. His model shows that the faster market makers can choose their counterparty and can shift adverse selection risk to slower market makers. This intermediary chain also explains as one of the first articles why even a small speed increase can result in high profitability gains for market makers, even if the actual execution price improves by just a small amount or not at all. It is the relative speed between market makers which matters, and if a slight speed increase moves the firm up in the intermediary chain, more profit is guaranteed. His work also lines out that removing these effects by permitting the arms race for speed might harm the overall market or even lead to market collapse.

A recent paper also discussing latency arbitrage conducted by HFT firms which ultimately leads to the earlier mentioned arms race for speed, is written by Aquilina (2022) [23]. He focuses on potential negative aspects related to it. The author analyses high frequency data of the FTSE100 to show that per stock, about one stock race happens every minute, that they take on average only a few milliseconds each and account for roughly one-fifth of the total trading volume. The paper also shows that only a few fast market makers compete against each other and that being faster than other market makers will result in a higher profit margin. The authors also propose that introducing regulations on these stock races could reduce investors' cost of liquidity by up to 17%.

Focusing more on the speed improvement of single market makers, Ende(2011) [6] measures the impact of latency on trading with DAX30 data from Xetra. The authors show, similar to other articles, that bigger stocks in terms of market capitalization are more impacted by changes in

latency trading than other stocks. Their data shows a significant non-linear relationship between trading latency and the probability of being selected for unfavourable trades. Additionally, they describe an intraday pattern, which shows that the probability of unfavourable price changes in the order book due to high frequency trading is higher at the beginning and end of the trading day while being low in between. Negative price effects resulting from HFT on retail traders or long-term institutional traders are described as neglectable.

As a last notable paper, we would like to mention the work of Baron et al.(2018) [24], who studied latency impact on competition between HFT firms. He again notes that for firms with already low latency, the main profit gained by increasing execution speed results from beating other HFT firms in the latency ranking. The authors mention that these profits are related to the short-lived information channel but also to risk management and that faster HFT firms are able to have better risk management or can trade with less risk in general.

Chapter 3

Benchmarking

The importance for high frequency traders of measuring the latency of their trading systems raises the question of how and between which steps in the trade cycle the measurement should take place. In general, HFT firms want to reduce the time between the first receiving of the market data until the "send-order" message is ready to send. In programming, we have two common approaches to measuring our code. First, we can profile it, which examines what your code is doing, or second, we can benchmark it, which is timing the speed of our system. Both methods have certain pitfalls [11]. For profiling, it is necessary to keep in mind that it is, first of all, useful for catching unexpected things in your program, for example, the amount of time spent in a specific loop or function. But improvements or reducing the amount of these unexpected occurrences does not necessarily mean that your code is now running faster. On the other hand, a common mistake in micro-benchmarking is to apply wildly positive overall performance tests. Most benchmarking tools focus on throughput performance and report an average latency. However, while the identical code is run in production, the outcomes may be dramatically worse. The trouble with benchmarking average latency is that it now no longer effectively monitors the problem of excessive tail latency. For example, the code may have good latency 99% of the time, however, the latency for rare events exceeding the percentile may have latency ten to hundreds of times the average. This may result in drastic outcomes, especially in high frequency trading, where excess events can result in huge losses. Optiver, an international market making firm, therefore, uses an end-to-end measurement setup, similar to figure 3.1, as reported by Carl Cook [11]. Since we are focusing on code snippets/algorithms

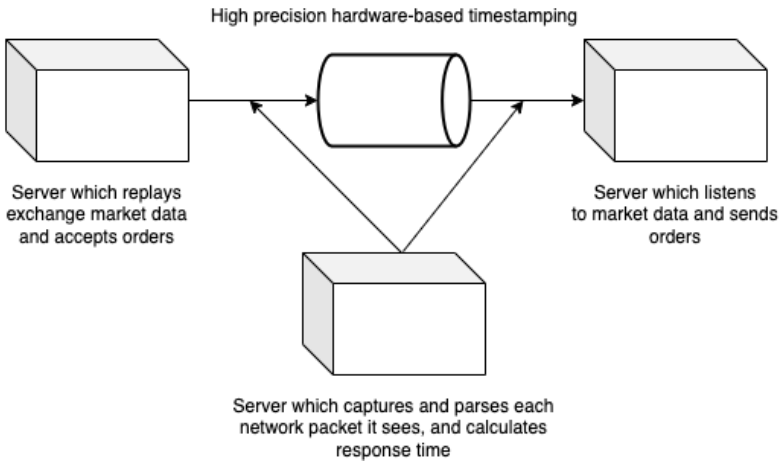


Figure 3.1: End-to-end time measurement in a production-like setup

instead of a whole trading system in this paper, we will use Google Tests/ Google Benchmark for conducting all latency measurements.

Google Test / Benchmark

In this section, we will introduce the Google Benchmark library shortly since we will use it to evaluate the performance numbers presented in this paper. Introduced by Google in 2014, Benchmark

is a C++ library for running microbenchmarks, supporting value- and type-parameterized benchmarks, including multithreading and custom report generation. It represents a lightweight but powerful framework - an example output of a run comparing a recursive implementation against a naive factorial implementation is shown in figure 3.2.

```
2022-06-10T15:53:22+01:00
Running /Users/janeckstein/Documents/Master Thesis/code_folder/release/code_folderlib_benchmarks/code_folderlib_benchmarks_run
Run on (8 X 2700 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB
  L1 Instruction 32 KiB
  L2 Unified 256 KiB (x4)
  L3 Unified 8192 KiB
Load Average: 6.44, 4.03, 3.25
-------------------------------------------------------------
Benchmark                   Time             CPU   Iterations
-------------------------------------------------------------
BM_SimpleRecursion        138 ns          135 ns      4413007
BM_FactNaive              107 ns          104 ns      6791040

Process finished with exit code 0
```

Figure 3.2: Naive factorial compared to recursive factorial function.

Additionally, Google Benchmark provides us with basic statistics for our code too, as shown in figure 3.3.

```
(base) janeckstein@MBP-von-jan code_folderlib_benchmarks % ./code_folderlib_benchmarks_run --benchmark_repetitions=30 --benchmark_report_aggregates_only=true
2022-07-05T14:46:36+01:00
Running ./code_folderlib_benchmarks_run
Run on (8 X 2700 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB
  L1 Instruction 32 KiB
  L2 Unified 256 KiB (x4)
  L3 Unified 8192 KiB
Load Average: 1.54, 2.29, 2.42
-------------------------------------------------------------------
Benchmark                         Time             CPU   Iterations
-------------------------------------------------------------------
BM_SimpleRecursion_mean         107 ns          107 ns           30
BM_SimpleRecursion_median       106 ns          105 ns           30
BM_SimpleRecursion_stddev      5.12 ns         4.39 ns           30
BM_SimpleRecursion_cv          4.77 %          4.11 %            30
BM_FactNaive_mean              85.3 ns         85.3 ns           30
BM_FactNaive_median            85.3 ns         85.3 ns           30
BM_FactNaive_stddev            1.02 ns        1.000 ns           30
BM_FactNaive_cv                1.19 %          1.17 %            30
```

Figure 3.3: Basic metrics in Google Benchmarks

Chapter 4

Low-Latency Techniques

Recent breakthroughs in technology also influenced our financial environment. It is known for a steadily increasing pace of gathering information on the one hand side and reacting to them on the other hand. The sheer efficiency of capital markets nowadays requires traders to respond quicker to market imbalances. While being faster minimizes the volatile nature of financial assets, it is also necessary to execute profitable trades before the competition does it. Thus, the nature of financial markets leads to high frequency trading companies competing in an arms race for speed. This results in major stock exchanges like the New York Stock Exchange (NYSE) offering the opportunity for "co-location" at their premise. This means that traders can place their equipment as close to the stock exchanges data output as possible, which results in reduced latency and network complexity [25]. Another example of striving for speed is the development of microwave connectivity between Chicago and New York and other financial hubs like London and Frankfurt, which results in data being sent faster compared to ordinary fibre cables. Actual number estimates are that the fastest fibre round-trip time from Chicago to New York is around 13milliseconds while companies like McKay Brothers claim to offer round-trips through microwave in up to 8.2milliseconds [26]. To put this into perspective, the theoretical minimum, bounded by the speed of light, is around 7.9 - 8milliseconds. Being closer to exchanges is always advantageous. Therefore, to provide a fair environment for trading companies that buy co-location spots at exchanges, it is regulated that all traders will get the same conditions; for example, MiFID II requires providing every trader with the same cable length between the servers [27]. Thus, it is even more natural to focus optimisation efforts on one's systems and minimise trading strategies' latency. Given the highly confidential nature of high-frequency trading, there is, to our knowledge, no existing exhaustive literature on low-latency techniques for trading systems. Therefore, we will outline in this section different techniques used to reduce trading systems' latency. We will discuss various optimisation opportunities for the general system architecture used by HFT firms and then focus on actual code optimisation. Given its general speed advantage and the provided control over speed, all code optimisations are discussed using the programming language C++20 [19].

4.1 System Architecture

A trading system can be seen as a simple three-step process first, receive market data from an exchange or multiple exchanges. Second, process this market data within your algorithm engine and lastly, send out your processed results (buy/sell/wait) to an order router. This is, of course, immensely simplified, and no regulatory and compliance checks were done as well, but it illustrates the general purpose of the trading system. Before we talk about how the actual code can be improved, we would like to discuss low-latency considerations in the general architecture.

4.1.1 Kernel Bypass

The kernel can be seen as a program that defines the fundamental core of an operating system. It does not interact directly with the user but with other programs and underlying hardware devices such as the CPU, memory and disk drives. It provides essential services for the operating system like memory management, process management, file management and input/output management, which are continuously requested by other parts of the operating system or by applications through a set of program interfaces called *system calls* [28]. In the following, if we mention kernel, we refer

to the Linux kernel. There are several reasons why doing kernel calls is inefficient while operating a high frequency trading system. First, network interface controllers (NIC) are nowadays capable of processing 10Gbps / 10M pps (packets per second). The kernel, on the other hand, can do only

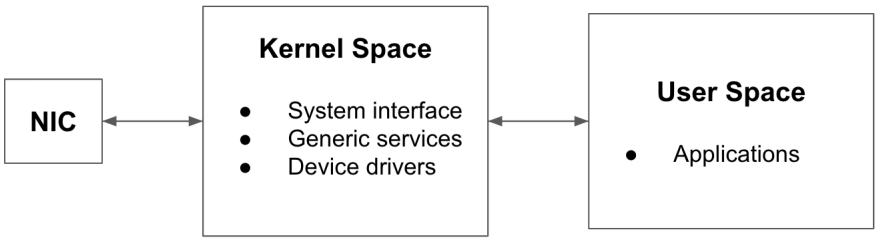


Figure 4.1: A simplified architecture diagram between, network adapter, kernel & user space.

around 1M pps. Thus, to process more packets from the hardware, we have to work around the kernel networking stack. This procedure is called a *kernel bypass*. In figure 4.2 it is checked that even without passing packets to user space, the kernel was only able to process less than 12% of the packets sent by the NIC per second [1]. While in figure 4.3 we illustrate that adding multiple

```
$ sudo iptables -t raw -I PREROUTING -p udp --dport 4321 --dst 192.168.254.1 -j DROP
$ sudo ethtool -X eth2 weight 1
$ watch 'ethtool -S eth2|grep rx'
     rx_packets:       12.2m/s
     rx-0.rx_packets:   1.4m/s
     rx-1.rx_packets:   0/s
     ...
```

Figure 4.2: Number of packets processed by the kernel under perfect conditions [1].

cores does not scale up the processing, but on the contrary, reduces packets processed, precisely only around 480k pps per core. We conclude that including the kernel will not work sufficiently for data processing in our trading system. Another issue with the kernel is that it further slows down

```
$ sudo ethtool -X eth2 weight 1 1 1 1
$ watch 'ethtool -S eth2|grep rx'
     rx_packets:     12.1m/s
     rx-0.rx_packets: 477.8k/s
     rx-1.rx_packets: 447.5k/s
     rx-2.rx_packets: 482.6k/s
     rx-3.rx_packets: 455.9k/s
```

Figure 4.3: Number of packets processed by the kernel when using multiple cores [1].

our system by conducting system calls. While the actual time taken varies for different system calls, a minimum of around 10-25 clock cycles per system call can be assumed [29]. This converts on a 3GHz CPU to 30-75ns per system call, which can additionally be skipped if we bypass the kernel. One public solution is Solarflare's OpenOnload kernel bypass technology, which, tested by RedHat, achieved great latency reduction without further code changes applied [30]. This means that HFT firms typically run all their software in user space, don't perform memory allocation in real-time and have specified network interface cards for their purposes [18].

4.1.2 Context switching, Queuing and Data transfer

Context switching is the process of storing the state of a process or thread so that it can be restored and resume execution at a later point. HFT firms will try to avoid this, queuing and transferring data between different threads as much as possible [18]!

4.2 Code Optimization

4.2.1 Cache Warming

While operating a trading strategy, most of the time, you will not buy or sell anything and wait for further information, which activates one of your trading signals. We call the hotpath the part of our code that executes the buy/sell orders. Given the vast amount of data we have to process every second, the full hotpath is only exercised very infrequently compared to waiting instructions - your cache has most likely been trampled by non-hotpath (slowpath) data and instructions. A simple solution, run a very frequent dummy path through your entire system, keeping both your data cache and instruction cache primed before the program "really" needs it. To illustrate it informally, we can think about it as being similar to an athlete doing stretching exercises to warm up their muscles before they need them for competition. More formal, we want to do cache warming to execute rarely executed but critical code faster. Cache warming means to keep the critical code/data in the cache by executing/accessing it in an artificial way, while the execution must not affect the program's actual state. In the following, we will outline an example of what cache warming in trading systems could look like and which impacts it has on the performance.

We illustrated the main idea in a simple flow chart in figure 4.4 and figure 4.5.

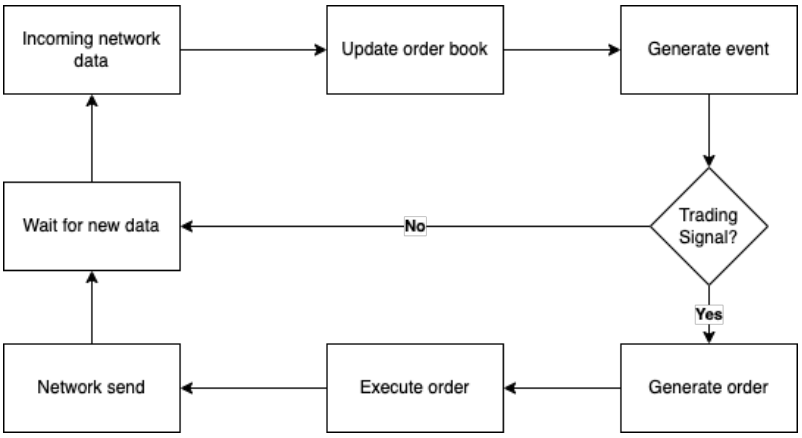


Figure 4.4: Simplified flowchart without cache warming.

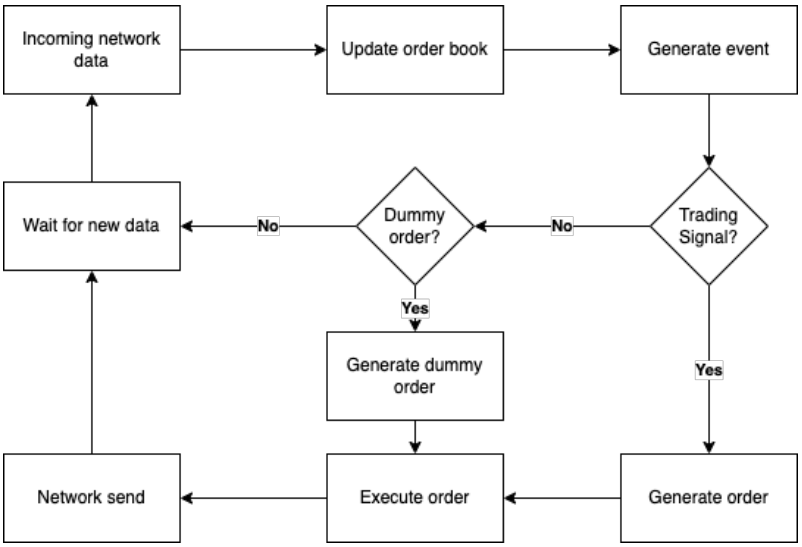


Figure 4.5: Simplified flowchart with cache warming.

We now introduce a simplified pseudo-code example on buying gold ETFs,

```
1  void buy_gold_etf();
2
3  void run() {
4      while (true) {
5          auto etf_price = get_etf_price();
6          if (not should_buy(etf_price)) {
7              update_etf_history(etf_price);
8          }
9          else {
10             buy_gold_etf(); // important code
11         }
12     }
13 }
```

Listing 4.1: Pseudocode on buying gold ETFs

In listing 4.1 line 9 and 10 refer to our hotpath code. As described above we warm the cache by adding a dummy structure for the hotpath.

```
1  void buy_gold_etf(bool only_warm_code);
2
3  void run() {
4      while (true) {
5          auto etf_price = get_etf_price();
6          if (not should_buy(etf_price)) {
7              update_etf_history(etf_price);
8              buy_gold_etf(true); // most calls originate here
9          }
10         else {
11             buy_gold_etf(false); // important code
12         }
13     }
14 }
```

Listing 4.2: Pseudocode with dummy structure for the hotpath

We added a Boolean in listing 4.2 to indicate whether we are warming the critical code or not. Additionally, we added a call to buy ETFs to initiate the warming process. In reality, we would add a frequency threshold that initiates the warming process, for example, 100 times per second, to reduce CPU workload. Let us now dig into the *buy_gold_etf()* code.

```
1  size_t total_bought_amount{};
2
3  void buy_gold_etf(bool only_warm_cache) {
4      const size_t available = get_available_amount();
5      If (available < 10)
6          return;
7      const size_t buy_amount = std::min(100, available);
8      const size_t bought_amount = send_order(buy_amount); // program state
9      total_bought_amount += bought_amount; // program state
10 }
```

Listing 4.3: Pseudo-Buy function for gold ETFs

The problem at hand is how to avoid the Boolean in listing 4.3, modifying the program's state. A naive approach would look as follows.

```
1  size_t total_bought_amount{};
2
3  void buy_gold_etf(bool only_warm_cache) {
4      const size_t available = get_available_amount(only_warm_code);
5      If (available < 10)
6          return;
7      const size_t buy_amount = std::min(100, available);
8      const size_t bought_amount = send_order(buy_amount, only_warm_code);
9      if (not only_warm_code)
```

```
        total_bought_amount += bought_amount;
}
```

Listing 4.4: Pseudocode on buying gold ETFs - naive approach

In listing 4.4 we forwarded the Boolean to all function calls, which may alter the program's state. Furthermore, we added an if-statement to avoid incrementing the counter while warming the cache. Through this if-statement, we introduced a new problem, branch misprediction. The internal branch prediction will assume that we rarely increment the counter, which results in slow execution when speed is needed the most. We can solve this by altering the code.

```
std::array<size_t, 2> total_bought_amount{};

void buy_gold_etf(bool only_warm_cache) {
    const size_t available = get_available_amount(only_warm_code);
    If (available < 10)
        return;
    const size_t buy_amount = std::min(100, available);
    const size_t bought_amount = send_order(buy_amount, only_warm_code);
    total_bought_amount[only_warm_code] += bought_amount;
}
```

Listing 4.5: Pseudocode on buying gold ETFs - optimized approach

Note that we removed the if and that we split the code in listing 4.5 into two parts. We now use the Boolean as an index to indicate which counter we increment and avoid branch misprediction. An additional bonus we get is that we most likely also warm the data since the counters are probably on the same memory page. In figure 4.6 the time difference for accessing different levels of cache are outlined. Therefore, having the majority of needed data in the cache can speed the reference process up by more than 200 times. Furthermore, high frequency trading firms also aim to reduce/eliminate the number of cache misses in their hotpath since a single cache in one trading cycle might not change the mean performance of the program but result in a delay for this specific trade, hence a loss in profit.

L1 cache reference	0.5	ns			
Branch misprediction	5	ns			
L2 cache reference	7	ns			14x L1 cache
Mutex lock/unlock	25	ns			
Main memory reference	100	ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000	ns			
Send 1K bytes over 1 Gbps network	10,000	ns	0.01	ms	
Read 4K randomly from SSD	150,000	ns	0.15	ms	
Read 1 MB sequentially from memory	250,000	ns	0.25	ms	
Round trip within same datacenter	500,000	ns	0.5	ms	
Read 1 MB sequentially from SSD	1,000,000	ns	1	ms	4X memory
Disk seek	10,000,000	ns	10	ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000	ns	20	ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150	ms	

Figure 4.6: Latency comparison numbers [2].

4.2.2 Slowpath Removal

As mentioned in the last section, in a trading system, the hotpath, which generally refers to the code which executes a trade given all checks were performed, is only exercised 0.01% of the time. The other 99.99% the system is idle or doing administrative work [18]. Therefore besides warming up the cache, another technique which will reduce latency is to separate the slowpath from the hotpath code. To achieve the best performance, we have to aim for minimal hotpath code and keep the slowpath away from our critical code. To sketch the main idea in listing 4.6 we look at a classical if-statement which executes our hotpath if everything is ok and does some error handling if not.

```
void trading_function(bool checks_okay) {
    if (checks_okay) {
```

```
        // ...
        execute_hotpath
    }
    else {
        stop_order();
        send_to_client("Order Failed");
        log_Message("Order Failed");
        // ...
    }
}
```

Listing 4.6: Pseudocode with a classic if-statement

Error handling should occur only rarely. Thus all the error handling code in our *trading_function* increases the assembly instruction unnecessary. A better solution, provided in listing 4.7, would split hotpath and slowpath.

```
void trading_function(bool checks_okay) {
    if (checks_okay) {
        // ...
        execute_hotpath
    }
    else {
        Handle_Error();
    }
}
void Handle_Error() {
    // ...
}
```

Listing 4.7: Pseudocode with split hot- and slowpath

We will discuss a hands-on example provided by Mateusz Pusz [2], which improves a part of Google's "FlatBuffers" C++ library by separating hotpath and slowpath. By taking a look at FlatBuffers vector_downward class in listing 4.8,

```
class vector_downward {
    uint8_t *make_space(size_t len) {
        if (len > static_cast<size_t>(cur_  - buf_)){
            auto old_size = size();
            auto largest_align = AlignOf<largest_scalar_t>();
            reserved_ += (std::max)(len, growth_policy(reserved_));
            // Round up to avoid undefined behavior from unaligned loads and ↩
                stores.
            reserved_ = (reserved_ + (largest_align - 1)) & ~(largest_align -1);
            auto new_buf = allocator_.allocate(reserved_);
            auto new_cur = new_buf + reserved_ - old_size;
            memcpy(new_cur, cur_, old_size);
            cur_ = new_cur;
            allocator_.deallocate(buf_);
            buf_ = new_buf;
        }
        cur_ -= len;
        // Beyond this, signed offsets may not have enough range:
        // (FlatBuffers > 2GB not supported).
        assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
        return cur_;
        }
        // ...
};
```

Listing 4.8: Google's FlatBuffer vector_downward class

We notice the big if-statement, which handles memory allocation in the case that the length surpasses the currently allocated memory. In a high-performance trading system, memory allocation is handled extremely carefully, and dynamic allocations are avoided in general. Therefore, this part of the code should rarely, if at all, be executed. This results in a not instruction cache-friendly big assembly code block in our hotpath. The problem can simply be solved by decomposing the class. A possible solution presented in listing 4.9, implements a *reallocate()* function which gets defined separately.

```cpp
class vector_downward {
    uint8_t *make_space(size_t len) {
        if (len > static_cast<size_t>(cur_  - buf_))
            reallocate(len);
        cur_ -= len;
        // Beyond this, signed offsets may not have enough range:
        // (FlatBuffers > 2GB not supported).
        assert(size() < FLATBUFFERS_MAX_BUFFER_SIZE);
        return cur_;
    }

    void reallocate(size_t len) {
        auto old_size = size();
        auto largest_align = AlignOf<largest_scalar_t>();
        reserved_ += (std::max)(len, growth_policy(reserved_));
        // Round up to avoid undefined behavior from unaligned loads and stores.
        reserved_ = (reserved_ + (largest_align - 1)) & ~(largest_align -1);
        auto new_buf = allocator_.allocate(reserved_);
        auto new_cur = new_buf + reserved_ - old_size;
        memcpy(new_cur, cur_, old_size);
        cur_ = new_cur;
        allocator_.deallocate(buf_);
        buf_ = new_buf;
    }
    // ...
};
```

Listing 4.9: Vector_downward class with added reallocate() function

Through this simple step, the compiler can easily inline and optimise the actually used part of the code snippet. EPAM Systems reported a performance boost of around 20% by implementing this optimization [2]. Additionally, we should also try not to inline slowpath functions neither by declaring them inline nor by unwanted compiler behaviour. We can utilise C++'s "noinline" attribute for this.

```cpp
void __attribute__ ((noinline)) slowpath_function(){
    ...
}
```

Listing 4.10: C++'s noinline attribute

Benchmarking a simple bubble sort algorithm on uniformly distributed random data (4.11 & 4.12), we note that especially for small-sized arrays (2 to 8 bytes) moving the slowpath into a separate function results in faster execution.

```cpp
void bubble_sort(auto first, auto last) {
    for (auto i = first; i != last; ++i) {
        for (auto j = first; j < i; ++j) {
            if (*i > 100000) { // this will never get called in our example
                    std::cout << "Error detected" << std::endl;
                    std::cout << "Stopping operations" << std::endl;
                    break;
            }
            if (*i < *j) {
                std::iter_swap(i, j);
            }
        }
    }
}
```

Listing 4.11: Bubble sort implementation

```cpp
void __attribute__ ((noinline)) handleerror() {
    std::cout << "Error detected" << std::endl;
    std::cout << "Stopping operations" << std::endl;
    // ...
}

```

```
7  void bubble_sort(auto first, auto last) {
8      for (auto i = first; i != last; ++i) {
9          for (auto j = first; j < i; ++j) {
10             if (*i > 100000) { // this will never get called in our example
11                 handleerror();
12             }
13             if (*i < *j) {
14                 std::iter_swap(i, j);
15             }
16         }
17     }
18 }
```

Listing 4.12: Bubble sort with separate slowpath

Benchmark results are shown in table 4.1, with lower latency of nearly 20%. We conclude that

	Time	CPU	Iterations
Standard BS 2Byte	12.2 ns	12.2 ns	30
Standard BS 4Byte	21.0 ns	20.9 ns	30
Standard BS 8Byte	52.4 ns	52.4 ns	30
Optimized BS 2Byte	10.1 ns	10.0 ns	30
Optimized BS 4Byte	21.0 ns	21.0 ns	30
Optimized BS 8Byte	47.3 ns	47.2 ns	30

Table 4.1: Mean benchmark results for Bubble sort

while striving for ultra-low-latency, it is necessary to view code as data, in a sense that it is also stored in memory, has to be pre-fetched into the instruction cache, and less code is usually faster.

4.2.3 Branching Minimization

Modern CPUs follow a pipeline architecture, which means that there are multiple sequential instructions being executed simultaneously. But the pipeline can only be fully utilised if it is able to read the next instruction from memory on every clock cycle, which in turn means it needs to know which instruction to read. For completeness, we will give a very high-level overview of how a classic five-stage reduced construction set computer (RISC) pipeline is designed. The first stage is the instruction fetch (IF), in which the instruction is fetched from the dedicated instruction cache. The following stage is called instruction decode (ID) and enables the CPU to determine which instruction has to be performed and how many operands for performing this instruction are needed. The third stage is the execution stage (EX), in which the read and decoded instruction is sent to the computer components as control signals. Finally, the last two stages are memory access (MEM) and writeback (WB), where the first handle the need for data memory and the latter writes the results into the register file [3]. A visualized five stage RISC pipeline can be seen in figure 4.7. If we now introduce a conditional branch in our code, the CPU usually does not

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 4.7: Basic five-stage RISC pipeline [3]

know ahead of time which path will be taken, or even worse, it predicts the wrong path. When this

happens, the CPU has to stall until the decision has been resolved. This means that all preloaded instructions must be unloaded. Also, any possible side effects, like data that has changed due to misprediction, must be reverted. This lowers CPU utilisation and, therefore, performance. In a low latency system, we aim to reduce branching. In this section, we outline three common techniques for branch optimisation. First, the goal is to minimise conditional branches in general, and this can be achieved by utilising flags instead of multiple if-else statements or if-cascades. A simple code example is to avoid the code in listing 4.13,

```
void control_function() {
    //...
    if (check_error_A())
        handle_error_A();
    if (check_error_B())
        handle_error_B();
    if (check_error_C())
        handle_error_C();
    else
        continue_hotpath();
}
```

Listing 4.13: Branched code example

and instead aim for this less branched version in 4.14:

```
void control_function() {
    //...
    uint32_t error_flags;
    ...
    if (error_flags)
        HandleError(error_flags)
    else
        continue_hotpath();
}
```

Listing 4.14: Version with less branches

We can further reduce run-time branches by moving the branching to compile-time. One common approach is to templatize branches in C++, as shown in [11]. Let us introduce a trading example where we calculate the theoretical profit made as a market maker by gaining the difference between the midprice and actual bid/ask price of an underlying asset. We then compare this profit with our expenses to decide if we should post the offer or not. The branched approach would again look like in listing 4.15;

```
enum Class Side { Bid, Ask };

void trading_strategy(Side side) {
    const float trade_profit = Calc_profit(side, midprice, bidprice, askprice);
    Check_profitability(side, trade_profit);
    Send_order(side, bidprice, askprice);
}

float Calc_price(Side side, float midprice, float bidprice, float askprice) {
    return side == Side::Bid ? midprice - bidprice : askprice - midprice;
}
```

Listing 4.15: Code with run-time branching

But as a trading firm, we can assume that we already know if we want to place a bid or an ask offer (or both) at compile time. Therefore we can utilize C++ templates to eliminate all branches by determining the specific values at compile time. In our case, this can be seen in listing 4.16,

```
template<Side T>
void Strategy<T>::trading_strategy() {
    const float trade_profit = Calc_profit(midprice, bidprice, askprice);
    Check_profitability(trade_profit);
    Send_order();
```

```
}
template<>
float Strategy<Side::Bid>::Calc_price(float midprice, float bidprice, float ↩
    askprice) {
    return midprice - bidprice;
}

template<>
float Strategy<Side::Ask>::Calc_price(float midprice, float bidprice, float ↩
    askprice) {
    return askprice - midprice;
}
};
```

Listing 4.16: Code with compile-time branching

Which provides us with complete branch free code at run-time. The last technique we can utilize to reduce branch latency is called short-circuiting. This should be used if branching can not be avoided by one of the two earlier mentioned methods. The main idea behind this is to order the conditions done in a conditional branch by computation expense, as shown in 4.17. Starting with the least expensive checks first, the program is not performing the more expensive checks if the earlier conditions are not fulfilled. This saves computational power and, thus, time. Short-circuiting might be necessary because compilers are unable to reorder statements where they are not sure of what side effects might occur [11].

```
// Rewrite:
if (expensive_check() && inexpensive_check()) {...}
// As:
if (inexpensive_check() && expensive_check()) {...}
```

Listing 4.17: Sorting if-checks by complexity

Benchmarks and results for this technique are discussed in chapter 6.

4.2.4 Memory Allocation

In this section, we talk about static and dynamic memory allocation and the associated costs. Note, that only dynamic memory allocation will call *malloc* and *free*. Let us outline why memory allocations and deallocations are slow. There are two things to consider when evaluating the performance of dynamic memory, namely allocation speed which measures how fast the program can allocate and deallocate memory and access speed, i.e. how fast we can access the memory returned by the system allocator. While the latter depends on the underlying hardware memory subsystems. Therefore, we will focus on the allocation speed aspect. One reason that makes dynamic memory allocation slow is *memory fragmentation*. When allocating memory, the allocator requests one or more large blocks of memory from the operating system. It then performs a search algorithm, depending on the allocator, to find an available block of a given size in this memory. Consequently, as time progresses, it gets more difficult to find blocks of appropriate size. This results in longer search times for the allocator and subsequently in the slower allocation of memory. A typical picture of memory fragmentation can be seen in figure 4.8. For a low-latency trading system, we should aim to not do any dynamic allocations in our fast path. Thus, static allocation ahead of time is a possible solution to save run-time allocation costs. Utilizing C++'s *placement new* function enables us, additionally, to hold onto memory forever, which helps us to keep related data together, avoids long-term memory fragmentation and eliminates deallocations. HFT companies, therefore, will increase their memory space by buying/adding additional memory instead of moving data around; "Just buy more memory; it's cheap." [11]. In listing 4.18 we illustrate a simple example which implements a string class with static memory allocation.

```
template<size_t max_size>
class fast_string {
private:
    std::array< type, max_size +1> chars_;
public:
    ... // classic string implementation
};
```

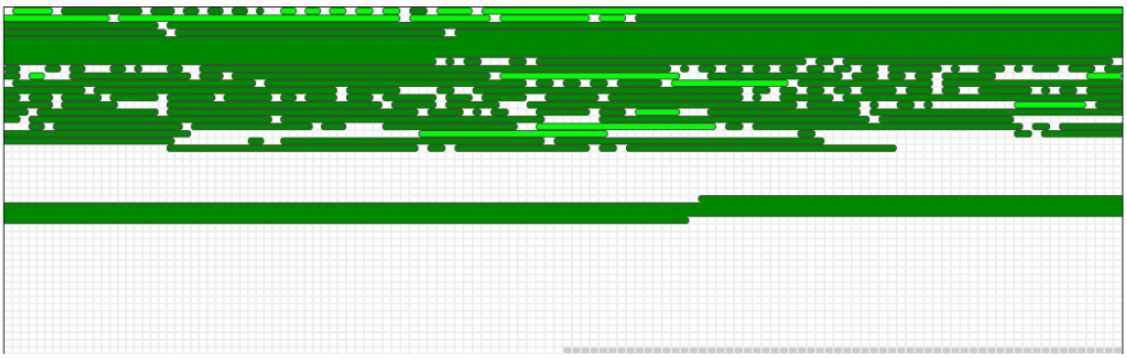


Figure 4.8: Memory fragmentation on a real-life embedded system. Green blocks represent taken, white blocks free memory. [4]

Listing 4.18: String class with static memory allocation

We can then create inplace strings by simple using the template syntax.

```
// inplace examples:
fast_string<4> high;
fast_string<9> frequency;
fast_string<7> trading;
```

Listing 4.19: Example usage of the string class

Since we mentioned placement new earlier, it is useful to keep in mind that for outdated compiler versions, placement new will perform a *nullptr* check on the passed memory address. [11] This adds unnecessary overhead, and striving for low latency, we should consider writing an overloaded version of placement new to avoid this. Besides that, it is necessary to check the inbuilt standard library functions for unintentional allocations. A classic example would be *std::function* which may allocate memory. The reason for the dynamic memory allocation is that *std::function* is generic and has no idea how much data is captured, but that captured data has to exist inside of it and also has to be able to be copied when moving a *std::function* around. Therefore, it needs storage space for your captures. An example is given in 4.20,

```
double i[5];
auto my_lambda = [=] () -> int {
    return (i[0] + i[1] * i[2] + i[3]) ^ i[4];
};
// A dynamic memory allocation of sizeof(my_lambda) + some change is done on ↩
    assignment
std::function f = my_lambda;
```

Listing 4.20: Example of std::function allocating memory

and a workaround is either using an *inplace_function* library [31] or templatizing it as shown in listing 4.21,

```
template <typename T>
void new_lambda (const T &lambda) {
  // No virtual function calls or dynamic memory allocations!
  lambda();
}
new_lambda([] () {(i[0] + i[1] * i[2] + i[3]) ^ i[4];}
```

Listing 4.21: Templatized version to avoid memory allocation

The compiler might also be able to inline the lambda code, which eliminates overhead. An in-depth example of how memory allocation will alter execution speed is provided in the algorithm optimization section 6.

4.2.5 Multi-Threading

Multithreading allows for the exploitation of parallel hardware and the effective use of multiple processor subsystems. Some of its advantages include improved throughput, program structure simplification, and simultaneous use of multiple processors for computation and Input/Output(IO)[32]. But in a high frequency trading setting, we are primarily concerned about latency and not throughput. Consequently, multithreading will, in most cases, not provide any speed benefits but more likely slow our code down. Therefore, as long as the incoming market data can be handled by one process, a single thread solution is generally better than a multithreaded one [11]. We should even further avoid it since introducing it comes with shared Level3 caches and above, a shared memory bus (responsible for connecting the main memory to the memory controller) and a shared network. Additional overhead will be introduced by locking mechanisms on a software and hardware level [33]. If using multiple threads is inevitable, it is important to keep shared data to a minimum since multiple threads writing to the same cache line come with additional costs. Instead of sharing data, it should be passed between threads, and synchronization should be avoided if possible.

4.2.6 Deterministic Code Flow

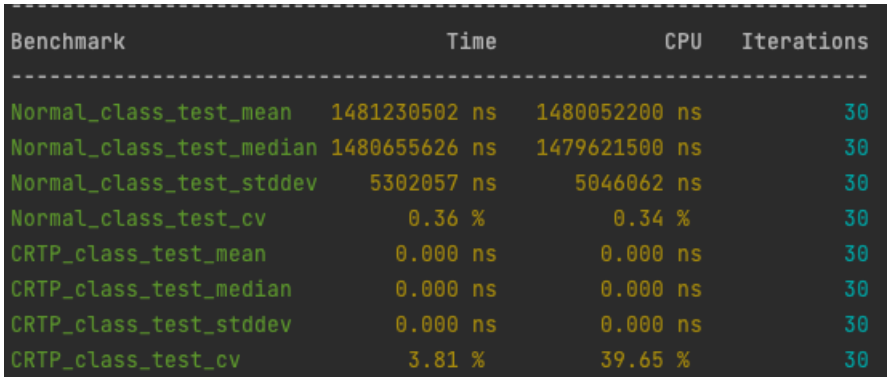
As seen in the branching or memory allocation section, we aim to move as many computations as possible from run time to compile time. In C++, the *constexpr* specifier declares that it is possible to evaluate the value of the function or variable at compile time. Additionally, using *constexpr* in an object declaration implies a *const* declaration, and used in a function or static member variable declaration it implies *inline*. Once declared, this expression can be used in compile-time constant expressions [34]. Since C++20 only a handful of things are not allowed in *constexpr* statements [2]. Therefore, low latency applications can benefit by using it in many ways. A simple example of utilizing compile time expression is given in listing 4.22,

```
consteval int factorial(int n){
int res = n;
while(n > 1)
    res *= --n;
return res
}

std::array<int, factorial(4)> array;

constexpr std::array precalc_values = {
factorial(4);
factorial(11);
factorial(12);
};

static_assert(factorial(4) == 24);
Some_function(factorial(4)); // computed in compile-time
```

Listing 4.22: Compile time expression using *consteval*

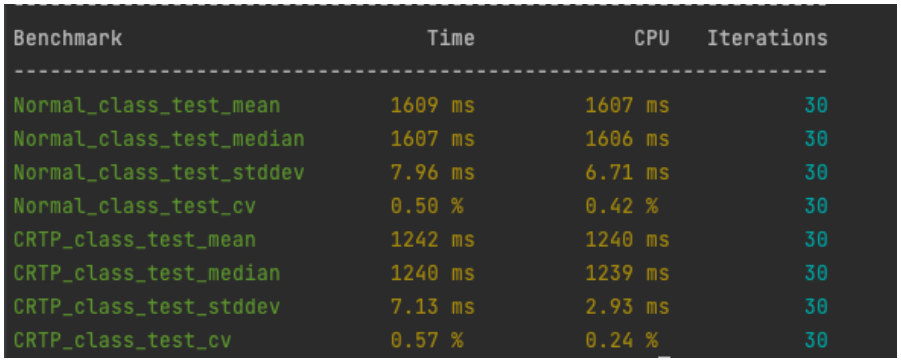
Note that *consteval* will guarantee us that the factorial is computed at compile time which would not be the case if we declared it *constexpr*. As a trade-off general statements like *factorial(n)* would now return an error instead of getting computed in run time (which would be the case for *constexpr*) [2]. We can further move computations to compile time by using a concept called curiously recurring template pattern(CRTP). CRTP is used when polymorph classes are needed, and the actual type of the used classes is already known at compile time. It is then possible to templatize the base class and cast the type of the derived cast into the inheritance line. This reduces run time overhead generated by the v-table of virtual functions which would have otherwise been created. In the following, we present an example for pseudo trading code [18]. The classic approach would look as in listing 4.23,

```
class order {
virtual void send_order() {// Generic implementation...}
};
class buy_order : public order {
virtual void send_order() override
```



Benchmark	Time	CPU	Iterations
Normal_class_test_mean	1481230502 ns	1480052200 ns	30
Normal_class_test_median	1480655626 ns	1479621500 ns	30
Normal_class_test_stddev	5302057 ns	5046062 ns	30
Normal_class_test_cv	0.36 %	0.34 %	30
CRTP_class_test_mean	0.000 ns	0.000 ns	30
CRTP_class_test_median	0.000 ns	0.000 ns	30
CRTP_class_test_stddev	0.000 ns	0.000 ns	30
CRTP_class_test_cv	3.81 %	39.65 %	30

Figure 4.9: Benchmark of a simple loop addition using normal inheritance and CRTP



Benchmark	Time	CPU	Iterations
Normal_class_test_mean	1609 ms	1607 ms	30
Normal_class_test_median	1607 ms	1606 ms	30
Normal_class_test_stddev	7.96 ms	6.71 ms	30
Normal_class_test_cv	0.50 %	0.42 %	30
CRTP_class_test_mean	1242 ms	1240 ms	30
CRTP_class_test_median	1240 ms	1239 ms	30
CRTP_class_test_stddev	7.13 ms	2.93 ms	30
CRTP_class_test_cv	0.57 %	0.24 %	30

Figure 4.10: Benchmark of a loop addition using normal inheritance and CRTP without loop optimization

```
6       {// Specific implementation...}
7   };
8   class sell_order : public order {// No implementation};
```

Listing 4.23: Order class with classic polymorphism

while we prefer the faster CRTP, presented in 4.24.

```
1   template <typename actual_type>
2   class order {
3   void send_order() {static_cast<actual_type*>(this)->actual_place();}
4   void actual_place() { // Generic implementation... }
5   };
6   class buy_order : public order<buy_order>
7   {
8   void actual_place() { // Specific implementation... }
9   };
10  class sell_order : public order<sell_order> {...};
```

Listing 4.24: Order class with CRTP

While measuring performance for a simple example of CRTP, we found another advantageous feature of using CRTP. Using templated classes enables the compiler to heavily optimize your code which it would not be able to do when using normal inheritance structures. In our example, we performed simple loop addition, and by using CRTP, the compiler was able to skip the loop completely - which results in the speed increase shown in figure 4.9 When forcing the compiler to not optimize out the loop, we observe the performance boost resulting from the eliminated v-table, which we show in figure 4.10. Another part of our code which can often be accelerated by using compile-time techniques are functions which can be replaced by lambdas if we know which function we want to call at compile time. We can even use templates to generate general interfaces. Consider the example in 4.25,

```cpp
template <typename T>
void SendOrder(T&& target) {
    // log and send order
    logging(...);
    send(...);
}
```

```cpp
SendOrder([&](auto& order) {
    order.ticker = w;
    order.size = x
    ...
    order.price = z;
});
```

Listing 4.25: Compile-time functions using lambdas and templates

t The compiler will inline this, resulting in no function calls and using templates will again reduce run time overhead [11].

4.2.7 Tailor-made Data Structures

Data structures provided by the C++ standard library are created to fit a variety of general use cases. Therefore, using them in a high-performance environment can sometimes not be optimal. Lower latency can be achieved by constructing own tailor-made data structures. If we consider commonly used C++ data structures like *std::list*, we note that this is a doubly-linked list where each node is separately allocated. Thus, every traversal operation made chases pointers to totally new memory. Resulting in a cache miss for most steps. This is costly and, therefore, we should only use *std::list* when we have a use-case of rarely traversing the list but frequently updating it. Best practice in high frequency trading is preferring *std::vector* over other standard library data structures [19]. By benchmarking it against *std::list* and *std::deque* we note that in theory, a list should be used when random insert and remove operations will be performed, which has a complexity of O(1) compared to O(n) for a vector, or a deque. For linear search, the general complexity should be O(n) for all data structures. The size of the data structures matters most when random insert or random replace operations are performed. For vector and deque, we then have to move every following element, which results in copying those elements. In practice, however, we will notice a more significant difference due to the underlying hardware or, to be precise, the underlying memory caches. A vector is stored contiguous in C++, while *std::list* is not. Thus, besides knowing the time complexity for different data structures, it is important to know their interaction with hardware to build low-latency systems [19]. For operations adding elements to the end, a preallocated vector (green line) clearly outperforms the other data structures, as can be seen in figure 4.11. The same pattern holds for linear search as seen in figure 4.12. More interestingly is

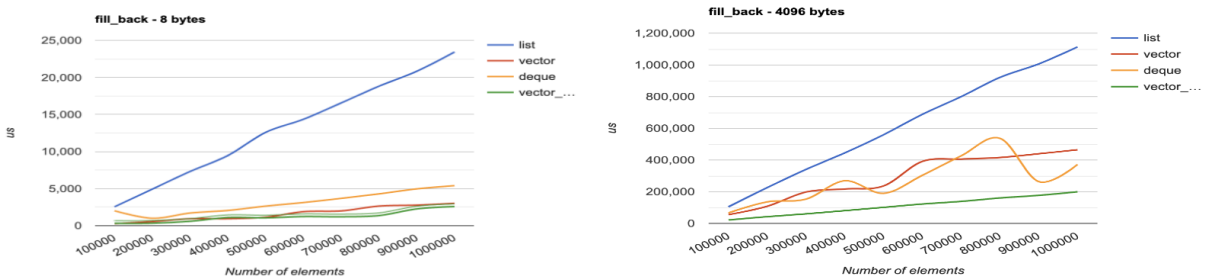


Figure 4.11: Speed comparison for adding elements(of small and big size) to the back of different data structures [5].

even that for small data sizes *std::vector* is still the best option when performing random remove operations, despite its theoretical worse time complexity as can be seen in figure 4.13. The test was performed with an Intel Core i7 Q 820 @ 1.73GHz. We conclude that knowing the different

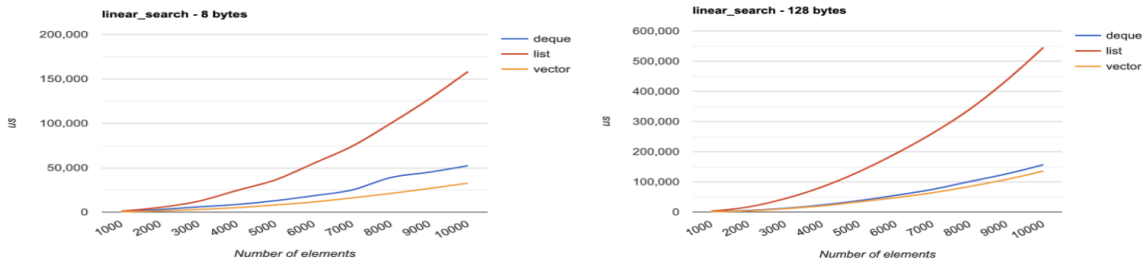


Figure 4.12: Speed comparison for linear search(of small and big size) in different data structures [5].

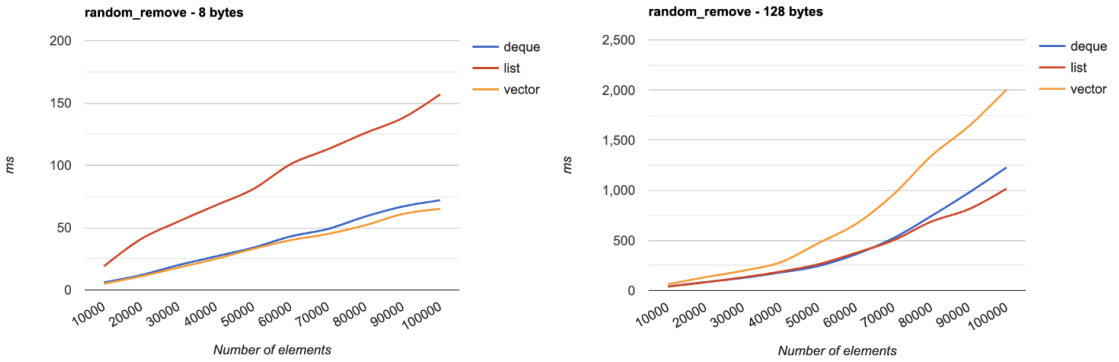


Figure 4.13: Speed comparison for random remove operations(of small and big size) in different data structures [5].

time complexities might be a good indicator to start with, but while striving for low latency, we should always measure the performance of different data structures to guarantee the performance. Additionally, there is no one fits all data structure. Thus, we have to consider building our own data structures for specific applications.

4.2.8 Type Engineering

In this section, we want to emphasize that knowing your types and using different types might increase the performance of your code. We introduce two simple examples to highlight this fact. First, we would like to highlight the difference between signed and unsigned integer indexing. In the standard library, loops are commonly indexed by *size_t* which is an unsigned integer [35]. Given a simple loop using signed integer for indexing as in 4.26,

```
int foo(int i)
{
int k = 0;
for(int j = i; j < i + 10; ++j)
    ++k;
return k;
}
```

Listing 4.26: Standard C++ loop with signed integer counter

results in the assembly code in listing 4.27 (x84-64 gcc 12.1 -O2 -std=c++20).

```
foo(int):
        mov     eax, 10
        ret
```

Listing 4.27: Assembly for the standard loop

If we now write the same loop, but using an unsigned integer for indexing as in 4.28,

```
int foo(unsigned i)
{
int k = 0;
for(unsigned j = i; j < i + 10; ++j)
    ++k;
return k;
}
```

Listing 4.28: C++ loop with unsigned integer counter

we receive the different assembly code presented in 4.29:

```
foo(unsigned int):
        cmp     edi, -11
        ja      .L3
        lea     eax, [rdi+1]
        add     edi, 10
        cmp     edi, eax
        sbb     eax, eax
        and     eax, -9
        add     eax, 10
        ret
.L3:
        xor     eax, eax
        ret
```

Listing 4.29: Assembly for the loop with an unsigned integer counter

This is explained by the fact that in C++, integer overflow is defined for unsigned integers; thus, the program checks if an integer overflow occurs or not. For signed integers, this is not the case, and it might be possible to get integer overflow, which, therefore, the user has to make sure that it will not occur in contrast to when using an unsigned integer. Taking this into consideration, doing integer arithmetic on signed integers is, in general, faster (less assembly) than on unsigned integers [2].
In general floating point calculations, no matter if we are using single or double precision, take the same time. But outlined in Agner [36] there is a penalty in 64-bit operating systems for mixing them up. This is especially important in C++ since floating point literals will be converted as double by default and not as a float.

```
float a, b;
a = b * 1.2; // here 1.2 is a constant double
```

Listing 4.30: Floating point literals

We can therefore speed the program up by choosing the desired datatype for the constant. An example is given in 4.31,

```
float a, b;
a = b * 1.2f; // everything is float
// ---------- (or) ----------
double a, b;
a = b * 1.2; // everything is double
```

Listing 4.31: Performance increase by using doubles

4.2.9 Other Miscellaneous Considerations

One micro-optimization we should consider when dealing with C++ code is, furthermore, to try to **fill the cache line**. As we already mentioned earlier, our cache is the fastest memory we have. We can further accelerate things if we reduce the number of cache lines we have to read in for our

data. Let us assume that for our hardware, the size of one cache line is 8 bytes. Then we can construct the same simple object either like in listing 4.32,

```
struct NotAligned {
    char c; // size=1, alignment=1, padding=7
    double d; // size=8, alignment=8, padding=0
    short s; // size=2, alignment=2, padding=2
    int i; // size=4, alignment=4, padding=0
    static double dd;
};
```

Listing 4.32: Struct without considering cache line alignment

Where we did not think about possible alignment effects of our data, and the seven free bytes of the first cache line had to be padded unnecessarily. A more efficient way of structuring this object would be to reorder its members as in listing 4.33,

```
struct A {
    char c; // size=1, alignment=1, padding=1
    short s; // size=2, alignment=2, padding=0
    int i; // size=4, alignment=4, padding=0
    double d; // size=8, alignment=8, padding=0
    static double dd;
};
```

Listing 4.33: Struct with optimized cache line alignment

This results in reducing the number of cache lines which we have to read by one [2]. This can easily add up to big data inefficiencies once we use large vectors of such unaligned objects.

A similar observation can be made when accessing member data of objects through functions. We should aim to construct our objects **data oriented** in a way that we initialize memory which is often used together close to each other. This way, we can often increase the speed of member functions. We illustrate the idea with a simple example. Consider a simple company stock object which also stores some information about the company and an object manager class, presented in 4.34, which simplifies the interface for accessing the price of the stock and checks for profitability.

```
struct prices { double bid, ask; };
void display(const coordinates& coord);
void profitable(int threshold);
constexpr int OBJECT_COUNT = 10000;
class objectMgr;
void process(const objectMgr& mgr)
{
const auto size = mgr.size();
for(auto i = 0UL; i < size; ++i) { display(mgr.index(i)); }
for(auto i = 0UL; i < size; ++i) { profitable(mgr.threshold(i)); }
}
```

```
class objectMgr {
private:
    struct object {
    prices price;
    std::string exchange_name;
    std::string ticker_name;
    double threshold;
    std::array<char, 100> otherData;
    };
    std::vector<object> data_;
public:
    explicit objectMgr(std::size_t size) : data_{size} {}
    std::size_t size() const { return data_.size(); }
    const prices& index(std::size_t idx) const { return data_[idx].price; }
    int threshold(std::size_t idx) const { return data_[idx].threshold; }
};
```

Listing 4.34: Common implementation of a Stock object

Note that the function *process* will always access the price and threshold together, but the data is not initialized close to each other in the *objectMgr* class. Thus, we have to deal with separated data which leads to an increase in cache misses while accessing the data. A more efficient implementation would look as in 4.35.

```
class objectMgr {
private:
    std::vector<prices> price_;
    std::vector<double> threshold_;

    struct otherData {
        struct compInfo { std::string exchange_name, ticker_name; };
        compInfo info;
        std::array<char, 100> data;
    };
    std::vector<otherData> coldData_;

public:
    explicit objectMgr(std::size_t size) :
        price_{size}, threshold_{size}, coldData_{size} {}
    std::size_t size() const { return price_.size(); }
    const prices& index(std::size_t idx) const { return price_[idx]; }
    int threshold(std::size_t idx) const { return threshold_[idx]; }
};
```

Listing 4.35: Stock object implemented in a data oriented way

This data-oriented design optimizes cache coherency but results in object decomposition. This comes with the trade-off of increasing the maintenance complexity of such objects [2].

Chapter 5

Statistical Arbitrage

One objective of this paper is to show how the speed of a statistical arbitrage algorithm can be effectively improved. We want to introduce the reader to current advances in statistical arbitrage theory and outline different methods discussed in academia and practice. Note that whenever we refer to statistical arbitrage, we mean pairs trading, which is outlined below.

Let us now explain the idea behind pairs trading. A trader who wants to execute a pairs trading strategy wants, in general, to exploit relative mispricing between two assets. As a first step, she has to identify a suitable pair. The usual approach is to find two highly correlated normalized price processes. In general, it is not necessary to have other similarities between the assets; it makes sense to choose only securities within a specific industry, sector or index. Once identified, she tracks the price of the two assets, and once one of the two normalized price series separates far enough from the other, a trading signal is generated. Here, the one moved first is the lead asset, while the other is the lag asset. The trader now sees mispricing in the assets and expects the price processes to converge back again. She can then exploit this expected behaviour by either opening a short position in the lead asset (if its price is relatively higher than the lag asset) and a long position in the lagging asset or a long position in the lead asset (if its price is relatively lower than the lag asset) and short the lagging asset. Note that we do not have an opinion at any time of the actual value of our asset prices, it might be wrong, but the trader is only interested in whether the prices are relatively the same or not. The potential mispricing between the assets is captured by the difference in their normalized price series, and this spread can be seen as the profit potential for a trading pair. Another feature of this strategy is that we go long and short in a positively correlated asset pair which should minimize our exposure to general market movements and reduce our market risk.

Note that this strategy consists of matching asset pairs with similar normalized historical prices into buckets and then trading these assets whenever the difference between their prices diverges over a specified threshold. Subsequently, it is natural for us to split the discussion into two processes: one that discovers related asset pairs and one that focuses on trade execution. We will introduce different strategies for choosing suitable pairs and then focus on the actual trading part.

5.1 Choosing Pairs

Our literature review shows that more research has been conducted on finding asset pairs with correlated moves than on the actual trade execution. This research has resulted in a variety of approaches to building matching pairs. In this section, we will discuss three overarching strategies: the distance approach and the co-integration approach are the best-studied ones in the literature.

5.1.1 Distance Approach

The most simple of the three strategies and, therefore, easy to implement and empirical studies with large data sets can be conducted. To identify correlated asset pairs, the euclidean distance is used. The origin dates back to the paper of Gatev et al. [12] in which a study on a vast amount of liquid U.S. stocks between 1962 and 2002 is conducted. The co-movement is measured by the euclidean distance between their normalized price series in the first step for two given assets. Let $(P_t)_t$ denote the price series for any given asset and $(NP_t)_t$ the respective normalized price series.

Then we define,

$$NP_t = \prod_{t=1}^{T} (1 + r_t), \quad \text{where} \quad r_t = \frac{P_t - P_{t-1}}{P_{t-1}},$$

$$\text{Dist(A,B)} \ = \frac{1}{T} \sum_{t=1}^{T} \left(NP_t^{\text{Asset } A} - NP_t^{\text{Asset B}} \right)^2,$$

for two assets A and B. Assuming that we have n assets selected, we compute the distance for all pairs, i.e. $n(n-1)/2$ pairs. We can then sort the pairs with regard to their distance, and the ones with a smaller distance are favoured. Note that perfect pairs, in the sense that the distance between the assets is zero (Dist(A,B) = 0), have a spread of zero and thus produce no profit opportunity. This contradicts the fact that we expect the highest ranked pair to generate the highest profit. Additionally, note that the sample variance for the spread is given by,

$$\text{SV(A,B)} = \frac{1}{T} \sum_{t=1}^{T} \left(NP_t^{\text{Asset } A} - NP_t^{\text{Asset B}} \right)^2 - \left(\frac{1}{T} \sum_{t=1}^{T} \left(NP_t^{\text{Asset } A} - NP_t^{\text{Asset B}} \right) \right)^2$$

$$\Leftrightarrow Dist(A, B) = \text{SV(A,B)} + \left(\frac{1}{T} \sum_{t=1}^{T} \left(NP_t^{\text{Asset } A} - NP_t^{\text{Asset B}} \right) \right)^2.$$

Thus, minimizing Dist(A,B) is equivalent to reducing SV(A.B), and the squared spread mean [37]. The first term increases in relation to the deviation from the mean, while the second term increases the more the same mean drifts away from zero. Gatev et al.[12] outlined that it is difficult to show which term is more important in the minimization problem. Still, their empirical results showed that low spread volatility results in a low distance between assets. This concludes that the distance method favours pairs with limited profit opportunities.

5.1.2 Co-Integration Approach

This approach was first discovered by Vidyamurthy (2004) and is the second most popular approach after the distance one[38]. We must introduce the notion of (weekly-)stationary processes to dive deeper into the co-integration approach. A time series $(x_t)_t$ is weakly stationary if the first two moments do not depend on time t and the auto-covariance between different time stamps only depends on the passed time between the stamps. This is denoted by,

$$\begin{aligned} \mathbb{E}[x_t] &= \mu, \\ \text{Var}(x_t) &= \gamma_0, \\ \text{Cov}(x_t, x_{t-s}) &= \gamma_s, \end{aligned}$$

where μ denotes the mean and γ_0 the variance of the process; now note that if we have a stationary time series, we can make profits by buying(selling) an asset when it diverges from the mean μ since it is mean reverting. Unfortunately, asset prices alone are not generally mean reverting, which intuitively makes sense because, for example, a firm's stock price is not fixed at a given price level. Furthermore, let's think about the random walk as a model of the asset price. We get the property of infinite variance [39], thus the expected time of crossing a certain threshold again is infinite too. We conclude that it is impossible to forecast profit opportunities under such a model. The primary approach for non-stationary strategies, as outlined in Franses(1991) [40], is to use differenced time series to get the desired properties. In finance, this is often obtained by using the logarithmic series,

$$z_t = \log(P_t) - \log(P_{t-1}).$$

While studying multivariate time series Engle and Granger (1987) [13] observed that even if two time series, which in our case are represented by two price processes of different assets, are non-stationary, in some instances, a specific linear combination of them is stationary. They named this property *cointegration*. Mathematically it is described by having two (non-stationary) time series $(x_t)_t$ and $(y_t)_t$. If for any γ, $(y_t - \gamma x_t)_t$ is stationary then we call this series cointegrated. Furthermore, Engle and Granger describe cointegration through error correction. Thus, if $(y_t - \gamma x_t)_t$ is stationary, i.e. has a long-term mean, any deviation from this mean has to be corrected

at some point in time. Therefore, $(x_t)_t$ or $(y_t)_t$ have to adjust accordingly. This is represented by the error correction statement given by,[13]

$$y_t - y_{t-1} = \alpha_y \left(y_{t-1} - \gamma x_{t-1}\right) + \varepsilon_{y_t},$$
$$x_t - x_{t-1} = \alpha_x \left(y_{t-1} - \gamma x_{t-1}\right) + \varepsilon_{x_t},$$

where $(\epsilon_{x_t})_t$ and $(\epsilon_{y_t})_t$ are two white noise processes. In our case, the left-hand side of the equation can be interpreted as the return of the assets A and B, and $(y_{t-1} - \gamma x_{t-1})$ denotes the long-term mean μ. Vidyamurthy [38] uses the representation of nonstationary logarithmic price series for two assets, A and B, to get the error correction form,

$$\log\left(P_t^A\right) - \log\left(P_{t-1}^A\right) = \alpha_A \log\left(P_{t-1}^A\right) - \gamma \log\left(P_{t-1}^B\right) + \varepsilon_A,$$
$$\log\left(P_t^B\right) - \log\left(P_{t-1}^B\right) = \alpha_B \log\left(P_{t-1}^A\right) - \gamma \log\left(P_{t-1}^B\right) + \varepsilon_B,$$

and introduces a modified version of the Engle-Grange test to see if pairs are cointegrated or not. By regressing the logarithmic price of both assets against each other, which yields,

$$\log\left(P_t^A\right) - \gamma \log\left(P_t^B\right) = \mu + \varepsilon_t, \tag{5.1.1}$$

where ε denotes the price ratio between A and B. In a second step, the residual series of this approach is tested for being stationary by measuring the number of mean crossings. If we perform this approach to all pairs in our bucket, we can rank them similar to the distance approach, descending with regards to the number of equilibrium crossings [38].

5.2 Modeling the Underlying

In this section, we will quickly outline one popular approach which aims to model the price series of different assets in the light of pair trading. It receives criticism for not solving the general problem of selecting security pairs firsthand, as mentioned by Krauss [37].

5.2.1 Time-Series Approach

We will outline the most famous work by Elliott et al.(2005) [41] who proposes a mean reverting Gaussian Markov chain model with Gaussian noise for the spread between a long position and a short position in two assets. The model for this spread is constructed as follows.
Let x_k be our state variable. We assume that it follows a mean reverting process,

$$x_{k+1} - x_k = (a - bx_k)\,\tau + \sigma\sqrt{\tau}\varepsilon_{k+1},$$

for $\alpha, \sigma > 0$, $b \geq 0$, and as usual ϵ_k are standard normal identically independent distributed random variables for all k. In continuous tine this is just a Ornstein-Uhlenbeck (OU) process:

$$dx_t = b(\frac{a}{b} - x_t)\mathrm{d}t + \sigma \mathrm{d}W_t,$$

where $\frac{a}{b} =: \mu$ is the long term mean, $b =: \rho$ the speed of mean reversion and for some probability space W_t a standard Brownian motion. Eliott then models the spread by

$$y_k = x_k + S\omega_k, \quad S > 0, \tag{5.2.1}$$

with ω_k being again i.i.d standard normal.
Note that, indeed, we only considered one price time series. This means that the model only works for assets which follow the same price process, which is rarely observed in reality, as criticised in Do et al. [42]. One practical application could be dual-listed companies or cross-listings. Additional criticism can be made to the assumption of an underlying OU-process which, in reality, clashes with heavy-tailed, not normal distributed financial data. For the sake of completeness, we want to mention that Avellanda et al. [43] successfully tested this approach for other assets.

5.3 Trade Execution

After constructing asset pairs or modelling the underlying price process as shown in above sections the next objective for the arbitrageur is to execute trades to gain as much profit as possible. In pairs trading, this naturally leads to the problem of when to enter a long and short position in the asset pair, or to be more specific, how far the price has to diverge from its equilibrium state before the trader starts opening the spread position. Again we have to make a trade-off between simplicity and acting analytical optimal.
We investigate the trade signals produced by the **distance** method. Following the same notation introduced in 5.1, the signals get activated once the spread crosses its mean plus a multiple of its standard deviation. Thus, let $NP_t^{\mathrm{AB}} = NP_t^{\mathrm{A}} - NP_t^{\mathrm{B}}$ and we get,

$$NP_t^{AB} < \mu^{AB} - k \times \sigma^{AB} \rightarrow \text{Buy asset } A, \text{ short asset } B,$$
$$NP_t^{AB} > \mu^{AB} + k \times \sigma^{AB} \rightarrow \text{Short asset } A, \text{ buy asset } B,$$

Where the parameter k can be seen as a risk appetite factor of the trader, i.e. lower values for k result in many trades and higher risk and vice versa. Most common in literature [12] is a value of $k = 2$. We discuss the distance approach by outlining some advantages and disadvantages. Pro arguments are that it is model-free and, therefore, not subject to misspecifications or misestimations. While on the other hand, the cost of not having an underlying model comes with the inability to perform estimates regarding the holding duration or time until mean reversion.

For the **cointegration** approach in general, the same idea for generating trading signals as for the distance method is used. To present further research, we want to briefly discuss an approach to finding a minimum profit condition, discovered by Lin et al. (2006) [44]. Let's assume without loss of generality that the cointegration coefficient $\gamma < 0$ and that we short asset A and go long in asset B. Thus, the relative price of B is lower than that of A and holding a number n of shares in asset B results in a short position of $\frac{n}{|\gamma|}$ in A. Furthermore, t denotes the opening timestamp and T the closing timestamp of our trade. Remember from 5.1.1 that,

$$\log\left(P_t^A\right) - \gamma \log\left(P_t^B\right) = \mu + \varepsilon_t,$$
$$\Leftrightarrow \log\left(P_t^B\right) = -\frac{1}{\gamma}(\mu + \varepsilon_t - \log\left(P_t^A\right)).$$

Now calculating the total profit per trade $P_{A,B}(t,T)$ we get, (we use logarithmic prices to keep consistency in notation. Dropping the logarithm wouldn't change the result.)

$$\begin{aligned} P_{A,B}(t,T) &= n(\log\left(P_T^B\right) - \log\left(P_t^B\right)) - \frac{n}{|\gamma|}(\log\left(P_T^A\right) - \log\left(P_t^A\right)) \\ &= -\frac{n}{|\gamma|}(\varepsilon_T - \log\left(P_T^A\right) - \varepsilon_t + \log\left(P_t^A\right)) + \frac{n}{|\gamma|}(\log\left(P_t^A\right) - \log\left(P_T^A\right)) \\ &= \frac{n(\varepsilon_T - \varepsilon_t)}{|\gamma|} > K, \end{aligned}$$

if an arbitrageur sets K as a lower bound for the profit per trade. This enables us to calculate the number of shares n as seen in [44] dependent on ε_t and ε_T. Lin's concept has some weaknesses since it only considers the profit of one trade. In contrast, the nature of a pairs trading strategies generates a high amount of trades over a short time frame; thus, a proper strategy should consider the interdependence between several trades. Puspaningrum (2010) [45] develops different concepts considering this concern, which we will, for brevity, not further discuss in this paper.

Our last section is about trade execution under **time series** model specification. Remember that we modeled the spread in 5.2.1 by $y_k = x_k + S\omega_k, \mu = \frac{a}{b}$ and ρ is the speed of mean reversion. Following Elliott's approach [41], we enter into the trade once

$$y_k \geq \mu + c(\frac{\sigma}{\sqrt{2\rho}}), \quad \text{or}$$
$$y_k \leq \mu - c(\frac{\sigma}{\sqrt{2\rho}}).$$

Where c again can be chosen by the trader depending on how aggressive she wants to trade. The position is then unwound once again $y_k = \mu$ at time T. The advantage we get from the model

assumption is that Elliott provided a formula for T depending on the parameter c, which, thus, enables us to gain more information about future trading behaviour like the expected holding time. Additionally, using Kalman filtering and the expectation maximization (EM) algorithm gives a method to fully estimate all parameters in the model [41].

5.4 Other Approaches

To be exhaustive, we want to briefly mention other techniques related to pairs trading that have gained popularity in recent literature. Huck (2010) [46] developed a machine learning approach that performs a forecast of asset returns on a set of given assets in a first step. As we advance in time, it then ranks the assets from overvalued to undervalued, given the divergence from the estimations of step 1. The actual trading strategy then suggests shorting the top listed assets and buying the ones at the bottom. Note that no long-term mean or equilibrium state is assumed in this model.
Other approaches use copula functions to generate trade signals after finding the pairs with either the distance or cointegration approach as studied in Stübinger (2015) [16] and Stander et al. (2013) [47]. Lastly, Avellanda and Lee (2010) [43] introduced a principal component approach for pairs trading.

Chapter 6

Algorithm Optimization

In chapter 5.1, we discussed different statistical arbitrage/pairs trading strategies. These strategies require special needs in a High-Frequency Trading(HFT) setting. Typically HFT algorithms operate at a microsecond time scale and thus face two challenges. First, they must be able to receive a large amount of data every microsecond. Additionally, they have to react to this data without any latency since the profitability signal they observe decays very quickly. As outlined in chapter 5.1, several strategies exist to find correlated trading pairs for statistical arbitrage algorithms. Since this project aims to show how low latency techniques can be used to accelerate trading strategies, we will not discuss pair-choosing implementations, given that the dependency between different assets can be computed in advance and is, therefore, not bound to speed issues in real-time. We will focus on the trade execution described in chapter 5.3, in which we start with a short introduction of the implemented algorithm, followed by performing low-latency optimizations. We then conclude this section by summarizing our findings. All benchmarks in this section are conducted on a 2.7GHz Quad-Core Intel Core i7, with 32Kb L1, 256Kb L2 and 8192Kb L3-Cache.

6.1 Algorithm Introduction

We implement a classic pairs trading strategy using tick data from S&P500 and NASDAQ. Historically the indices are strongly correlated and, thus, a suitable pair for performing a pairs trading strategy [48]. The strategy tracks the mean and standard deviation of the price ratio between the S&P500 and NASDAQ over a predetermined sliding window period. Now, a trading signal is generated whenever the ratio deviates from the computed mean by more than twice the standard deviation. As described in chapter 5.3, we will go short S&P500 and long NASDAQ if the ratio is above the mean plus twice the standard deviation, and we will open a long position in S&P500 and short NASDAQ if the ratio is below the mean by a magnitude greater than twice the standard deviation. Once the ratio moves inside the above outlined upper and lower thresholds, we will close our position by reverting the trade.
For the C++ implementation, we will operate an Asset and a Tick class. Data will be pushed from an external source into the program and stored into tick objects which will then be associated with two asset objects. All computations for the trading strategy will then be performed on the asset objects, namely S&P500 and NASDAQ. Pseudocode for the strategy is provided in 6.1.

```
1  void trading_algo() {
2      Asset SP500, NASDAQ;
3      loadData(SP500, "sp500.data"); // load tick data into asset object
4      loadData(NASDAQ, "nasdaq.data");
5
6      const double INITIAL_MONEY=1000.0; // Start with 1000$
7      double money=INITIAL_MONEY; // Portfolio worth
8      double portfolio[3]={1.0, -1.0, 1}; // The portfolio {x, y, z} indicates ↩
           weights in the portfolio of SP500, NASDAQ and cash, respectively.
9      const int TIME_CONSIDERED=60; // Amount of minutes we consider
10
11     double mean = calculate_initial_mean(SP500, NASDAQ);
12     double sd = calculate_initial_sd(SP500, NASDAQ);
13     double MARGIN = 2*sd;
```

```
14
15      while (continue_trading) {
16          // Update portfolio worth
17          money*=portfolio[2]+portfolio[0]*SP500.getData(i).getPrice()/SP500.↩
                getData(i-1).getPrice()+ \ portfolio[1]*NASDAQ.getData(i).getPrice()/↩
                NASDAQ.getData(i-1).getPrice();
18
19          update_mean(); // update sliding window mean if new tick is registered
20          update_sd(); // update sliding window sd if new tick is registered
21          MARGIN = 2*sd;
22
23          if (SP500.getData(i).getPrice()/NASDAQ.getData(i).getPrice()>mean+MARGIN)↩
                 {
24              // SP500 is outperforming NASDAQ, as their ratio is big. Thus, short ↩
                    SP500 and buy NASDAQ.
25              portfolio[0]=-1.0;
26              portfolio[1]=1.0;
27              portfolio[2]=1.0;
28          } else if (SP500.getData(i).getPrice()/NASDAQ.getData(i).getPrice()<mean-↩
                MARGIN) {
29              // SP500 is underperforming NASDAQ, as their ratio is small. Thus, ↩
                    buy SP500 and short NASDAQ.
30              portfolio[0]=1.0;
31              portfolio[1]=-1.0;
32              portfolio[2]=1.0;
33          } else {
34              // The ratio of SP500 and NASDAQ is inside the "normal" values.
35              portfolio[0]=0.0;
36              portfolio[1]=0.0;
37              portfolio[2]=1.0;
38          }
39      }
40  }
```

Listing 6.1: Pseudocode pairs trading strategy

6.2 Optimization

As already mentioned, the high frequency trader aims to reduce the latency of her code as much as possible. We modified our code to document the impact of several low latency techniques introduced in chapter 4.2. Ahead of this section, we would like to remind the reader that micro-optimizations, as performed in this chapter, are only worth considering if we can be sure that the rest of our setting, for example, the chosen algorithms or the infrastructure we work in, are already optimized.

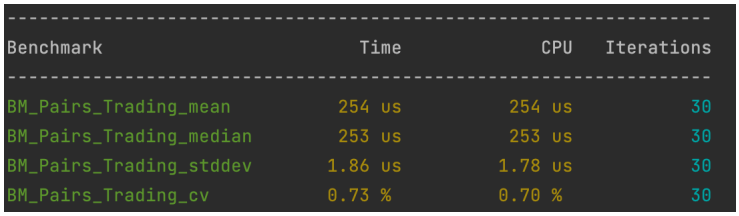
6.2.1 Macro Optimizations and Kernel Tuning

Optimizing those usually results in far greater speed improvements than starting with micro-optimizations immediately. In our case, replacing the mean and standard deviation algorithm with an online algorithm that only needs the latest tick data to update the metrics and storing the price ratio in a variable instead of recalculating it several times resulted in a more than 10x speed increase in our code. Computation metrics without these optimizations can be found in figure 6.1. Note that the computation time needed is for all 2388 data points; thus, it takes around 1.25us per

```
-------------------------------------------------------------------
Benchmark                        Time             CPU   Iterations
-------------------------------------------------------------------
BM_Pairs_Trading_mean         2991 us         2979 us           30
BM_Pairs_Trading_median       2953 us         2951 us           30
BM_Pairs_Trading_stddev       91.3 us         73.3 us           30
BM_Pairs_Trading_cv           3.05 %          2.46 %            30
```

Figure 6.1: Strategy speed without macro-optimization, performed for 2388 tickdata-points

tick before optimization. Metrics after optimization are shown in figure 6.2. We already decreased



Benchmark	Time	CPU	Iterations
BM_Pairs_Trading_mean	254 us	254 us	30
BM_Pairs_Trading_median	253 us	253 us	30
BM_Pairs_Trading_stddev	1.86 us	1.78 us	30
BM_Pairs_Trading_cv	0.73 %	0.70 %	30

Figure 6.2: Strategy speed with macro-optimization, performed for 2388 tickdata-points

computation time per tick to around 106ns. On an infrastructure level, isolating one CPU kernel and dedicating the application to it resulted in an average latency decrease of around 2ns per tick. Since Mac OS does not support kernel isolation and only task prioritization, which did not result in any measurable improvements, we performed the kernel isolation on a virtual Linux machine using tuned profiles as documented by RedHat [49].

6.2.2 Micro Optimizations

Interestingly, in line with our observation in listing 4.17 we see performance differences by reordering the if / else-if statements order. The best performance is obtained by checking the boundary breaches first, as done in 6.1. As expected, if we perform the "in-bound" check first, we get the slowest execution speed since we have to check two conditions instead of one and, thus, perform the most expensive check first and for all iterations. The slower statement is presented in 6.2. Note that by stating the in-bound check last, we never perform the check but use the *else* logic instead.

```
1  void trading_algo() {
2  //... some code
3      if ((SP500.getData(i).getPrice()/NASDAQ.getData(i).getPrice()<=mean+MARGIN) ↩
           && (SP500.getData(i).getPrice()/NASDAQ.getData(i).getPrice()>=mean-MARGIN↩
           )) {
4          // The ratio of SP500 and NASDAQ is inside the "normal" values.
5          portfolio[0]=0.0;
6          portfolio[1]=0.0;
7          portfolio[2]=1.0;
8      } else if (SP500.getData(i).getPrice()/NASDAQ.getData(i).getPrice()<mean-↩
           MARGIN) {
9          // SP500 is underperforming NASDAQ, as their ratio is small. Thus, buy ↩
               SP500 and short NASDAQ.
10         portfolio[0]=1.0;
11         portfolio[1]=-1.0;
12         portfolio[2]=1.0;
13     } else {
14         portfolio[0]=-1.0;
15         portfolio[1]=1.0;
16         portfolio[2]=1.0;
17     }
18 }
```

Listing 6.2: Slower if-check order

Using the logic in 6.2 instead of 6.1 results in a 9% slower code execution [A.3]. Interestingly we also note a speed difference by changing the order of the ">mean+MARGIN" and "<mean-MARGIN" statements, where putting the latter first results in 5% slower code. This can be explained by the fact that in our case, the ratio between SP500 and NASDAQ breached the upper bound more often; thus, stating ">mean+MARGIN" as the first check results in fewer checks overall, hence faster code. We note that testing different if-statement orders streamlined to the data/assets you are working with will result in faster code. Reordering different possibilities might lead to interesting results.
We then examined the effect of error handling in our code. As described in listing 4.7 we can increase latency performance by removing the slowpath from our hotpath. In our case, the slowpath is the error handling code. We consider the "getData()" and "getPrice()" method in 6.1. The code for these methods is given in 6.3.

```
//getData
Tick Asset::getData(int i) {
    if (i<0 || i>=mSize)  {throw Exception("Out of bound", "The index is out of bound"); }
    return mData[i];
}
//getPrice
double Tick::getPrice(void) {
    if (mPrice<0) {
        throw Exception("Negative price","The price of the stock must be non negative"); }
    return mPrice;
}
```

Listing 6.3: "getData" and "getPrice" implementation

In both cases, we throw an exception if we detect anomalies. Removing the checks and rerunning the strategy will result in a speed increase of nearly 8% or 9ns per tick [A.5], even if, as in our example, no exception is thrown at all. This can be explained by looking at the assembly code generated by the method with and without exception. Using the compiler explorer, we can see the output in figure 6.3. On the other hand, the code from the method without exception call

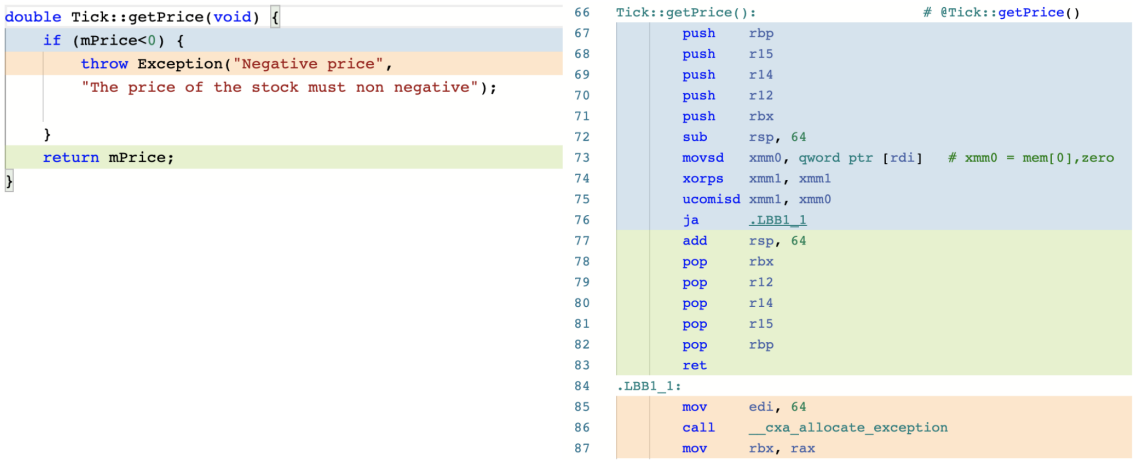


Figure 6.3: Assembly output for the "getPrice()" method with exception

is shown in figure 6.4, generating way less assembly code resulting in different execution speeds. We now focus on branch minimization. By taking a closer look at our *update_mean* and *update_sd*

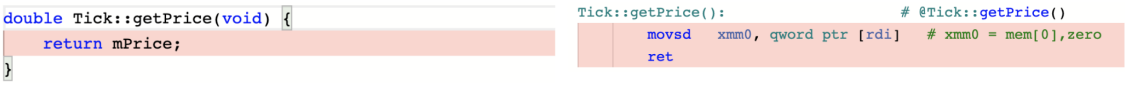


Figure 6.4: Assembly output for "getPrice()" without exception call

code from listing 6.1. As outlined in 6.4 we loop over our tick-data and update our sliding window metrics. The problem is that we have to change between the online update for mean and standard deviation and the first calculation over the window size every time we reset the trading strategy.

```
for(int i=Window_size; i<SP500.end()-1; i++) {
    if (i == Window_size) {
        for (int j = i - Window_size; j < i; j++) {
            // increment temporary sum counters
            sum_m += SP500.getData(j).getPrice() / NASDAQ.getData(j).getPrice();
            sum_s += pow(SP500.getData(j).getPrice() / NASDAQ.getData(j).getPrice(),2);
    } else{
```

```
8         //online update for mean and sd.
9         new_v = SP500.getData(i-1).getPrice() / NASDAQ.getData(i-1).getPrice();
10        old_v = SP500.getData(i-1-DAYS_CONSIDERED).getPrice() / NASDAQ.getData(i↩
              -1-DAYS_CONSIDERED).getPrice();
11        sum_m += new_v - old_v;
12        sum_s += new_v*new_v - old_v*old_v; };
13
14    mean= sum_m / Window_size;
15    sd=sqrt(sum_s/Window_size-mean*mean);
16    MARGIN = 2*sd
17    ..(pairs trading strategy code) // this might be a lot of code
```

Listing 6.4: Mean and standard deviation calculation with branching

Now instead of branching, we can unroll the if statement, which ultimately results in a lot more code and is against the software development principle "Don't repeat yourself" (DRY)[50], but can result in minor performance increases. In our case, we measure a computation time improvement of around 1.5% [A.6].

```
1  for (int j = 0; j < Window_size; j++) {
2      // starting computation for mean and standard deviation
3      sum_m += SP500.getData(j).getPrice() / NASDAQ.getData(j).getPrice();
4      sum_s += pow(SP500.getData(j).getPrice() / NASDAQ.getData(j).getPrice(),2);
5
6  mean= sum_m / Window_size;
7  sd=sqrt(sum_s/Window_size-mean*mean);
8  MARGIn = 2*sd
9  ..(pairs trading strategy code) // this might be a lot of code
10
11 for(int i=Window_size+1; i<SP500.end()-1; i++) {
12     //online update for mean and sd.
13     new_v = SP500.getData(i-1).getPrice() / NASDAQ.getData(i-1).getPrice();
14     old_v = SP500.getData(i-1-DAYS_CONSIDERED).getPrice() / NASDAQ.getData(i-1-↩
           DAYS_CONSIDERED).getPrice();
15     sum_m += new_v - old_v;
16     sum_s += new_v*new_v - old_v*old_v; };
17
18     mean= sum_m / Window_size;
19     sd=sqrt(sum_s/Window_size-mean*mean);
20     MARGIN = 2*sd
21     ..(pairs trading strategy code) // repeated code
```

Listing 6.5: Mean and standard deviation calculation without branching

One caveat by doing if-unwinding is that, as mentioned by Carl Cook [11], most of the time, your compiler will already optimize this out. Thus, the only way to check if it actually adds performance advantages or only makes your code less readable is by performing rigorous measurements.
In chapter 4.2 we explained the additional overhead added by heap memory allocation in C++. Contrary to preceding benchmarks, we will now time the strategy, including the data storing process instead of just the trading execution. We timed four different allocating possibilities. First, we stored our tick data in a vector by adding new data using the *push_back* method from the standard library. This automatically increases the vector's size every time its size limit is reached. This adds a lot of overhead and memory handling to the code; therefore, this method performed the slowest [A.8]. As mentioned in chapter 4.2 adding more memory to your infrastructure should nearly always be possible in a high frequency setting. Thus, our next approach heap allocates a vector of reasonable size in advance for storing the tick data. For our arguably small data size, this already results in a speed increase of around 17% [A.9]. Similar results are achieved when storing the data in a heap-allocated array [A.10]. The fastest approach was to store the data in a pre-allocated stack array. Doing so reduced latency by another 3% [A.11]. But storing data in a pre-allocated stack array should not be considered for large applications since this might result in a stack-overflow and crash the program or lead to losing or corrupting data [51].
As outlined in listing 4.26 the assembly code for using *unsigned_int* instead of *int* is longer because of integer overflow checks. Testing our algorithm with both approaches, on the other hand, did not deliver consistent, measurable results in faster performance. While over several iterations, the best speed was achieved by the version using *int* [A.12], it was not possible to consistently reproduce this result.

6.3 Findings

We now perform all optimization at once and compare it against the non-optimized code. We will only consider micro-optimizations for the comparison and perform all macro-optimizations before we test the code. Using Google Benchmark performing 100 iterations each, we receive a result for the slow code of 128nanoseconds per tick. Total results for all ticks are shown in figure 6.5. Using

Benchmark	Time	CPU	Iterations
BM_Pairs_Trading_mean	306236 ns	305395 ns	100
BM_Pairs_Trading_median	302611 ns	302228 ns	100
BM_Pairs_Trading_stddev	17180 ns	16300 ns	100
BM_Pairs_Trading_cv	5.61 %	5.34 %	100

Figure 6.5: Bechmark results for all ticks and non-optimized code

the same setup and re-evaluating the optimized code over 100 iterations with Google Benchmark, we received a result of 108nanosecond per tick. We are yielding a latency reduction of nearly 19%. Again, the result for all 2389 ticks is displayed in figure 6.6. The next chapter will discuss how

Benchmark	Time	CPU	Iterations
BM_Pairs_Trading_mean	259010 ns	258798 ns	100
BM_Pairs_Trading_median	257786 ns	257680 ns	100
BM_Pairs_Trading_stddev	4988 ns	4813 ns	100
BM_Pairs_Trading_cv	1.93 %	1.86 %	100

Figure 6.6: Bechmark results for all ticks and optimized code

these results might affect the profitability of HFT firms.

Chapter 7

Profitability Analysis

With this section, we build the link between high frequency trading latency and trading profits. Other research, as in Baron [24] has shown that HFT firms generate excess revenue with high sharp ratios, they do so by trading with all market participants over short time horizons and that the fastest firms generally report the best performance. We will provide a brief overview of the impact of absolute speed and relative speed on HFT profitability and then show how our techniques performed in chapter 6 would impact a firm's profitability.

7.1 Relative Speed Increase

We want to provide an overview of the concentration of profits and trading volume within the high frequency trading industry. Baron et al. [24] use the Herfindahl index to measure these concentrations. The index is famous for measuring market concentration and, in our case, can be constructed in the following way [52].

$$\mathrm{HP}_{i,t} = \sum_{i=1}^{N} \left[\frac{\mathrm{P}_{i,t}}{\mathrm{T\_P}_t} \right]^2, \tag{7.1.1}$$

where HP stands for the Herfindahl measure with regards to the profit, i is the i-th HFT firm in the market, N the total number of firms and $t \in \{0, 1, ..., T\}$ is the month in which the profit was reported. P and T_P are the profit and total profit, respectively. Similar notation yields the Herfindahl measure for the traded volume:

$$\mathrm{HV}_{i,t} = \sum_{i=1}^{N} \left[\frac{\mathrm{V}_{i,t}}{\mathrm{T\_V}_t} \right]^2, \tag{7.1.2}$$

where the notation is the same as for profit, but V and T_V now denote the volume traded per firm and total volume traded. We note that the domain for the measure is between 0 and 1, a higher number indicates a concentrated market within a few big players, and a lower number stands for a fragmented market. In a study conducted by Van Ness et al.(2005) [53], the authors report, using NASDAQ data, a number for the trading volume at around 0.14. More informative than the number itself might be the trend it follows. In 2005, the industry was relatively new; logically, we would expect more and more players to join the market and, thus, a decreasing Herfindahl measure. Interestingly, Van Ness reported precisely the opposite, which is a good indicator that a relatively small number of firms in the HFT industry with a technology-wise edge can earn consistent profits despite increasing competition. This gives rise to measuring the impact of latency on profits. A common way in literature to measure the latency of HFT firms is to time the duration between a passive trade in a specific asset, i.e. an open limit buy/sell order for this asset and switching to an aggressive trade in this security, i.e. sending in a market order [54]. To get the fastest possible response time Baron [24] for example, forms a distribution out of all the measured time differences and then chooses the 0.1% quantile of this distribution as the actual latency. Given that this procedure only measures the latency of HFT firms that switch from passive to aggressive trading orders, it is likely that studies conducted with this procedure underestimate the critical link between speed and profitability in high frequency trading. For completeness reasons, we would also like to sketch the idea of another method to measure relative speed among HFT firms. Queuing

latency, as outlined by Yueshen [55] does not measure the implicit speed of high frequency trading firms but sorts them by the number of limit orders posted at a new best bid/ask level. To be more specific, recent price changes in an order book which lead to a gap in the book, give rise to HFT firms to fill the gap and gain queue priority. Queuing latency now measures how often each specific firm submits the first limit order to fill the gap - the firm with the highest number is considered the fastest. Baron et al. [24] used decision latency to perform a linear regression on Swedish stock data to measure the impact of speed on HFT performance(P).

$$P_{i,t} = \beta_1 \log(\text{Decision-Latency}_{i,t}) + \beta_2 \text{Fastest}_{i,t} + \beta_3 \text{Top5}_{i,t} + \alpha + \epsilon_{i,t}. \tag{7.1.3}$$

where "Decision-Latency" denotes the speed of the HFT firm in absolute values, "Fastest" denotes if the firm has the lowest latency, "Top5" if the firm is among the five fastest firms, and "α" is the intercept. The authors report statistical significance for being among the Top5 or the fastest high frequency trading firm. On the other hand, the absolute latency value is not statistically significant, which lets us infer that relative speed is more important than absolute speed in terms of profit/performance. Evaluating the regression 7.1.3 Baron et al. report trading revenues as much as five times as high for the fastest firm compared to a firm not among the top5. Additionally, being under the five fastest firms results in a 2.5 times increase in revenue. Regarding the sharp ratio of HFT firms, being the fastest firm results in a ratio of 12.07 compared to 8.87 for being among the top5 and 2.26 for slower HFT firms. Similar observations can be made by considering profit instead of revenue. The results imply, on the one hand, that it is highly profitable to strive for faster trading systems and, on the other hand, that relative speed compared to competitors might be more important than the absolute execution speed of the strategy.

7.2 Absolute Speed Increase

We already outlined that it is more important to focus on the actual competition for HFT firms and that beating them in terms of speed results in most of the profitability increases linked to latency in high frequency trading. This still leads to the question of how extensively the fastest firm beats other firms or how much the second fastest firm would have to increase its trading speed. Recent studies answered this question in the case of latency arbitrage strategies. These strategies can be described as a class of strategies that use speed advantages to exploit price discrepancies of similar financial assets in different markets. A mathematical definition is formulated by Wah [21]: Given two markets M1 and M2, let B be short for bid and A for ask. A latency arbitrage opportunity LAO is defined by:

- Two crossed markets, $\text{B}_{M1} > \text{A}_{M2}$.
- Bid_A and A_{M2} are better or egeual than the national best bid (NBB) and ask (NBA), i.e. $\text{B}_{M1} \geq \text{NBB}$ and $\text{A}_{M2} \leq \text{NBO}$.
- The time between the beginning and end of the above measured occurrences is positive, i.e. it is actual possible to exploit the opportunity.

Aquilina [23] showed that for latency arbitrage strategies applied in the UK equity market, the average time between the fastest HFT firm and the second fastest was 5-10microseconds.
An interesting result related to the impact of absolute speed on HFT firms is given by Ende et al.(2011) [6]. Rather than showing an actual profitability increase in combination with a reduction in latency, the authors show the connection between trading speed and adverse selection / unfavourable order book changes. They analysed an active trading strategy with different latencies for a single stock in the German stock market. Regressing over the data points yields the interesting result that an increase of 1% in latency leads to a probability increase of 0.9% in being prone to unfavourable order book changes. Their results are shown in figure 7.1

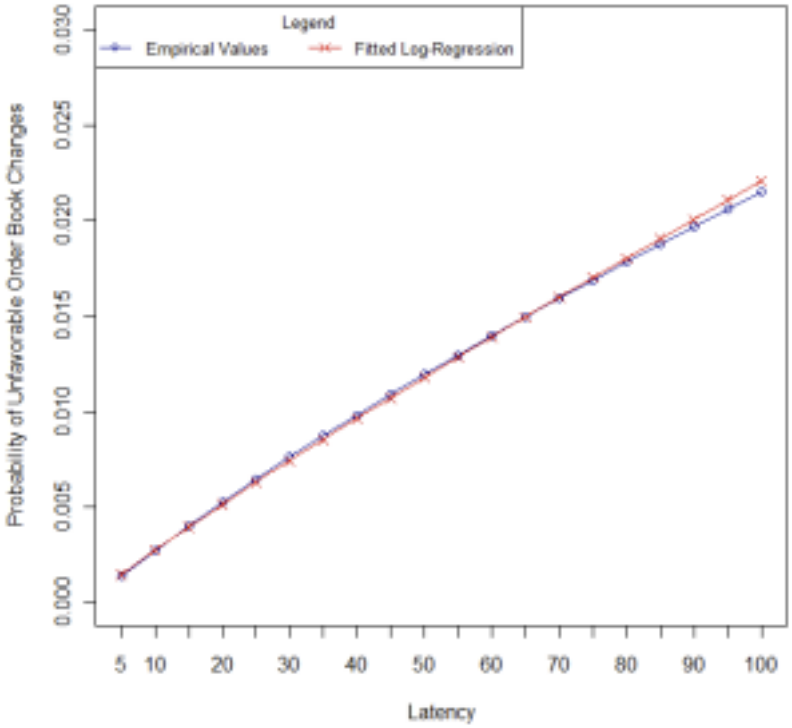


Figure 7.1: Probability of unfavourable order book changes concerning latency changes in % [6]

7.3 Profitability Conclusion

Taking the findings from chapter 7.1 and 7.2 into consideration, combined with our optimisation results of section 6, we note that we are unfortunately only able to make conclusions about profitability increases due to absolute speed, but not to relative speed, since we are not operating on a practically used trading engine, neither do we know the speed difference to other HFT firms. As shown in 6.3 we could increase the execution speed of our trading algorithm by nearly 19%, which would result in a decrease of around 17% in being exposed to unfavourable order book changes using the results obtained by Ende [6].

Chapter 8

Conclusion

8.1 Concerns and Benefits of HFT

Does high frequency trading make the markets more efficient / better? Trading firms and other users of HFT strategies think so, and there are supporting arguments one can make in their case. On the other hand, there are plenty of reasonable concerns too. The following section will attempt to outline both positions.

8.1.1 Benefits

We will discuss the following supporting arguments, tighter, deeper markets, consistent prices, and the lack of panic in computers.

Tighter and deeper Markets

It is presumably without debate that the technological capabilities of trading firms have advanced considerably in recent years. With more and more firms having access to the technology, it is not unreasonable to see how this can benefit tighter markets (smaller bid-ask spread) and deeper markets (bigger sizes). All else being equal tighter markets provide the investor with better prices, and deeper markets result in bigger possible trade sizes at these better prices. A simple example would look like the following. Imagine two market makers just bought 100 shares of a stock for one dollar each. Both market makers want to complete the round-trip trade by selling their shares for a higher price. Market maker A starts offering its shares at 1.10$. To be more attractive to other market participants, market maker B offers its shares for 1.09$ to gain priority over market maker A. A sees that and decides to lower its offer to 1.08$, followed by B reducing it to 1.07$ for the same reasoning. This continues until the offer price is as low as possible such that the market makers do not make a loss by trading the shares. Now, if we had even more market makers - the market would not only tighten but also make more securities available at those tighter prices. "We have moved from a market in which humans manually traded to one in which computers execute the bulk of trades without human intervention. Volume is higher. Trade size has become smaller as it is now cheaper for institutions to divide orders into smaller slices to reduce market impact." [56] This argument is further backed by Brogard, who showed in a study that HFT firms provide the best bid and offer quotes for the majority of the trading day in U.S. markets [57].

Price Consistency

Arbitrageurs utilising high frequency trading strategies are solely looking for different prices in securities across other markets and, if successful, buy for the lower price and sell for the higher one - all in a matter of microseconds. Thus, investors buying financial assets do not have to worry about which of the multiple exchanges they should choose because we can be very sure that the price will be virtually the same everywhere the asset trades. To be exhaustive, we note that additional measures to provide fair prices for the investor across different markets are also guaranteed by regulations, like the National Best Bid and Offer (NBBO) in the U.S., which requires the broker to execute customer trades at the best available price across exchanges [58]. Additionally, as described

in the pairs trading section, HFT firms will ensure that no misprices exist between highly correlated financial assets, again providing the investor with fair prices.

Rational markets

Considering how much of the downwards movement during a stock crash is related to people emotionally selling their positions is interesting. It is natural to sell your stocks when everyone else does, and we also think about selling our positions once the value starts to decline. High frequency trading systems or automated trading systems are not bound to that emotional decision-making but stick to their trading signals. It can even stabilise the market by counteracting to above-mentioned emotional sell-offs. Think about a market sell-off accelerated by panic selling - then some trading signals will notice that some financial assets are undervalued and start buying these to make a profit. This, in return, results in slowing down the market's downward movement. The same reasoning holds if a company's stock rallies anomalous due to erroneous news or similar; thus, automatic trading slows down the upward trend and protects retail investors from buying mispriced securities. "During the largest price increase, HFTs buy and demand less liquidity than normal. The same relationship is true during price declines: during the largest price declines, HFTs decrease their liquidity demand and increase the liquidity they provide." [57]

8.1.2 Concerns

On the concern side, we will discuss price manipulations, colocation, and other risk concerns.

Price manipulation

Let us state an example which is sometimes seen as price manipulation. A common way for investors to not display their whole intention in the order book is to use iceberg orders, presenting their order in small pieces and re-posting it every time the order has been filled. Knowing this practice market makers have developed a counter strategy known as towing the iceberg [8]. Imagine a market with 500 stocks for the best bid at 1.02$ and 300 stocks for the best ask at 1.07$. Every time the 300 stocks ask is bought, the limit order instantly refreshes, which is an indicator for our market maker that we have an iceberg order in the order book. She will now attempt to short the stock to drive its price down to make the investor adjust its iceberg order downwards and then repurchase these shares to make a profit. The market maker starts doing this by selling 500 stocks for the 1.02$ bid, resulting in a new market of 1.01$ - 1.06$(downward adjustment). The investor adjusts its iceberg ask accordingly to 1.05$ to get priority in the market. Then the market maker sells again at the 1.01$ bid to drive prices further down and so on. Assume that at the end, we have a market of 0.98$ - 1.01$, then the market maker closes its short position by buying back the stocks at 1.01$ and makes a profit, while the iceberg investor got worse prices for his order. While this practice is perfectly legal, critics argue that this is an attempt to move prices for someone's own benefit and should be forbidden.

Increased Volatility

The rapid spread of high frequency trading activity has increased the amount of research conducted on its impact on the financial markets. Currently, papers are not unambiguous about its effect on market quality [59]. In contrast, some research states that HFT reduces market volatility [60], while other papers conclude that it might increase volatility [61]. The main argumentation behind saying that it will increase volatility is the following. HFT strategies execute a large number of trades at a large scale at the same time since every trade has only a small profit margin per asset. Now taking into consideration the number of HFT firms operating in the financial markets, each with particular strategies and agendas, taking big trades all the time. Those strategies are bound to interact, resulting in market movements that otherwise would not exist. Additionally, we must remember that high frequency traders make the most profit in volatile markets since it makes it easier to finalise round-trip trades. Thus, it just seems logical that HFT firms want to increase volatility. It is, furthermore, not only the stock market but also options trading. Volatility is arguably the most critical input into an option pricing formula. The more uncertainty about its value, the more opportunities exist to make money with volatility in the options market. We conclude that it is difficult to find statistical evidence for this point, but advocates of this claim have a reasonable chain of arguments supporting their statement.

Trading for the sake of trading

Data from Nasdaq estimates that around 50% of all stock trading volume in the U.S. is driven by high frequency trading [62]. Given this number and thinking that a sizeable amount of this volume is HFT firms trading with other HFT firms, then the question about how this is efficient and if all of this trading is overtaxing the system is legit. One example of how this can harm other investors is the increased bandwidth required in trading nowadays. If the extra traffic is attributable to HFT-HFT trading, then why should other market participants also pay the additional costs for more bandwidth. Additionally, this raises the question of whether extra costs for exchanges are created, which will ultimately be passed back to the investor [8].

Colocation

HFT firms with their server colocated at one of the exchanges will get their orders into the systems matching engine way earlier than firms elsewhere in the region. This raises the point of fairness, whether exchanges should discriminate between those who can pay for faster access and those who cannot. In India, for example, it also happened that the colocation service could be used for scam activities, where some trading firms tricked the system with getting access to data in front of their competition [63]. But if implemented correctly and offering everyone who wants to pay for it the opportunity to get access, it is difficult to defend this argument. If deemed unfair, the same point could be made regarding the superior technological infrastructure built by HFT firms and so on. Therefore, as long as there is no fraud involved, colocation seems to be a fair service which increases the speed/efficiency of global markets.

Suppose we keep the high complexity of high frequency trading in mind, combined with the fast-paced dynamics of financial markets and the amount of money involved or can be made. In that case, it is no surprise that, on the one hand, many practitioners are trying to find loopholes in the system, and, on the other hand, many critics are complaining about possible unfairness. Fortunately, if found, most dubious practices are self-eliminating and only work until the competition figures them out or if the rules change accordingly. Additionally, the increase in public awareness about HFT firms pressures regulators and firms to be more transparent [8].

8.2 Future Work

To give inspiration on how future work on this topic could look like we want to line out that still barely any literature on low latency reduction methods in high frequency trading exists. Therefore, it would be interesting to collaborate closer with high frequency trading firms and test the optimisations in this paper on an actual live trading system with end-to-end time measurement. Further, an interesting result would be to analyse the average speed difference between single HFT firms on different exchanges, which could then be used to connect the profitability analysis of relative trading speed with the one of absolute values. Finally, investigating a fast model-based pairs trading strategy superior to the classic approach would be valuable.

8.3 Summary

In this thesis, we have focused on applying low latency programming techniques to high frequency trading systems. We have demonstrated that several techniques introduced in C++ meetings over the past years reduce trading code latency times. We did so by starting with system architecture optimisations like kernel tuning and then moved on to actual micro-optimisations, which included, among other things, slowpath removal, branching minimisation, deterministic code flow and type engineering. We then introduced different academic strategies for finding and building statistical arbitrage opportunities. We outlined the challenge of finding suitable trading pairs for a strategy, which can range from simple but efficient strategies like the distance approach to more mathematically sophisticated time series approaches that model the underlying asset price processes. We finished this section by showing standard methods in literature to execute the trading strategy, combining the two preceding chapters, we then built our own trading strategy, which used a pairs

trading strategy for the SP500 and NASDAQ index of the U.S. stock market and performed several code micro-optimisations to obtain an overall latency reduction of more than 18%. To set this number into perspective, the next section of this thesis evaluated the relation between the profitability of HFT firms and their execution speed. We found that for fast high frequency firms, the relative speed compared to their competitors is more important for generating excess profit than speed increase in absolute terms. The speed increase generated on our own implemented trading algorithm would decrease the probability of being prone to negative order book changes by around 17%. We then finished the thesis by outlining common concerns about high frequency trading and providing inspiration for future work on this topic.

Appendix A

First Appendix

This appendix section is dedicated to the results produced by our Google Benchmark tests. For conciseness, we will only include the tests conducted for the full data set and not single tick tests. For comparison, it is only suitable to compare the results of each section against each other since we modified the code slightly from section to section to enable benchmarking and testing of the desired modifications.

If-check Order

	Time	CPU	Iterations
Mean	276285 ns	275491 ns	100
Median	275512 ns	275275 ns	100
Stddev	4583 ns	2535 ns	100

Table A.1: Performing the "In-bound" check first

	Time	CPU	Iterations
Mean	277631 ns	277146 ns	100
Median	277166 ns	276755 ns	100
Stddev	3390 ns	2514 ns	100

Table A.2: "<mean-MARGIN" first

	Time	CPU	Iterations
Mean	266679 ns	266222 ns	100
Median	266265 ns	265963 ns	100
Stddev	3104 ns	2459 ns	100

Table A.3: ">mean+MARGIN" first

Exception Handling

	Time	CPU	Iterations
Mean	279750 ns	278759 ns	100
Median	278041 ns	277686 ns	100
Stddev	5933 ns	4431 ns	100

Table A.4: Trading strategy with exceptions

	Time	CPU	Iterations
Mean	255880 ns	254399 ns	100
Median	254413 ns	254068 ns	100
Stddev	9168 ns	3015 ns	100

Table A.5: Trading strategy without exceptions

Branch Minimization

	Time	CPU	Iterations
Mean	281382 ns	280793 ns	100
Median	280536 ns	280261 ns	100
Stddev	3389 ns	2120 ns	100

Table A.6: Metric computation with branching

	Time	CPU	Iterations
Mean	277799 ns	276865 ns	100
Median	276467 ns	276083 ns	100
Stddev	5144 ns	3120 ns	100

Table A.7: Metric computation without branching

Memory Allocation

	Time	CPU	Iterations
Mean	101369 us	100203 us	100
Median	101105 us	100147 us	100
Stddev	1331 us	551 us	100

Table A.8: Vector allocation without preallocation

	Time	CPU	Iterations
Mean	86020 us	85743 us	100
Median	85510 us	85423 us	100
Stddev	2410 us	1427 us	100

Table A.9: Vector with preallocation

	Time	CPU	Iterations
Mean	89933 us	89614 us	100
Median	89061 us	88989 us	100
Stddev	2846 us	2122 us	100

Table A.10: Heap array allocation

	Time	CPU	Iterations
Mean	83541 us	83409 us	100
Median	83335 us	83204 us	100
Stddev	760 us	616 us	100

Table A.11: Stack array allocation

Type Engineering

	Time	CPU	Iterations
Mean	263761 ns	263588 ns	100
Median	262491 ns	262332 ns	100
Stddev	4407 ns	4337 ns	100

Table A.12: Fastest result obtained by using normal integer

	Time	CPU	Iterations
Mean	266809 ns	266453 ns	100
Median	265786 ns	265625 ns	100
Stddev	5646 ns	4554 ns	100

Table A.13: Fastest result obtained by using unsigned integer

Bibliography

[1] Kernel bypass, howpublished = https://blog.cloudflare.com/kernel-bypass/, note = Accessed: 2022-06-02.

[2] Striving for Ultimate Low Latency, howpublished = https://www.youtube.com/watch?v=vzdl0q91mrm, note = Accessed: 2022-06-02.

[3] Instruction pipeline, howpublished = https://en-academic.com/dic.nsf/enwiki/141209, note = Accessed: 2022-07-11.

[4] The price of dynamic memory: Allocation, howpublished = https://johnysswlab.com/the-price-of-dynamic-memory-allocation/, note = Accessed: 2022-06-06.

[5] Comparison of data structures, howpublished = https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html, note = Accessed: 2022-06-06.

[6] Bartholomäus Ende, Tim Uhle, and Moritz Weber. The impact of a millisecond: Measuring latency effects in securities trading. page 116, 01 2011.

[7] LODEWIJK PETRAM and LYNNE RICHARDS. *The World's First Stock Exchange*. Columbia University Press, 2014.

[8] Michael Durbin. *All About High-Frequency Trading*. McGraw-Hill, ¡country¿US¡/country¿, 2010.

[9] Specialists SEC, howpublished = https://www.sec.gov/fast-answers/answersspecialisthtm.html, note = Accessed: 2022-07-12.

[10] Scott Gibson, Rajdeep Singh, and Vijay Yerramilli. The effect of decimalization on the components of the bid-ask spread. *Journal of Financial Intermediation*, 12(2):121–148, April 2003.

[11] When a Microsecond Is an Eternity: High Performance Trading Systems in C++, howpublished = https://www.youtube.com/watch?v=nh1tta7purm, note = Accessed: 2022-06-02.

[12] William Goetzmann, K. Rouwenhorst, and Evan Gatev. Pairs trading: Performance of a relative value arbitrage rule. *Review of Financial Studies*, 19:797–827, 02 2006.

[13] Binh Do and Robert Faff. Does simple pairs trading still work? *Financial Analysts Journal*, 66(4):83–95, 2010.

[14] David Bowen, Mark Hutchinson, and Niall O'Sullivan. High frequency equity pairs trading: Transaction costs, speed of execution and patterns in returns. *The Journal of Trading*, 5:31–38, 06 2010.

[15] Vayu Kishore. Optimizing pairs trading of us equities in a high frequency setting. 04 2012.

[16] Christopher Krauss and Johannes Stübinger. Nonlinear dependence modeling with bivariate copulas: Statistical arbitrage pairs trading on the sp 100. FAU Discussion Papers in Economics 15/2015, Friedrich-Alexander University Erlangen-Nuremberg, Institute for Economics, 2015.

[17] In-Memory Techniques, Low-Latency Trading, howpublished = https://www.youtube.com/watch?v=ybnpsqooork, note = Accessed: 2022-06-02.

[18] HFT Low Latency Code, howpublished = https://www.youtube.com/watch?v=_0au8s-hfqi, note = Accessed: 2022-06-02.

[19] Efficiency with Algorithms, Performance with Data Structures, howpublished = https://www.youtube.com/watch?v=fhnmrkzxhws, note = Accessed: 2022-06-02.

[20] HFT Low Latency Code, howpublished = https://www.youtube.com/watch?v=6alw1gwqmts, note = Accessed: 2022-06-02.

[21] Wah Elaine. How prevalent and profitable are latency arbitrage opportunities on u.s. stock exchanges. 02 2016.

[22] B. Weller and University of Chicago. *Intermediation Chains and Specialization by Speed: Evidence from Commodity Futures Markets*. University of Chicago, Division of the Social Sciences, Department of Business and Economics, 2013.

[23] Matteo Aquilina, Eric Budish, and Peter O'Neill. Quantifying the High-Frequency Trading "Arms Race"*. *The Quarterly Journal of Economics*, 137(1):493–564, 09 2021.

[24] Matthew Baron, Jonathan Brogaard, Björn Hagströmer, and Andrei Kirilenko. Risk and return in high-frequency trading. *Journal of Financial and Quantitative Analysis*, 54(3):993–1024, 2019.

[25] Stock Exchange Data Center Trading, howpublished = https://www.nasdaq.com/solutions/nasdaq-co-location, note = Accessed: 2022-06-02.

[26] Spread networks, howpublished = https://en.wikipedia.org/wiki/spread_networks, note = Accessed: 2022-06-02.

[27] Fair and non-discriminatory co-location services, howpublished = https://www.handbook.fca.org.uk/techstandards/mifid-mifir/2017/reg_del_2017_573_oj/005.html?date=2021-01-01, note = Accessed: 2022-06-02.

[28] Kernel definition, howpublished = http://www.linfo.org/kernel.html, note = Accessed: 2022-06-02.

[29] Improving Linux networking performance, howpublished = https://lwn.net/articles/629155/, note = Accessed: 2022-06-02.

[30] RedHat Kernel Bypass, howpublished = https://access.redhat.com/articles/1391433, note = Accessed: 2022-06-02.

[31] inplace_function.h, howpublished = https://github.com/wg21-sg14/sg14/blob/master/sg14/inplace_function.h, note = Accessed: 2022-06-06.

[32] Xin Wei, Liang Ma, Huizhen Zhang, and Yong Liu. Multi-core-, multi-thread-based optimization algorithm for large-scale traveling salesman problem. *Alexandria Engineering Journal*, 60(1):189–197, 2021.

[33] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, USA, 1st edition, 2019.

[34] Gabriel Dos Reis and Bjarne Stroustrup. General constant expressions for system programming languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, page 2131–2136, New York, NY, USA, 2010. Association for Computing Machinery.

[35] size_t cpp reference, howpublished = https://en.cppreference.com/w/cpp/types/size_t, note = Accessed: 2022-06-06.

[36] Agner Fog. Optimizing software in c++: An optimization guide for windows, linux and mac platforms, 2004.

[37] Christopher Krauss. Statistical arbitrage pairs trading strategies: Review and outlook. *Journal of Economic Surveys*, 31(2):513–545, 2017.

[38] Ganapathy Vidyamurthy. Pairs trading : Quantitative methods and analysis / g. vidyamurthy. 01 2004.

[39] Rongmao Zhang and Ngai Hang Chan. Nonstationary linear processes with infinite variance garch errors. *Econometric Theory*, 37(5):892–925, 2021.

[40] Philip Hans Franses. Seasonality, non-stationarity and the forecasting of monthly time series. *International Journal of Forecasting*, 7(2):199–208, 1991.

[41] Robert Elliott, John van der Hoek, and William Malcolm. Pairs trading. *Quantitative Finance*, 5:271–276, 06 2005.

[42] Binh Do, Robert Faff, and Kais Hamza. A new approach to modeling and estimation for pairs trading, 2006.

[43] Marco Avellaneda and Jeong-Hyun Lee. Statistical arbitrage in the us equities market. *Quantitative Finance*, 10(7):761–782, 2010.

[44] Yan-xia Lin, Mccrae Michael, and Gulati Chandra. Loss protection in pairs trading through minimum profit bounds: A cointegration approach. *Journal of Applied Mathematics and Decision Sciences*, 2006, 08 2006.

[45] Heni Puspaningrum, Yan-xia Lin, and Chandra Gulati. Finding the optimal pre-set boundaries for pairs trading strategy based on cointegration technique. *Journal of statistical theory and practice*, 4, 09 2010.

[46] Nicolas Huck. Pairs trading and outranking: The multi-step-ahead forecasting case. *European Journal of Operational Research*, 207(3):1702–1716, 2010.

[47] Ilse Botha, Yolanda Stander, and Daniel Marais. Trading strategies with copulas. *Journal of Economic and Financial Sciences*, 6:103–126, 04 2013.

[48] Srikanth Potharla and Dr Aparna. Global stock market integration-a study of select world major stock markets. *Researchers-world: Journal of Arts, Science Commerce*, III:72–80, 01 2012.

[49] Isolating CPUs Using tuned-profiles-realtime, howpublished = `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/isolating_cpus_using_tuned-profiles-realtime`, note = Accessed: 2022-07-01.

[50] Andrew Hunt and David Thomas. *The Pragmatic programmer : from journeyman to master*. Addison-Wesley, Boston [etc.], 2000.

[51] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2007.

[52] Charles R. Laine. The herfindahl-hirschman index: A concentration measure taking the consumer's point of view. *The Antitrust Bulletin*, 40(2):423–432, 1995.

[53] Bonnie F. Van Ness, Robert A. Van Ness, and Richard S. Warr. The impact of market maker concentration on adverse-selection costs for nasdaq stocks. *Journal of Financial Research*, 28(3):461–485, 2005.

[54] B. Weller. Liquidity and high frequency trading. *USC FBE FINANCE SEMINAR*, 2012.

[55] Bart Zhou Yueshen. (preliminary and incomplete) queuing uncertainty (a job market paper). 2013.

[56] James Angel, Lawrence Harris, and Chester Spatt. Equity trading in the 21 st century. *Quarterly Journal of Finance*, 01:1–53, 03 2011.

[57] Jonathan Brogaard. High frequency trading and its impact on market quality. 04 2011.

[58] National best Bid and Offer, howpublished = `https://www.law.cornell.edu/cfr/text/17/242.600`, note = Accessed: 2022-07-01.

[59] Valeria Caivano. The impact of high-frequency trading on volatility. evidence from the italian market. *SSRN Electronic Journal*, 01 2015.

[60] Cristina McEachern Gibbs. High-frequency trading benefits traditional trader. *Wall Street Technology*, 6(3):12–, 2010.

[61] Frank Zhang. High-frequency trading, stock volatility, and price discovery. *SSRN Electronic Journal*, 12 2010.

[62] 50percent Nasdaq, howpublished = `https://www.nasdaq.com/glossary/h/high-frequency-trading`, note = Accessed: 2022-07-01.

[63] NSE colocation scam, howpublished = `https://www.moneycontrol.com/news/business/companies/explainer-nse-colocation-case-what-happened-faq-3985511.html`, note = Accessed: 2022-07-01.