A Multithreaded Forex Algo Backtesting System

Yoshiharu Sato†
Warsaw School of Economics

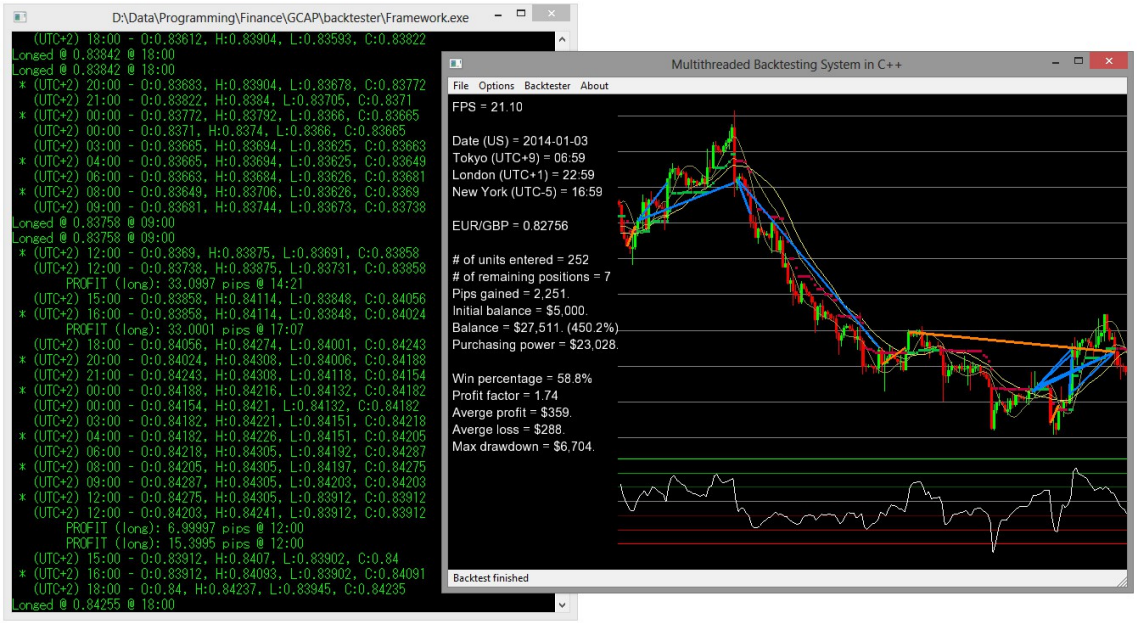

Figure 1. *Our backtesting system in action*

Abstract

Over the past decade, algorithmic trading has been increasingly growing in the foreign exchange market. Thanks to the introduction and subsequent popularization of the MetaTrader 4 electronic trading platform, even individual traders are now able to use their own algorithms to trade currencies in the market.

In order to devise a forex trading algorithm that achieves a high rate of return, it is vital for developers to backtest their algorithms using different sets of parameters, different time frames, different time periods, and different currency pairs. Backtesting therefore is inherently a time-consuming process, thus it is critical for developers to reduce the amount of time taken up by backtesting in order to increase productivity.

For this reason, we developed a forex trading algorithm backtesting system completely from scratch so as to maximize the speed, flexibility, and accuracy of backtesting. Using our framework, C++ programmers are able to implement their own forex trading algorithms and to backtest them efficiently under various kinds of conditions.

Keywords: Algorithmic Trading, Foreign Exchange Market, Technical Analysis, Quantitative Finance, Backtesting System, Concurrent Programming

---

†E-mail: yoshi2233@gmail.com

1. Introduction

Advancements in information technology as well as the full computerization of financial markets in the late 1980's and 1990's gave birth to algorithmic trading – the application of electronic platforms for entering trading orders using computer algorithms which automatically execute pre-defined trading strategies without human intervention. Algorithmic trading is also called black-box trading, as it is vital for trading algorithm developers to keep confidential the mechanics of their algorithms in order to maintain their competitive advantage in the market. Today, algorithmic trading is extensively employed by various types of financial institutions (both sell-side and buy-side firms), the most notable ones of which are investment banks and hedge funds.

Since the introduction of algorithmic trading, public attention has been primarily focused on the rate of algorithmic trading adoption in the stock market – computer algorithms today account for more than 50% of all US stock trading volume (Figure 2). One of the typical ways in which buy-side firms use algorithms in the stock market is high-frequency trading (HFT), in which firms trade stocks rapidly and recurrently by moving in and out of positions in fractions of a second. Sell-side firms usually use algorithms to provide liquidity to the market by generating and executing orders automatically. Most stock exchanges around the world nowadays accept trading orders created by algorithms via direct market access (DMA) [1], through which buy-side firms are able to place orders directly onto the order book of an exchange without recourse to market makers. A number of stock exchanges around the world also provide co-location facilities wherein firms are able to locate their algorithmic trading systems within the exchange's premises, thereby ultimately minimizing latency.

In the past years, the emergence of currencies as a legitimate asset class has resulted in rapid adoption of algorithmic trading in the forex market. Figure 2 shows the percentage shares of algorithmic trading in the stock, future, forex and bond markets in the US, where a clear upward trend of algorithmic trading adoption in the forex market can be observed (n.b., the reason that the share of algorithmic trading in the stock market has decreased over the past few years is because the regulation on algorithmic trading has been raised after HFT was blamed to have caused a significant market disturbance in 2010, commonly known as the Flash Crash [2]).

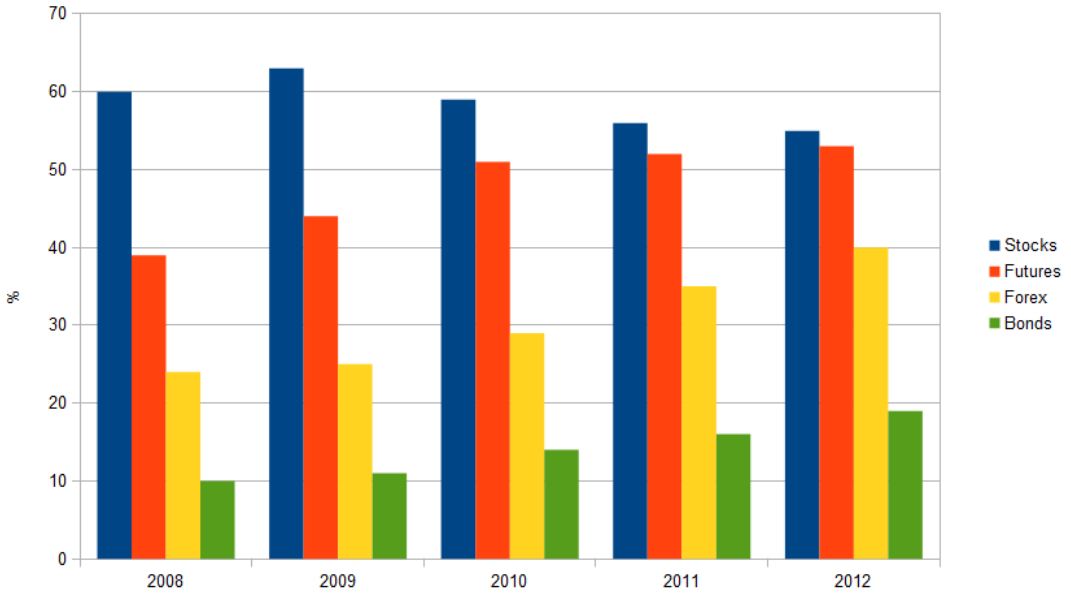

Figure 2. *Percentage shares of algorithmic trading in US markets (FSOC: Financial Stability Oversight Council [3])*

One of the advantages of algorithmic trading in the forex market is that traders themselves do not have to always keep an eye on the market to catch a signal for new entering/exiting orders. In fact, the forex market is open 24 hours a day except weekends (i.e., from Sunday 20:15 GMT until Friday 22:00 GMT), hence only computers can possibly monitor the market throughout the entire period, unless there are multiple traders working collaboratively in different time frames. Being able to catch more signals in the market gives traders more opportunities for making profits, as well as smaller risks to expand losses. Algorithmic trading in the forex market is therefore advantageous and prospective.

Since the introduction of MetaTrader 4 (MT4) [4] electronic trading platform in 2005, algorithmic trading in the retail forex market has been growing exponentially as MT4 allows its users to create their own trading algorithms. However, MT4 is not necessarily the best platform to develop algorithms for, particularly because the platform's backtester has several serious limitations. Therefore, as the goal of this research, we developed a backtesting system of forex trading algorithms completely from scratch so as to maximize the speed, flexibility, and accuracy of backtesting. Using our framework, C++ [5] programmers are able to implement their own forex trading algorithms and to backtest under various kinds of conditions efficiently.

2. Foreign Exchange Market

2-1. Overview

Traders who participate in the forex market usually buy (long) and/or sell (short) currency pairs, such as EUR/USD (i.e., the exchange rate of Euro denominated in the US dollar), in which example EUR is called the quote currency and USD is called the base currency. Theoretically there are C(n, 2) currency pairs where n is the number of all the currencies in the world, however the number of currency pairs that are actually tradable in the market is much smaller since some currencies are either restricted for free trade, or too minor and thus there is not much demand for trading. What should be noted in regards to currency pairs is the fact that even if a pair does not include USD, its exchange transaction in the forex market always involves buying and selling of USD internally because the US dollar is the world's reserve currency. For instance, when a trader buys EUR/GBP, in actuality he first buys US dollars with British pounds and subsequently sells the US dollars in exchange for Euros. The currency rate in this case is therefore calculated as: $EUR/GBP = USD/GBP \times EUR/USD$.

In the forex market the amount of profits or losses is often denoted in *pip* (percentage in point; 1/100th of a percentage), which is a unit of change in a currency pair's exchange rate. For major currency pairs (except Japanese yen crosses) a pip is one unit of the fourth decimal point. A currency pair is typically traded in lot size of 100,000 units of the base currency (a.k.a., “standard lot”). A trading position of one standard lot that experiences a change in the exchange rate of 1 pip thus changes in value by 10 units of the quoted currency.

Prior to the development of retail forex electronic trading platforms in the late 1990's, the forex market was restricted to large financial institutions. It was the commoditization of personal computers, the rapid expansion of the internet, the advent of electronic trading platforms, and the inflows of many retail forex brokers that started the exponential growth of retail forex trading. Individual traders are today not only able to trade currencies on margin (i.e., they need to put down only a small percentage of the trade size) but also able to trade based on algorithms using technical analysis software that has a broker interface, such as MetaTrader 4 (MT4) [4] and NinjaTrader [6].

2-2. Market Structure

The significant difference of the forex market compared to the stock market is the fact that the forex market is decentralized, meaning there is no centralized location for order execution. This is attributed to the very nature of currencies (i.e., they are independently issued by central banks around the world). Another important aspect of the forex market is that it is cascaded, or divided into levels of access. At the top is the interbank market, which is mostly consisted of the larger commercial banks (e.g., Deutche Bank, Citi, UBS; a.k.a., "prime banks"), who act as broker-dealers and involve in the determination of actual exchange rates. From the interbank market, the exchange rates are disseminated to the lower levels with wider spreads (i.e., the difference between the bid and ask prices, which is the primary source of a broker's profits).

Although individual traders constitute a growing segment of the forex market, they are usually at the lowest level of access in the market. This is because of the size of their trading volume – if a trader can guarantee large numbers of transactions for large amounts, they can demand a tighter spread. Most individual traders therefore are price takers and participate in the forex market indirectly through retail brokers or banks:

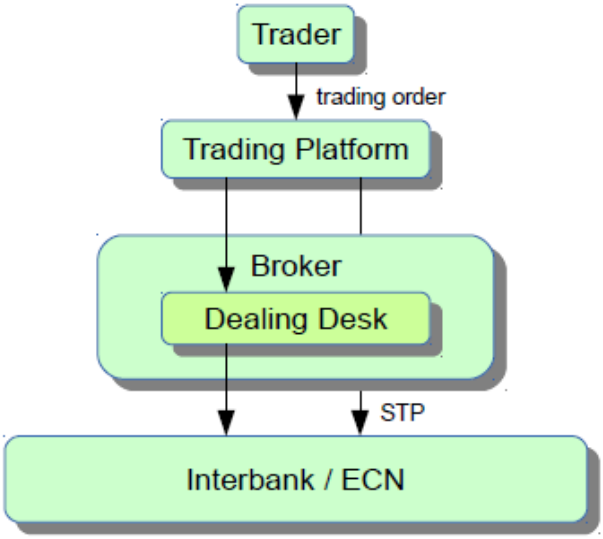

Figure 3. *Flow diagram of trading orders in the retail forex market*

Figure 3 illustrates how a trading order made by a trader is executed by a retail forex broker. First, the trading platform on which the trader made the order sends the order information (e.g., currency pair, lot size, long/short, order type, etc) to the broker's server over the network (typically via TCP/IP [7][8]). If the broker routes clients' orders through its dealing desk (DD) and trade against them, it is called a *DD broker* or a *market maker*, for it literally "makes the market" for traders – the broker buys from its clients when they want to sell, and sells to them when they want to buy. Losing trades of clients are counter-traded within the dealing desk's system and become the broker's profit. Winning trades of clients on the other hand are usually hedged in the interbank market to offset the broker's risks.

If the broker processes clients' orders without passing through the dealing desk, it is called a *non dealing desk (NDD) broker*. Such brokers offer clients direct access to the interbank market or to an electronic communication network (ECN) along with a straight through processing (STP) bridge [9], providing clients with a transparent and low-latency trading environment. However, some NDD brokers may have what is called the "last look" provision [10], meaning they check the contents of traders' orders before

they send them to the market. By having this functionality the broker is, in many cases, able to negotiate more favorable terms from its liquidity provider because the chances of the liquidity provider losing are significantly reduced by inspecting orders before they are actually executed in the market.

3. Forex Algorithmic Trading Systems

3-1. Development of Trading Algorithms

Developers who try to make their own algoritmic trading systems for the retail forex market are required to create at least the following software components: (1) a backtester module that allows them to backtest trading algorithms in numerous conditions (e.g., currency pairs, time frames, time periods); (2) a renderer module that displays on the screen charts of currency pairs along with various types of indicators (e.g., moving avergaes, oscillators, etc); (3) a network module that receives live data feeds and sends trading orders to the server using the broker's API; and (4) a trading algorithm which was thoroughly-tested and is most likely to generate profits in live trading. Developing such a profitable algorithm is highly sophisticated art and science, and requires knowledge and skills not only in computer programming and finance but also in various quantitative methods (e.g., mathematics, statistics, econometrics, etc).

Consequently, it requires a substantial amount of study and work for developers to build an algorithmic trading system from scratch. As a result, many of them choose to focus solely on creating their own trading algorithm using existing algorithmic trading systems that are available in the marketplace, some of which are offered at no extra cost for those who open an account with a broker. The MetaTrader 4 (MT4) [4] is by far the most popular algorithmic trading system among developers of forex trading algorithms. we will discuss the MT4 platform in detail in Chapter 3-4.

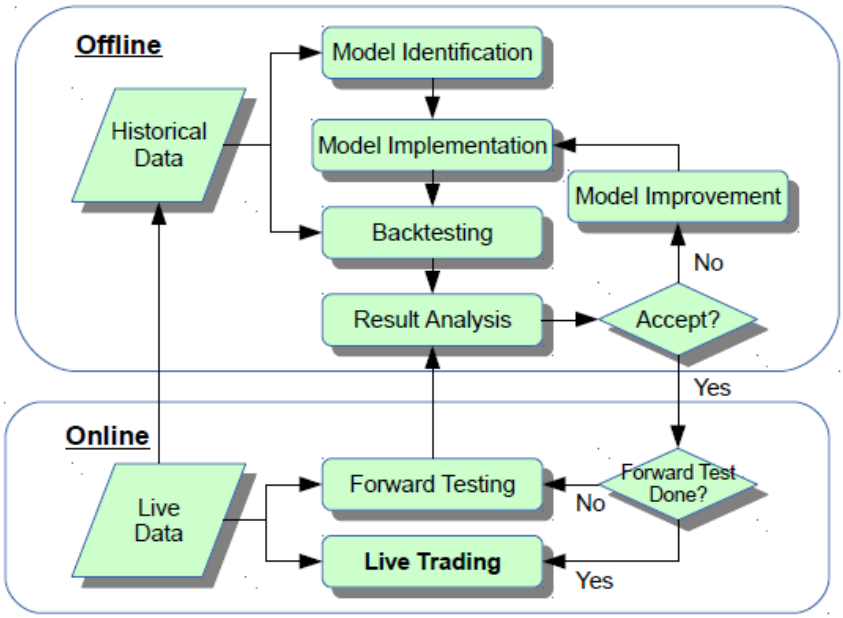

Figure 4. *Flowchart of the development of a trading algorithm*

Figure 4 illustrates how a forex trading algorithm is typically developed (this is also the case for developing trading algorithms for other assets, such as stocks). There are mainly two groups of stages in this process: offline stages (backtesting process) and

online stages (forward testing process).

Model Identificatin

The first thing that developers of trading algorithms need to do is to analyze historical market data, make a hypothesis, and build a model. Although this process often involves mathematical analysis of the historical data, developers are not necessarily required to build a mathematical model for forecasting. What they are required to do is to come up with a trading strategy whose rules of market entry/exit are clearly defined and detailed enough to actually be coded as an algorithm. Trading strategies are in many cases devised using indicators used for technical analysis.

Model Implementation

The second stage is to implement the model as an algorithm which computers are able to execute. This means for developers to write code of the trading algorithm in a programming language or a custom scripting language employed by an existing trading system. Although the latter often comes with built-in functions and other useful features for trading, it has limitations in many ways. Some developers therefore still prefer to write their own code in order to get maximum flexibility.

Backtesting

The third stage is to backtest the trading algorithm using historcial market data. This stage is especially important as it is indispensable for developers to backtest their trading algorithm so as not only to estimate the profitability of the algorithm but also to validate its implementation. Unexpected behavior and operation of the algorithm during backtesting must be scrutinized, since it is highly likely that the algorithm is not implemented properly in such a case. we will disscuss backtesting in detail in Chapter 3-2.

Result Analysis

The fourth stage is to analyze the result obtained from backtests of the trading algorithm and to determine the reliability and suitability thereof. To this end various types of performance measures are calculated dynamically during backtesting, and developers can subsequently estimate the profitability as well as the risk of their algorithm based on those measures (cf., Chapter 3-3).

Forward Testing

If the results of backtests are acceptable, developers usually proceed to forward testing, meaning they run their trading algorithm by feeding live market data in real time so as to see if it still generates profits as it did in backtesting. Naturally, forward testing takes much longer than backtesting. Some developers therefore may prefer to substitute forward tests with out-of-sample backtests (cf., Chapter 3-2).

Developers are finally able to use their algorithm in live trading after it produced an acceptable result in forward testing. However, it is always recommended for developers to start live trading using their algorithm only with a limited amount of money in order to minimize the initial risk associated with the live trading using the algorithm. This is also because of the fact that, if the algorithm is truly profitable in a constant and stable manner, developers are able to increase their money in an exponential way with compound interests. Consequently, developers do not have to take an excessive risk from the beginning of their live trading.

3-2. Backtesting

Backtesting, also called *historical testing* or *historical simulation*, is an indispensable process for developers to devise a profitable trading algorithm, because it provides developers with a simulation result of the performance that is likely to have been achieved had the algorithm been actually used in live trading on prior time periods. In other words, developers are able to estimate the expected return and risk of their algorithm by doing a simulation against existing historical data, and therefore they are able to determine the historical profitability of their algorithm and to reject potentially unprofitable or even harmful models.

Backtesting is extremely useful since it provides information that cannot be attained when algorithms are tested on synthetic data. It also provides developers with a sense of security, which is a significant psychological factor when they are actually participating in the market and risking real money.

In backtesting the initial part of historical data on which an algorithm is tested and optimized is specifically referred to as the *in-sample data*, while it is conventional among developers to reserve a portion of the historical data as the *out-of-sample data* (Figure 5) [11]. By having this setup of historical data, developers can substitute forward testing with out-of-sample backtesting because the algorithm has never been tested on the out-of-sample data during its development, yet the out-of-sample data is based on the real market, thus effectively acting as forward testing. This allows developers to avoid spending a long time on forward testing. In order for out-of-sample backtesting to be valid, it is crucial for developers to strictly use only the in-sample data for the backtests and optimizations of their algorithm during the development.

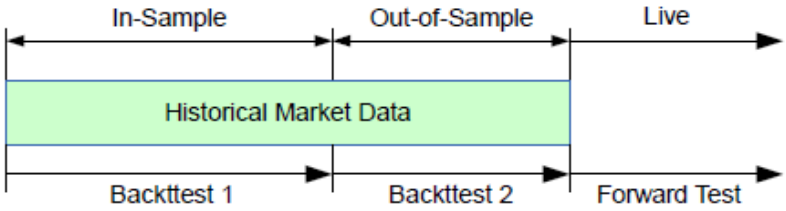

Figure 5. *In-sample and out-of-sample data*

It is important for trading algorithm developers to realize that backtesting does not guarantee by any means that historically profitable algorithms will maintain their profitability in the future. In fact, there are several limitations to backtesting. Firstly, it is not necessarily easy for developers to obtain high-quality historical market data with which they can sufficiently reconstruct past market conditions. This is one of the primary reasons why backtesting had historically been performed mostly by large financial institutions. Secondly, due to the static nature of historical data, it is very difficult to take into account the market impact of an algorithm in backtesting (as well as liquidity). While in many cases this is not an issue for most retail traders with small volumes, it is not negligible for large institutions and hedge funds. Thirdly, backtesting is limited by potential overfitting – it is often possible for developers to make an algorithm perform well in backtests by overly optimizing it to specific historical data. While the algorithm would have worked well in the time period in which it was optimized, the result achieved is highly dependent on the price movements during that period in the data, hence there is no guarantee that it will work well in the future. Finally, it is also difficult in backtesting to fully take into account implementation shortfall (a.k.a., "slippage"), which is the difference between the initial price at which a trading decision was made and the final price at which the order was actually executed

(including broker fees, taxes, etc). Implementation shortfall is an execution cost, it can therefore be problematic when the algorithm is high-frequency.

3-3. Measures of Algorithm Performance

It is indispensable for developers to collect and analyze objective information regarding the performance of their trading algorithm in order to assess its profitability and risk. There are various types of performance measures for trading algorithms, the most typical ones of which include: rate of return, rate of winning, average profit, average loss, profit factor, payoff ratio, Sharpe Ratio [12], and maximum drawdown.

The *average profit* per winning trade is equal to the sum of all profits (i.e., *gross profit*) divided by the number of winning trades. It is useful for getting an idea of how much one could expect to gain from a winning trade.

$$\overline{W} = \frac{1}{N_x} \sum_{i=1}^{N_x} W_i$$

Similarly, the *average loss* per losing trade is equal to the sum of all losses (i.e., *gross loss*) divided by the number of losing trades. It is useful for getting an idea of how much one could expect to lose from a losing trade.

$$\overline{L} = \frac{1}{N_L} \sum_{i=1}^{N_L} L_i$$

The *profit factor* is defined as the ratio of the gross profit to the gross loss:

$$P_f = \frac{\sum_{i=1}^{N_W} W_i}{\sum_{i=1}^{N_L} L_i}$$

If the profit factor is greater than 1, it indicates that the backtested algorithm has more wins than losses in the historical market data, hence it would have been profitable in the time period, otherwise it would have incurred losses.

Profit factor is a simple yet useful measure of how efficient the profitability of an algorithm is. For example, let us assume that we have an algorithm denoted by A which has made $10,000 of gross profit and $5,000 of gross loss (therefore has a profit factor of 2), and another algorithm B which has made $5,000 of gross profit and $2,000 of gross loss (i.e., profit factor of 2.5). While the algorithm A has a final profit of $5,000 and the algorithm B has a final profit of $3,000, the algorithm B is actually superior in terms of profitability because if we had doubled the leverage ratio for our margin account we would have made $10,000 of gross profit and $4,000 of gross loss using the algorithm B, therefore obtaining a final profit of $6,000.

It is always crucial in backtesting for developers to make their algorithm achieve a profit factor at the very least greater than 1 since as long as the profit factor is less than 1 the probability that the algorithm will not generate profits in live trading is quite high. That being said, higher values of profit factor do not necessarily guarantee the algorithm's prospect in live trading as it may have been overfitted to the historical data.

The ratio of the average profit to the average loss is referred to as the *payoff ratio*:

$$P_r = \frac{\overline{W}}{\overline{L}}$$

This alone indicates how many losses on average can be compensated with one winning trade. For example, an algorithm that has a payoff ratio of 2 can averagely compensate for 2 losses with 1 winning trade.

The payoff ratio is not a direct indication of the algorithm's profitability; instead, it is highly related to the *rate of winning* (or *winning percentage*):

$$R_W = \frac{N_W}{N}$$

where N is the total number of trades that the algorithm has made during the backtesting period. The profit factor can subsequently be derived from the payoff ratio and the winning percentage:

$$P_f = \frac{R_W}{1 - R_W} P_r = \frac{N_w}{N - N_W} P_r = \frac{N_W}{N_L} \frac{\overline{W}}{\overline{L}}$$

If we solve the equation above for Rw, we get:

$$R_W = \frac{1}{1 + \frac{P_r}{P_f}} = \frac{P_f}{P_f + P_r}$$

This equation is known as the Profitability Rule [13].

Even if the payoff ratio is below 1, the algorithm can be profitable as long as the winning percentage is sufficiently high so that the winning trades compensate for the losing trades. In the same manner, even if the winning percentage is low, the algorithm can be profitable if the payoff ratio is sufficiently high. Table 1 shows this relationship between payoff ratio and winning percentage. The required winning percentages are computed using the Profitability Rule with a fixed profit factor equal to 2 as an example.

Payoff ratio	**Required R_W (%)**
10	16.67
5	28.57
2	50.00
1	66.67
0.5	80.00
0.2	90.91
0.1	95.24

Table 1. *Required rate of winning as a function of a payoff ratio (profit factor = 2)*

Now, if we denote by R_t the cumulative daily return up to period t, then the associated Sharpe Ratio [12] is given by:

$$S_r = \frac{mean(R_t)}{std(R_t)}$$

where $mean(R_t)$ and $std(R_t)$ are respectively the average and the standard deviation of the daily return calculated over the period. Here, the mean value represents the gains and the standard deviation the risk. Concordantly, the Sharpe Ratio represents the risk premium per unit of risk of a trading algorithm. It helps us determine if an increase in risk is appropriate for a higher return (often, high returns may come at a cost of excess risk). A Sharpe Ratio of 1 or greater is sufficient, 2 or greater is ideal, and 3 or greater is excellent.

3-4. MetaTrader 4

The MetaTrader 4 (MT4) [4] is an electronic trading platform specifically designed for retail forex margin trading. Since its release in 2005, MT4 has been licensed to a large number of retail forex brokers worldwide due to its extreme popularity among individual traders. It has become the de facto trading platform for the retail forex market because it allows users to write their own scripts for custom indicators, trading algorithms (which are referred to as *Expert Advisors* or *EAs* on the platform) and other types of add-ins. The proprietary scripting language used by MT4 is called the MetaQuotes Language 4 (MQL4) [14], which is similar to the C programming language. It comes with various built-in functions that are useful for creating custom indicators and algorithms. MT4 also supports custom DLLs (dynamic-link libraries), allowing developers to write their own algorithm in C++ [5] or other languages that support DLL compilation, thereby providing developers with more flexibilities.

Although MT4 has a built-in algorithm backtesting system, it has some severe limitations and thus it has never been an ideal backtesting system for forex trading algorithms, at least in our opinion. Firstly, historical market data used by MT4 contains only bid prices. The platform therefore executes backtests only with a fixed spread (it uses the current live spread at the time a backtest is launched). Obviously such situations never exist in the real market, hence this is a serious issue as developers cannot take into account all the dynamic changes in spreads that constantly occur in the real market. Not only does it create overly optimistic results for spread-sensitive algorithms when the live spread is below average at the time the backtest is launched, but also it lacks in reproducibility because test results can vary significantly depending on the spread size. Secondly, historical data for the platform is not consisted of raw tick data, it instead is optimized for candlestick charts. The platform generates tick data for backtests from historical data by randomizing open, high, low and close prices. This is an issue for developers who want to reconstruct the market for spread-sensitive or high-frequency algorithms as much as possible. Finally, backtesting on the platform takes a lot of time since the MQL4 scripting language is not native (about 17 times slower than C++ [15]) and the platform does not support multithreading that utilizes multicore CPUs, which most of today's computers are equipped with.

Our backtesting system on the other hand solves all the limitations that have been mentioned above: our system uses full tick-by-tick historical data with dynamically changing spreads, and of course is multithreaded so as to take the best advantage of modern multicore CPUs.

4. System Design & Architecture

4-1. C++ Programming Language

Execution efficiency is the most critical aspect of a backtesting system inasmuch as backtesting is a time-consuming process. Minimizing the execution time of backtesting therefore results in a significant increase in the productivity of trading algorithm developers since it is indispensable for them to backtest their algorithm repeatedly during its development in order to make it truly profitable. For this purpose, we naturally selected the C++ programming language [5] to develop our backtsting system, because its execution efficiency is, due to its native code generation, unmatched with other programming languages other than the C language, of which C++ is a superset. Nowadays, most performance-critical software is written in C++ (e.g., operating systems, virtual machines, 3D games and other real-time systems). In fact, many investment banks and hedge funds use C++ to build their proprietary algorithmic trading systems.

C++ is sufficiently high-level to write complex, large-scale programs, while simultaneously it is sufficiently low-level to write efficient code (e.g., manual memory management, bitwise operations, SIMD intrinsic functions, and even inline assembly). Consequently, it is one of the most powerful, flexible and efficient programming languages, and is also highly efficient on hardware with limited computing resources (e.g., embedded systems, mobile devices) because efficiency is not just running faster or running bigger programs but also running using less resources – these are the two sides of the same coin.

Like every programming language, however, C++ has its own disadvantages, which include: (1) C++ does not offer developers a higher code productivity because it is an extremely complex language and thus learning and fully understanding the language require a tremendous effort; (2) C++ by default does not have any automatic memory management mechanism such as garbage collector, hence programmers must be prudent about memory leaks and memory protection errors; (3) the C++ standard library is relatively limited compared to other languages; in many cases programmers therefore need to adapt an external library, such as Boost [16], in order to obtain further functionality and productivity; (4) C++ is generally not suited for developing safety-critical software because of its complexity and compiler-dependent nature (although it is possible to build safety-critical systems in C++ securely by imposing strict coding standards [17]); and finally (5) C++ code generally takes a lot of time to be compiled, and every time a change is made in the code, it needs to be re-compiled since new native code must be generated.

With all of that being said, the new C++ standard introduced in 2011 (a.k.a., C++11 [18]) has greatly enhanced the language with new features and additional libraries; most notably smart pointers and multithreading facilities. We, however, did not use C++11 for our backtesting system mainly because of its immature compiler support at the time of the development of our system. we instead used the C++ standard library and Boost.

We also placed importance on portability so as to make our system able to run on multiple platforms (primarily Windows and Linux). To this end we wrote our C++ code in accordance with the language standard as much as possible and also used crossplatform libraries – namely, Qt [19] for GUI, OpenGL [20] for graphics, and TA-Lib [21] for technical analysis. we consider that Linux support is particularly important as many investment banks and hedge funds develop their proprietary algorithmic trading system on UNIX-like operating systems.

4-2. Asynchronous Function Parallel Model

As Sutter [22] predicted back in 2005, concurrent programming has become a new trend of programming paradigm today as single-threaded programs no longer get any significant increase in performance because the CPU's clock speed has hit a wall due to physical limitations (i.e., heat, power consumption, and current leakage). Yet, the transistor density of the CPU is still growing, and as a result we are having more processor cores instead of faster clock speeds. Consequently, it has become essential for programmers today to adapt concurrent programming in order to fully exploit the continuing gains in processor throughput.

Naturally, our backtesting system is multithreaded to take advantage of multicore CPUs. However, the biggest problem in parallelizing a backtester is the fact that backtesting is inherently a serial process because historical data used in backtesting is strictly sequential (i.e., time series), it therefore needs to be evaluated serially, which makes it impossible for developers to exploit data parallelism. Consequently, we instead exploited function parallelism by adapting the asynchronous function parallel model [23].

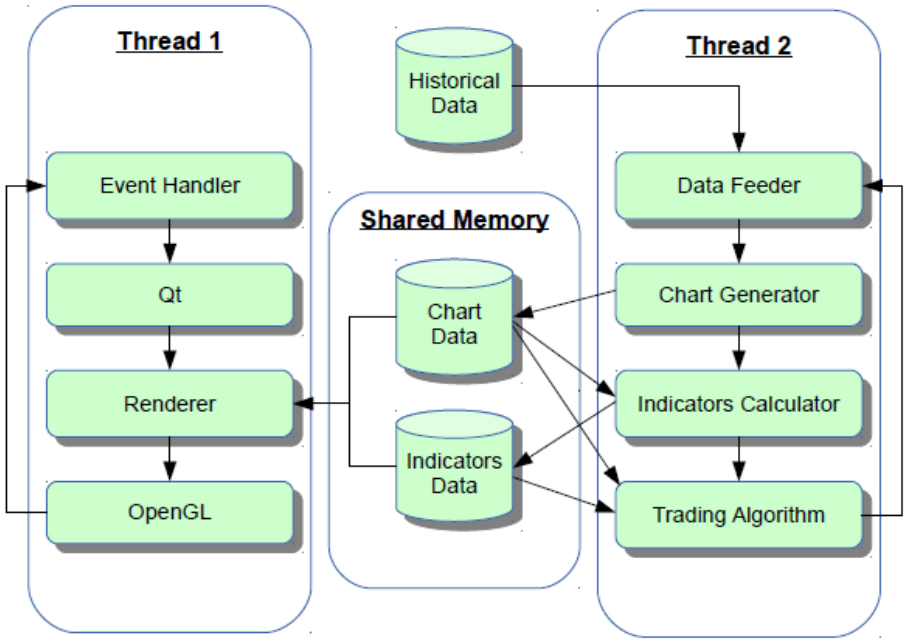

Figure 6. *Asynchronous function parallel model of our backtesting system*

Figure 6 illustrates the architecture of our backtesting system. Here, Thread 1 (main thread) is dedicated to the GUI loop, and Thread 2 (child thread) is dedicated to the backtester loop, both of which threads run in parallel independently without any synchronization, thereby minimizing the interactions between the two threads (which is ideal because frequent interactions between threads can often be problematic due to increased complexity).

For each step in Thread 2's backtester loop, the Data Feeder at the beginning of the loop reads the historical data one line at a time, retrieving the latest tick in the historical data, then the Chart Generator and the Indicators Calculator construct a chart and indicators respectively according to the tick data. The Chart Generator and the Indicators Calculator must be executed in that serial order because the latter requires the latest chart data to calculate indicators. Both the chart and indicators data are stored in shared memory, from which the Renderer in Thread 1 asynchronously reads the latest chart and indicators data and draws a chart on the screen. Each read/write operation

associated with the shared memory is done by locking a mutex so that only one thread at a time may access the data in the memory in order to avoid race conditions and data inconsistencies. (Nested locks of multiple mutexes are also carefully avoided in order to prevent deadlocks.) Finally, the Trading Algorithm is executed along with the latest tick, chart and indicators data.

These backtester modules as a whole constitute a sequential cohesion and data coupling (i.e., the output from one module is the input to another module).

4-3. Further Parallelization

Some trading algorithms use multiple time frame (MTF) charts and indicators for detecting entry/exit points in the market in a more precise manner. Backtests of such algorithms can be accelerated by having additional threads concurrently generating charts and indicators of different time frames.

While some other algorithms use combinations of a number of indicators whose calculations are often computationally expensive. Backtests of such algorithms can also be accelerated by assigning each additional thread a group of indicators that can be calculated independently in parallel. Note that some indicators are composites of other indicators, i.e., they depend on two or more other indicators; thus they need to be calculated serially according to their dependencies. It is also important to note that for the additional threading to be beneficial in this case, the granularity of indicators calculations assigned to each thread must be coarse. In other words, the amount of work per thread must be larger than the parallel overhead associated with the thread.

In both types of trading algorithms above, all the additional threads are synchronized before the Trading Algorithm gets executed (Figure 7):

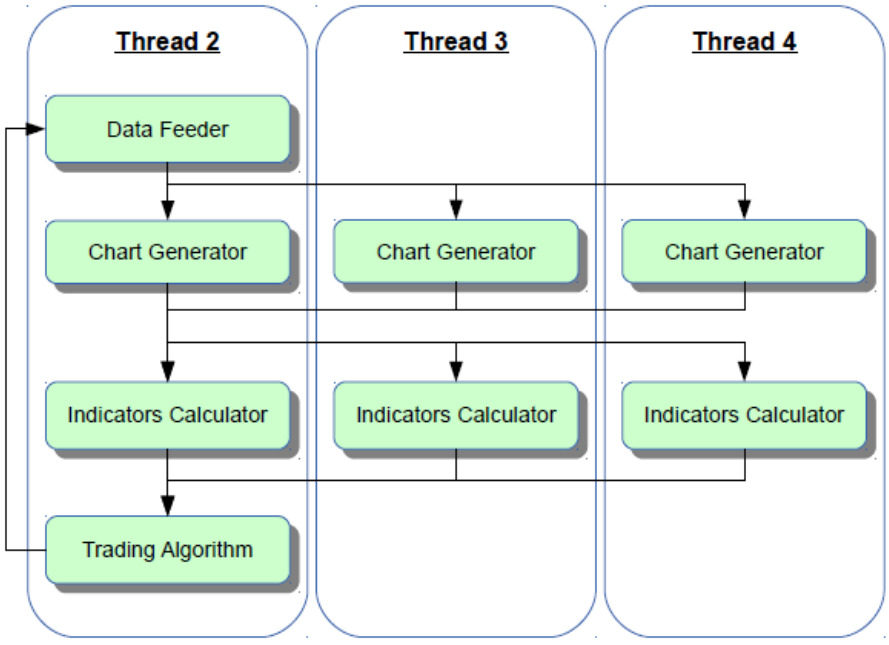

Figure 7. *Synchronous function parallel model for the backtester loop*

There are also some algorithms that refer to charts and indicators of multiple currency pairs at the same time. For such algorithms, the Data Feeder is assigned to each additional thread with different historical data, and subsequently charts and indicators are computed in parallel. In this model it is vital to synchronize the backtesting process based on the time recorded within all the historical data, because the execution speed of each thread is different.

As for backtesting multiple algorithms in parallel, we decided to run multiple instances of our system instead of having multiple threads executing multiple algorithms within the system. This allowed us to maintain the code complexity of the system as minimal as possible since the operating system would instead handle the additional thread creation and management. This is also useful when backtesting the same algorithm with different parameters, different time frames, different time periods, etc.

Figure 8. *Running multiple instances of our system in parallel*

5. Trading Algorithm Optimization

5-1. Walk-Forward Analysis

As already discussed in Chapter 3-2, trading algorithm developers usually split their historical market data in two sets: the in-sample data for optimization and the out-of-sample data for validation. However, this simple binary separation of data is not always the best, especially for the testing of low-frequency algorithms (e.g., the trend-following), because the out-of-sample data period is usually too short for such algorithms to make a sufficient amount of trades with which a statistically significant test can be conducted.

Pardo [24] devised a better approach called walk-forward analysis, in which developers still optimize their algorithm on the in-sample data and validate it on the out-of-sample data, but they repeat the process by forwarding the data period, i.e., after each out-of-sample test the in-sample data period is shifted forward by the same amount of data period covered by the out-of-sample test.

Let us illustrate this method with an example. Suppose we have 12 months of historical data (from January to December, 2013) and our algorithm requires at least 3 months of data for sufficient testing and optimization, and 1 month of data for proper validation. We start the development and optimization of our algorithm using only the first 3 months of data as the in-sample data (January–March), and then we validate the algorithm on the 1-month out-of-sample data (April), after which we record the optimum parameters of the algorithm in the data period. Subsequently, we shift the data

period forward by 1 month. As a result we now have the new in-sample data (February-April) and out-of-sample data (May). We proceed to optimize the algorithm on this new in-sample data using the optimal parameters that we found in the previous step, validate it on the new out-of-sample data, find and record new optimal parameters, and shift the data period forward again. We repeat this process until we reach the end of our historical data (December). When we reach the end, we revert to the first month and test the algorithm for the entire period recurrently by switching between parameters that were found optimal during all the earlier steps.

Apropos, backtesting systems are of no use without trading algorithms to test with. Therefore, not only did we develop the backtsting system but we also simultaneously developed our own trading algorithm and integrated it in the backtesting system as a black-box. Our trading algorithm is intended to trade the EUR/GBP currency pair, although it is possible to trade other currency pairs. Table 2 shows the setup of the EUR/GBP historical data for the walk-forward analysis of our trading algorithm.

Walk #	In-sample data period	Out-of-sample data period
1	06/01/'13 – 28/06/'13 (In1)	30/06/'13 – 30/08/'13 (Out1)
2	03/03/'13 – 30/08/'13 (In2)	01/09/'13 – 01/11/'13 (Out2)
3	05/05/'13 – 01/11/'13 (In3)	03/11/'13 – 03/01/'14 (Out3)
4	06/01/'13 – 03/01/'14 (In4)	N/A

Table 2. *Setup of the historical data for the walk-forward analysis*

In Walk #1, we initially optimize our algorithm by strictly using the 6-month in-sample data (In1) and subsequently validate it on the 2-month out-of-sample data (Out1), after which the in-sample data period is shifted forward by the period covered by the out-of-sample test (i.e., 2 months). we repeat this process up to Walk #3 and finally optimize the algorithm on the 12 months of data in Walk #4 using the information obtained from the previous steps.

5-2. Results

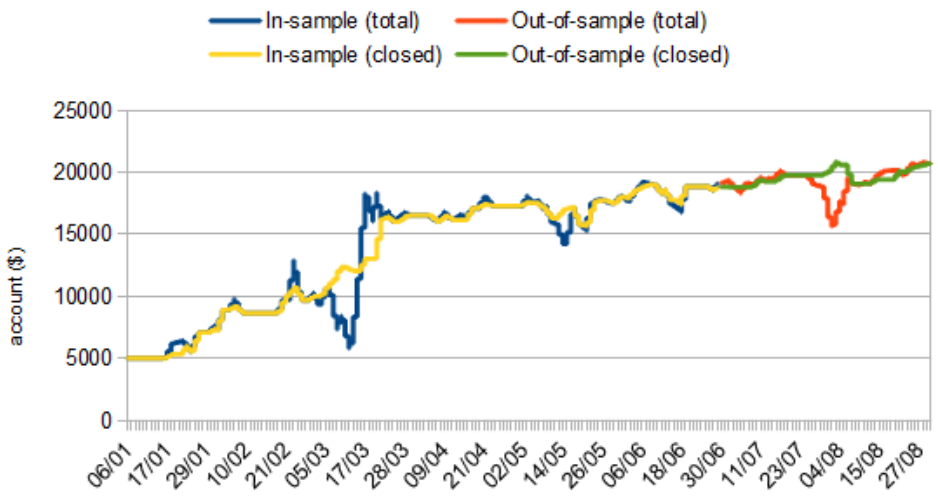

Figure 9. *Equity curves of our trading algorithm in Walk #1*

Figure 9 shows the equity curve generated in Walk #1 by backtesting our trading algorithm which we initially optimized for the in-sample data (In1)*. Although the algorithm experienced a drawdown during the out-of-sample backtest (Out1), it was still profitable. we proceeded to optimize the algorithm to the in-sample data in Walk #2 (In2), which gave us an astonishing result as shown in Figure 10 below.

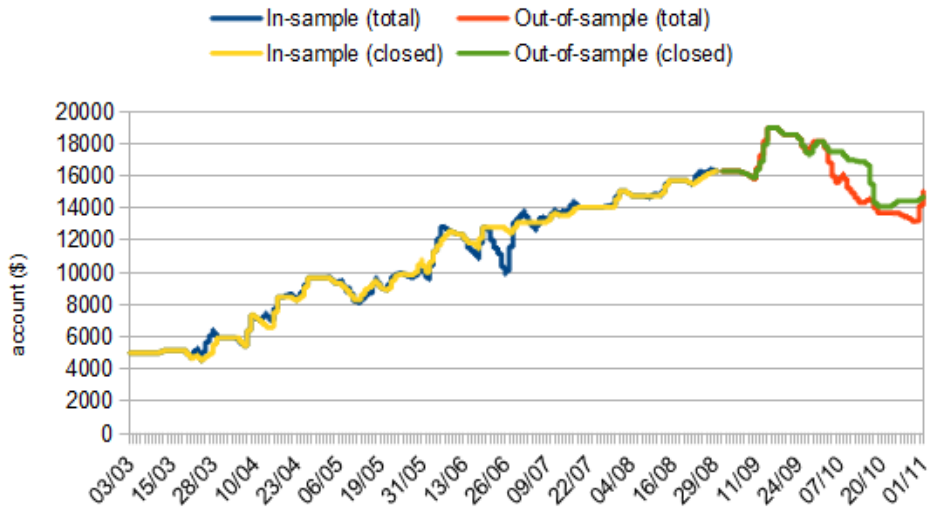

Figure 10. *Equity curves of our trading algorithm in Walk #2*

However, the subsequent out-of-sample backtest (Out2) suggested that the algorithm might be overfitted to the in-sample data (In2), since, as it can clearly be seen in Figure 10, the constant profitability of the algorithm during the in-sample testing ceased to continue on the out-of-sample data (but still the algorithm made a new high during the out-of-sample test).

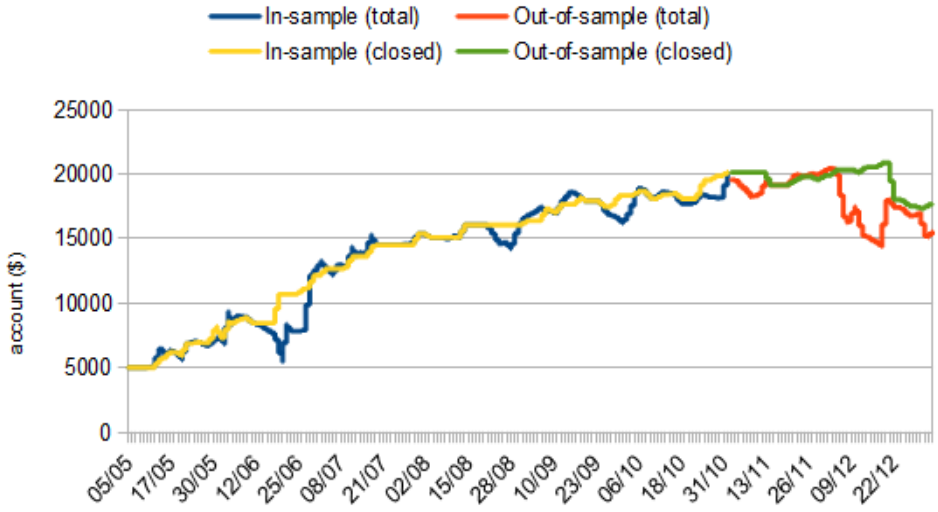

Figure 11. *Equity curves of our trading algorithm in Walk #3*

We further optimized the algorithm to the new in-sample data (In3); however, the new out-of-sample testing (Out3) also showed a sign of over-optimization with a decline in the account value.

---

*Video available at: https://www.youtube.com/watch?v=VtPTsXnxzT4

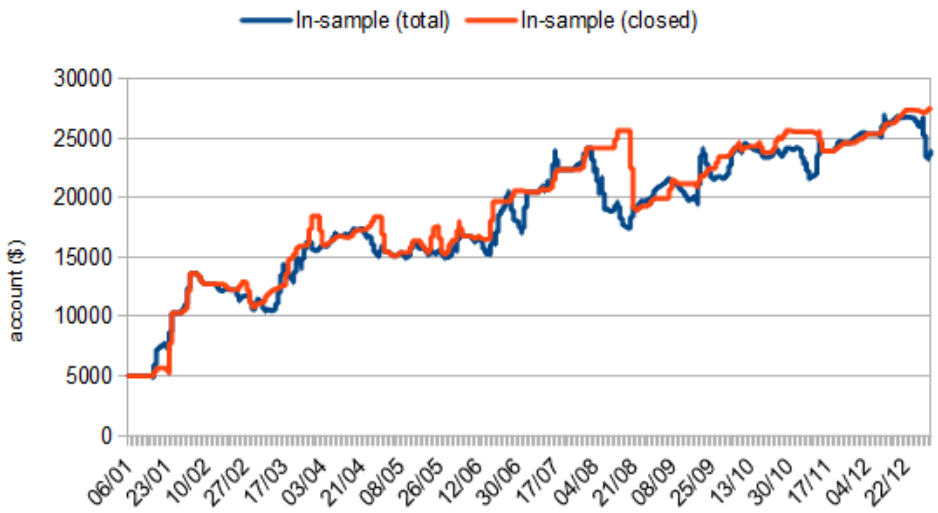

Figure 12. *Equity curves of our trading algorithm in Walk #4*

After analyzing the results of all the previous steps, we finally achieved 450% of return per annum in Walk #4 (Figure 12), with the winning percentage = 58.8%, profit facor = 1.74, average profit = $359, average loss = $288, maximum drawdown = $6,704, and Sharpe Ratio = 2.45.

Now that we have finished the walk-forward analysis of our trading algorithm, we can start porting it over to the MetaTrader 4 or other electronic trading platform and forward test it on a demo account with a broker. Once we are satisfied with the result, we may go live with it and risk our own real money.

6. Conclusion

The main goal of this research was to design and build a fast, flexible, and accurate backtesting system of trading algorithms for the foreign exchange market. Great importance was placed upon execution efficiency since backtesting is a time-consuming process as it is necessary for trading algorithm developers to backtest their algorithm recurrently with different conditions in order to make it profitable. For this reason we wrote the system in the C++ programming language and exploited function parallelism via multithreading so as to take the best advantage of modern multicore CPUs.

We also designed and implemented a trading algorithm for the EUR/GBP currency pair on the system, and analyzed its historical performance via walk-forwad analysis, according to which the algorithm is highly profitable at least in the data period used.

7. Future Work

As previously shown in Chapter 4-3, running our backtesting system in multiple instances allows users to backtest a trading algorithm with various conditions in parallel, helping them find an optimum set of parameters heuristically. This heuristic process of finding optimum parameters can be efficiently automated with genetic algorithms (GAs).

In order to reduce the time taken up by this finding process using GAs, it will be crucial to increase the degree of parallelism – that is, to increase the number of instances of the system (i.e., individuals) running in parallel. To this end, we can modify the

system and make it run on a computer cluster or alternatively on the GPU by using a GPGPU (general-purpose computing on GPUs) API, such as OpenCL [25].

Another possible modification for the backtesting system is to allow fast prototyping of trading algorithms using Runtime-Compiled C++ [26]. As already discussed in Chapter 4-1, it takes a long time to compile C++ code, and every time a change is made in the code, it needs to be re-compiled for the new native code generation. Using the Runtime-Compiled C++ allows us to run the modified code without re-compillation, thereby significantly increase the productivity of trading algorithm developers.

References

[1] Barry Johnson, *"Algorithmic Trading and DMA: An Introduction to Direct Access Trading Strategies,"* 4Myeloma Press, 2010.

[2] Andrei Kirilenko, Albert Kyle, Mehrdad Samadi, and Tugkan Tuzun, *"The Flash Crash: The Impact of High Frequency Trading on an Electronic Market,"* SSRN, 2010.

[3] FSOC, *"2012 Annual Report,"* US Department of the Treasury, http://www.treasury.gov/initiatives/fsoc/studies-reports/Pages/2012-Annual-Report.aspx, 2012.

[4] MetaTrader 4, http://www.metatrader4.com/.

[5] ISO/IEC 14882, *"Programming Languages – C++ 2nd Edition,"* 2003.

[6] NinjaTrader, http://www.ninjatrader.com/.

[7] Jon Postel, *"Internet Protocol (RFC 760),"* ISI, University of Southern California, http://www.rfc-editor.org/rfc/rfc760.txt, 1980.

[8] Jon Postel, *"Transmission Control Protocol (RFC 768),"* ISI, University of Southern California, http://www.rfc-editor.org/rfc/rfc761.txt, 1980.

[9] Jarrad Hee, Yining Chen, and Wayne Huang, *"Straight Through Processing Technology in Global Financial Market: Readiness Assessment and Implementation,"* Journal of Global Information Management, 11(2), 2003.

[10] Joey Horowitz, *"High Frequency Trading Challenges in Foreign Exchange,"* http://www.trdpnt.com/high-frequency-trading-challenges- foreign-exchange/, 2013.

[11] Jean Folger, *"Backtesting and Forward Testing: The Importance of Correlation,"* http://www.investopedia.com/articles/trading/10/backtesting-walkforward-important-correlation.asp, 2010.

[12] William Sharpe, *"The Sharpe Ratio,"* The Journal of Portfolio Management, 21(1), 1994.

[13] Michael Harris, *"The Profitability Rule,"* Stocks & Commodities, September Issue, 2002.

[14] MQL4, http://www.mql4.com/.

[15] MetaQuotes Software, *"Automated Trading with MetaTrader 4,"* http://www.metaquotes.net/en/metatrader4/automated_trading.

[16] Boost, http://www.boost.org/.

[17] Günter Obiltschnig, *"C++ for Safety-Critical Systems,"* http://obiltschnig.com/articles.html, 2009.

[18] ISO/IEC 14882:2011, *"Programming Languages – C++,"* 2011.

[19] Qt, https://qt-project.org/.

[20] OpenGL, http://www.opengl.org/.

[21] TA-Lib, http://ta-lib.org/.

[22] Herb Sutter, *"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,"* Dr. Dobb's Journal, 30(3), 2005.
[23] Ville Mönkkönen, *"Multithreaded Game Engine Architectures,"* 2006.
[24] Robert Pardo, *"Design, Testing, and Optimization of Trading Systems,"* Wiley Trader's Exchange, 1992.
[25] OpenCL, http://www.khronos.org/opencl/.
[26] Runtime-Compiled C++, http://runtimecompiledcplusplus.blogspot.com/.