Custom Framework for Run-time Trading Strategies

Ingrid Funie, Liucheng Guo, Xinyu Niu, Wayne Luk, and Mark Salmon

Imperial College London

Abstract. A trading strategy is generally optimised for a given market regime. If it takes too long to switch from one trading strategy to another, then a sub-optimal trading strategy may be adopted. This paper proposes the first FPGA-based framework which supports multiple trend-following trading strategies to obtain accurate market characterisation for various financial market regimes. The framework contains a trading strategy kernel library covering a number of well-known trend-following strategies, such as "triple moving average". Three types of design are targeted: a static reconfiguration trading strategy (SRTS), a full reconfiguration trading strategy (FRTS), and a partial reconfiguration trading strategy (PRTS). Our approach is evaluated using both synthetic and historical market data. Compared to a fully optimised CPU implementation, the SRTS design achieves 11 times speedup, the FRTS design achieves 2 times speedup, while the PRTS design achieves 7 times speedup. The FRTS and PRTS designs also reduce the amount of resources used on chip by 29% and 15% respectively, when compared to the SRTS design.

1 Introduction

In finance, a *trading strategy* is a fixed plan that is designed to achieve a profitable return by buying or selling stock on certain markets. Numerous trading strategies are employed in financial markets with many outcomes in mind - the most common being the identification of market trends. Understanding the different market characteristics is a first step towards being able to identify and measure them. This, in turn should link trend-following performance to the state of these market characteristics. Finally, this might be a step towards devising a way for a trend-following strategy to adapt to these changing market regimes.

Nowadays, with the advance of hardware acceleration devices such as field programmable gate arrays (FPGAs), it is possible to attain high component density and low power consumption, while achieving minimal latency [1]. Most of the existing solutions allow reconfiguration between different computations, but do not take advantage of their Partial Reconfigurability (PR): the possibility to reconfigure the device during the same computation. When using PR the application is represented as a sequence of operations that do not need to (or cannot) overlap their execution: each operation corresponds to a distinct configuration that can be loaded into the FPGA at run-time. Many of the hardware acceleration solutions use full-reconfiguration (FR) to switch at run-time between multiple FPGA designs. Full-reconfiguration based implementations have a small advantage when the data to be processed needs the FPGA on-board memory, as after each reconfiguration the data must to be sent again to the

FPGA, resulting in increased reconfiguration overhead time [2]. FR is however simpler since the complete chip is reconfigured, while it can be tedious to implement PR since much low-level detail is involved. Finally, it is also possible to use one or more multiplexers to switch between different circuits at run time; we call this Static Reconfiguration (SR). Clearly SR designs require the largest area but the least reconfiguration time compared to PR and FR designs.

Our study aims to provide the first generic framework for run-time customisation of trading strategies, with the following *contributions*:

1. Novel designs involving static reconfiguration, partial reconfiguration and full reconfiguration for multiple trading strategies.
2. Library of optimised trend-following trading strategies including fully-pipelined designs for Double Moving Average Crossover (TS1), Triple Moving Average Crossover (TS2), Price Rate of Change (TS3), and Bollinger Bands (TS4).
3. Demonstration of the proposed framework on both synthetic and historical market data points, showing a speedup of up to 11 times, 7 times and 2 times respectively for the SR, PR and FR designs when compared to an optimised multi-core CPU implementation.

2 Background and Related Work

The Foreign Exchange Market (FX) is tradable 24h/day excluding weekends, which makes it the largest asset class in the world suitable for high-frequency trading (HFT). FX market's dynamics make it change in regimes that move fast or slow, thus requiring the trading strategy to adapt to reflect the regime. HFT involves complex tools and algorithms to move in and out of certain positions in seconds or even milliseconds [3]. The following presents three well-known trend-following strategies which are included in our trading strategy kernel library.

1) *Exponential Moving Average (EMA)*: A *moving average* (MA) is a sequence of arithmetic means taken over a fixed interval which is then moved along consecutive data points. The regular time scale we are using could be interpreted either as fixed units of clock time units or fixed units of market time defined in some way for the strategy. An exponential moving average (EMA) is similar to a simple moving average, except that more weight is given to the latest data, using the following formula:

$$EMA_x = (\alpha * s_x) + (1 - \alpha) * EMA_{x-1} \tag{1}$$

where EMA_x is taken for each value $(x + n) < i < (x + k - n)$, s is the data set and α stands for the weight $\alpha = \frac{2}{movingAverageLength+1}$. In finance, it is often used to detect trends in price, in particular by comparing two simple moving averages: one over a long window and one over a short window [4]. Previous work has shown how to create a fully-pipelined simple 3-point moving average kernel [5]. However, this method has not been evaluated with market data [6] and it only provides small window moving averages, compared to what real-world applications need (e.g. 200-point moving average to predict financial trends [4]).

2) *Price Rate of Change*: The price rate of change (PROC) is a trading strategy that measures the percentage change between the most recent price and the price "n" periods in the past, using the following formula:

$$\frac{p_{now} - p_n}{p_{today-n}} * 100 \tag{2}$$

where p_{now} stands for the value of the closing price now and p_n represents the closing price value "n" periods ago. It is used by traders to confirm price movements, detect divergences, and determine potential overbought/oversold areas.

3) *Bollinger Bands*: A Bollinger Band (BB) is a band plotted two standard deviations away from a simple moving average. Because standard deviation is a measure of volatility, BBs adjust themselves to the market conditions, as follows:

$$SMA_X \pm (STD_X * 2) \tag{3}$$

where SMA_X stands for simple moving average for the past X closing prices (we use EMA instead of SMA), and STD_X represents the standard deviation of the past X closing prices. The closer the prices move to the upper band, the more overbought the market, and the reverse is true for oversold market identification.

In our study, we employ a BUY trading strategy. Taking the example of a GBP/USD FX trading market, whenever we choose to BUY 50 units of GBP, we will take the reverse position [3]. We compute our *returns* (which represent the profit in $\$M$) on the CPU, after the FPGA provides the final trading position decision, using the following formula:

$$R_c = (-d_t * p_t^A + d_t * p_t^B) - tc_t \tag{4}$$

where d_t takes the value 50 in our case. p_t^A and p_t^B represent our two currency prices, GBP and USD at time t, while tc_t stands for transaction costs. Transactions costs are neglected for our simulated synthetic market data, however they are included within our historical tick data.

3 Framework For Reconfigurable Trading Strategies

We develop a framework to support different combinations of trading strategy kernels. Our framework includes: (1) a static architecture which makes use of multiplexers to switch strategy kernels. (2) a full-reconfiguration architecture which changes the entire FPGA configuration and DRAM data. (3) a partial-reconfiguration architecture which only changes some parts of the FPGA configuration, while the remaining parts stay the same.

One challenge due to run-time full-reconfiguration (RTR) architectures comes from the overhead related to DRAM transfer. Because many commercial platforms use a soft memory controller on the FPGA fabric, reconfiguring the FPGA fabric results in the loss of DRAM contents, since the DRAM controller is no longer available to refresh DRAM. Thus before reconfiguration any intermediary data must be saved, and after reconfiguration any problem data must be loaded on-chip. Depending on the problem size, this may become a bottleneck. Depending on the total run time, the impact of RTR may be significant and one common way to address this is by increasing problem sizes (i.e. processing large amounts of data in our case) so the reconfiguration overhead becomes negligible compared to the savings in execution time. We choose to offer PR as an option as it removes the need to reload DRAM and even though it is harder to implement and uses more resources than FR because of the reconfiguration module, it offers a good trade-off between resource usage and performance.

Our framework can be configured as one of the three architectures whose design will be described throughout this section. It also provides a *trading strategy (TS) library* which includes, at the time of writing, some of the most well-known trading strategies (whose parameters can be changed by the user) which adopt a trend-following approach. Their respective performance will be analysed in Section 6. *Our framework is configured generically, such that any user can add a new trading strategy kernel to the library and use it freely.* We can also configure the area where a trading strategy will be placed on-chip as well as customise the way we choose to place the trading strategies on the FPGA: for example, we can choose to run all strategies in parallel so we get multiple feedback, or run just a number of them in parallel and switch between some others (according to a switch condition set by the user). The number of trading strategies is not fixed, it varies with the chosen architecture, how we choose to use it and how complex each of the strategies' design is, all being highly dependent on resource usage.

Static Reconfiguration (SR) Architecture. It contains all the strategy kernels and places them on the FPGA. The SRTS design can choose a group of the strategy kernels to run: for example in Figure 1, first the framework chooses TS_{x+1} to TS_n to run, and then it switches and chooses the group $TS_1 ... TS_x, TS_{m+1} ... TS_n$ to run. SRTS can run all the kernels at the same time, or use multiplexers to select some of the kernels. As it puts all the kernels on the FPGA, thus taking the most resources compared with other architectures. This architecture is the fastest because it uses multiplexers to do the switching within one cycle, although for more complex designs a pipelined multiplexer network can be used to reduce cycle time, at the expense of taking multiple cycles.

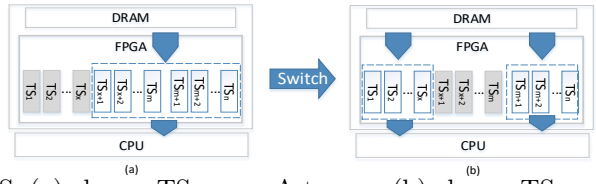


Fig. 1: SRTS: (a) choose TS group A to run. (b) choose TS group B to run

Full Reconfiguration (FR) Architecture. Figure 2 shows the overall FR architecture, which works as follows. (1) The data are collected on the CPU and are then transferred to the accelerator DRAM. (2) On each market data tick, the outcome of the evaluation (Buy, Switch or No Position to be taken) indicates what trading position the algorithm should take. (3) Every X number of cycles we check if the condition set by the user is met (i.e. in our case, the conditions stated in Section 5) and if it is a switch is performed (X corresponds to the market entry points from a particular trading window: e.g. 30-second trading window with a frequency of 1000 ticks per second and an FPGA clock rate of one tick/clock cycle, X = 30000). (4) Reload the market data onto accelerator DRAM. (5) Repeats steps (2) to (4) until there are no more data to evaluate. On the FPGA we used, we have to transfer the data to DRAM each time we switch between two FPGA designs. This is due to the limitations of the DRAM controller IP available for our device. If the controller had a user interface to

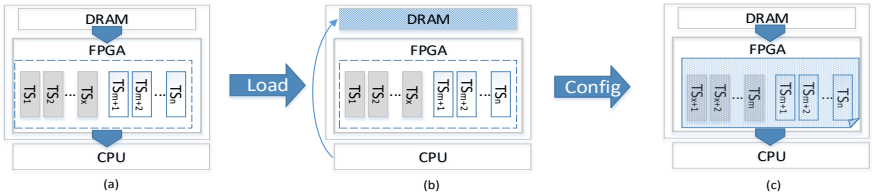


Fig. 2: FRTS: (a) Kernel Running, (b) Data Reload, (c) Kernels Reload

issue commands for self refresh mode on DRAM, this data reloading would not be necessary. Each of the reconfigurable blocks from our architecture can contain multiple trading strategies. The transferring operation, the unload and the load operations of the FPGA designs add further overload to our solution, increasing the switch time and thus making it less feasible from a time constraint point of view to switch between strategies as often as possible. On CPU we check all outputs from each FR module and compute the returns according to Formula 4.

The total *reconfiguration overhead* time for our FR design is:

$$O_f = r_t + d_t \tag{5}$$

where O_f is the reconfiguration overhead, r_t stands for reconfiguration time and d_t is the data transfer time. The total *data transfer* time is computed as follows:

$$d_t = data_{size} * (n_{bytes})/bandwidth \tag{6}$$

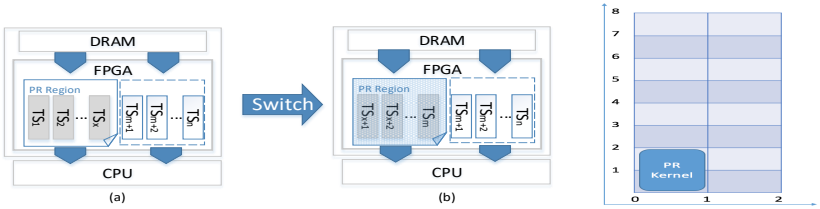
Partial Reconfiguration (PR) Architecture. We employ partial reconfiguration tools [7] to develop our PRTS, which can switch between the multiple trading strategies at run-time. Our PR architecture links a reconfigurable block (which can contain many trading strategies) to multiple static ones (that can be used as trend-following momentum filters or just as different trading strategies). Figure 3(i) shows the overall PR architecture, which works as follows: (1) The market data are simulated on the CPU and are then transferred to the accelerator DRAM. (2) TS_i kernel reads data from DRAM. (3) If the switch condition is met, then every X number of cycles we perform the switch to TS_{i+1} kernel (respectively TS_i kernel if previous state was TS_{i+1} kernel). (4) Repeats steps (2), (3) until there are no more data to evaluate.

The *PR configuration region* used for our design and implementation is ((0, 0), (1, 2)) and corresponds to the area split shown in Figure 3(ii). We choose this because we need to save area for data stored in DRAM - however the PR configuration area in our framework can be fully-customised by the user to accommodate its needs. The *common parts* of our design are the manager, the memory controller, the PCIe communication module and any other static kernel (e.g. TS3, TS4 in our case) that might be chosen to run in parallel with the partial-reconfiguration module. These static kernels stay fixed, and only the $TS_{n-1}..TS_n$ kernels switch at run-time (e.g. in our case TS1, TS2).

The total *reconfiguration overhead* time for our PR solution is given by:

$$O_p = r_t + t_e \tag{7}$$

where r_t stands for reconfiguration time and t_e stands for increased execution time due to reduced clock frequency. In our case, when we take the trading



(i) PRTS: (a) Kernel Running, (b) Kernel Switch (ii) PR Config. Region

Fig. 3: PR architecture and configuration region

decision and perform the returns using Formula 4 on CPU, we take into consideration our two different filters as follows: if one of our main trading strategies (i.e. TS1 - TMAC) suggests a BUY position should be taken - we check if one of our filters (i.e. TS3 - PROC) suggests the same - if not, we decide not to pursue with the position. It is important to note that this is just one particular approach in which this framework could be used.

4 Implementation Details

The implementation of the proposed designs depends heavily on the properties of the target system. The accelerator system we use is a Maxeler MPCX node. If we were to use a different architecture than the one based on Maxeler's infrastructure, then the implementation and its respective performance would change according to the new accelerator's specifications: e.g. the reconfiguration time could change using a different reconfiguration methodology on different boards, or the communication channel may change among different architectures. The system properties are summarised in Table 1 and it consists of a CPU node in 32nm transistor technology, and a DFE node with the FPGA in 40nm transistor technology. The two are connected via Infiniband through a Mellanox FDR Infiniband switch. The implementation of the architectures follows thoroughly their design as presented in Figure 3, Figure 2 and Figure 1.

Table 1: System Properties

CPU / CPU DRAM	Intel Xeon CPU X5650@2.67GHz / 48GB
FPGA / FPGA DRAM	Virtex6 SXT475 / 24 GB
CPU to FPGA BANDWIDTH	2 GB/s

The Virtex-6 SX475T FPGA used in this work has 16 clock regions: *we do not place PR regions in the central clock regions of the chip as this could reduce the impact on the routing process for memory controllers.*

Input/Output/Operation. The data-flow engine (DFE) is optimised for relatively large transfer bursts from on-board DRAM and maximum DRAM transfer efficiency is achieved when more bursts are read in a linear access pattern: both our designs enable this, as all market data are read from DRAM in a sequential fashion. We can read up to 1536 bits per clock cycle from DRAM. Since our market data variables are single precision floating point values (32 bits wide), we could read up to $1536/32 = 48$ different market variables from on-board DRAM without causing the designs to become memory bound. Since we try to simulate a continuously streaming environment, we use only one market variable per clock cycle, thus the default memory controller frequency is enough.

The returns are computed on the CPU considering every decision output from the FPGA. Because a good proportion of the market values will be reused for each trading strategy evaluated on every new market point entry, these will be stored in DRAM and only incur the transfer penalty over the slow interconnect between the CPU and FPGA once for both SRTS and PRTS. For FRTS we will however have to re-transfer the data every time we perform a switch.

CPU implementation. This is built using C++11 and parallelised using OpenMP. We compile the CPU implementation using Intel c++ with flags `-O3 -march=native -fopenmp` to enable general performance optimisations, architectural optimisations for the Intel XEON and the use of multithreading.

FPGA implementation. While both our run-time reconfiguration versions PRTS and FRTS are applicable to any number and combination of trading strategies, for the purpose of this paper we limit the trading strategies to the ones implemented in the provided library (see Section 5). The SRTS approach can contain as many trading strategies as we can fit on the available FPGA, thus being highly dependent on resource usage. We aim to set the basis of a generic framework which could include many more trading strategies in a lot of different combinations, depending on the user's trading interest, thus focusing on optimising the potential switch between trading strategies under different market conditions. Our proposed framework includes three different solutions: a PR, a FR approach as well as a SR one. The library designs included in our framework have fully-customisable parameters (e.g. different moving average lengths), thus allowing the potential users to be able to exploit the best financial returns.

5 Library of Trading Strategies

We propose to exploit the SR, PR and FR capability of FPGA technology by creating customisable designs which achieve the throughput rate of one data point per clock cycle. We implement the following trend-following strategies as an initial library for our framework:

1) *Exponential Moving Averages on FPGA.* The FPGA design of our EMA consists of a series of statements defining input and output streams and computations on streams, as follows: As we store all data elements in memory, we have a register which stores the sum and at each tick it shifts in new data and multiplies it to the present sum following Formula (1). We use the exponential moving average in the following two strategies [4]:

a) *Double Moving Averages Crossover Trading Strategy (DMAC).* It involves two MAs: one short and one long. We pick the most encountered case in practice for short-term market fluctuations, thus having the short and long MAs computed over a 25-point respectively 200-point trading window of closing prices. The strategy trades when the short MA crosses the long MA from above and below. In our system, DMAC will exit and switch to the triple moving average crossover trading strategy when the moving averages cross.

b) *Triple Moving Averages Crossover Trading Strategy (TMAC).* It uses three MAs: one short, one medium, and one long. The most common MA lengths proven to give good results in practice are: 25-point trading window for the short

MA, 100-point for the medium MA and 200-point for the long MA. The strategy takes a buy decision if: i) Short MA is above the medium MA; ii) Medium MA is above the long MA where the short MA is already over the medium MA.

2) *Price Rate of Change Trading Strategy (PROC).* The most common period for PROC is 12-periods for short-term signals. We decide to use this value for testing, as it is aligned with our high-frequency trading strategy trend-following approach. Generally, a negative PROC value shows that the market is being oversold, while a positive PROC value observes the market as being overbought. In our case, when the PROC value $\leq -30\%$ we decide to take a Buy position [4].

3) *Bollinger Bands Trading Strategy (BB).* We use a 20 period EMA as the "middle band" (one of the mostly used values for short-term trend identification in the financial markets), thus our "lower band" and "upper band" BB values being based on a 20 period prices standard deviation as well. This trading strategy acts as a filter on top of the other trading strategies, alongside the PROC trend-following strategy, for further trend direction strength optimisation [4].

6 Evaluation

Our PRTS implementation runs at a clock frequency of $150Mhz$, while our FRTS and SRTS implementation run at a clock frequency of $175Mhz$. For the PRTS approach an increase in clock frequency from $150Mhz$ to $175Mhz$ is difficult to obtain as it was hard to meet timing requirements. All the run times are measured by using the `chrono::high_resolution_clock` which is part of the C++11 standard library. Both CPU and FPGA times measured include the time to process the total market ticks (respectively the total market ticks between switches). We perform different experiments on both synthetically generated FX GBP/USD market data at different trading frequencies, as well as historical data: First, we analyse the speedup and returns for SRTS, PRTS and FRTS designs in an offline environment. Then, we identify the best trading opportunity when checking the obtained performance for multiple data set dimensions, and different trading-window switch frequencies. Last, we simulate a real-time trading environment, thus accounting for the data loss encountered during different trading-window switch durations when using FRTS and PRTS.

Resource Usage Results. We analyse what each of our framework configurations (SRTS, PRTS, FRTS) properties are, when using the trading strategies implemented. Table 2 presents the FPGA total resource usage expressed as a percentage of the total available resource on the chip for the single precision floating point implementation of all individual static trading strategies kernels. It also shows the FPGA total resource usage for the SRTS solution which implements all the presented trading strategies (TMAC, DMAC, PROC and BB) and runs them all in parallel. Our framework gives the users the flexibility for both SR and PR to pick if they want to run all strategies in the same time, or if they want to switch between a few of them and/or run others in the same time as well (possibly with different parameters - in practice, traders choose to run the same trading strategy with different parameters in parallel, so that they can identify its optimum coefficients at any point in time). Table 3 shows the resource usage for the generalised framework version as described previously.

Table 2: FPGA total resource usage for static kernels. Measurements are provided for 864M data points and 175Mhz clock frequency

# Kernel	LUTs	FFs	BRAMs	DSPs	Total Logic
TMAC	11.02%	14.02%	9.92%	1.19%	17.75%%
DMAC	10.65%	14.17%	9.92%	0.79%	17.79%
PROC	10.88%	14.13%	9.92%	0.20%	18.19%
BB	11.15%	14.48%	10.06%	0.74%	18.33%
All Static	14.12%	17.27%	14.94%	2.98%	22.22%

Table 3: Resource Usage for PRTS and FRTS provided for 864M data points

Property	PRTS	FRTS	Static
Compute Clock Frequency (MHz)	150	175	175
Memory Clock Frequency(MHz)	400	400	400
Total Logic (4 trading strategies)	18%	15%	23%
Total Logic (8 trading strategies)	32%	27%	38%

This framework can be used to provide guidance on which trading strategy performs best under different market regimes, but for this to be optimal many trading strategies would need to be implemented on-chip. Having just a static kernel solution would work as long as we have enough resources. However, adding new trading strategies kernels will quickly scale the total resource usage on-chip until it becomes completely unusable. Increasing the resources will further reduce the clock frequency thus making the FPGA approach slower as well. We can then use the FRTS approach to try and save area on-chip or we could use the PRTS solution to use many trading strategies to switch from, while still being able to run potentially a few more in parallel, as shown in Figure 3(i).

Offline - Static Reconfiguration Trading Strategies on FPGA. We run all trading strategies in parallel, with the parameters presented in Section 5, and we employ a majority rule trading strategy - if the majority of the trading strategies tell us to BUY then we BUY. Table 4 shows that our SRTS approach obtains up to 11 times speedup when compared to its fully optimised CPU C++ software version. Also, the speedup increases with the number of market data points evaluated, thus confirming the FPGA choice for this compute-bound problem.

Table 4: All Static - FPGA performance results

# Market Ticks	1.152M	28.8M	86.4M	432M	604.8M	864M
CPU Time (s)	0.05	1.38	4.15	24.51	32.97	54.23
FPGA Time (s)	0.03	0.14	0.44	2.45	3.45	4.91
FPGA Speedup	**1.67**	**9.85**	**9.43**	**10.00**	**9.55**	**11.04**
Returns ($\$M$)	**1.679**	**2.805**	**4.831**	**6.204**	**8.205**	**10.178**

Offline - Partial Reconfiguration Trading Strategy on FPGA. We evaluate our solution using TS1 (TMAC) and TS2 (DMAC) as part of the PR region and TS3 (PROC) and TS4 (BB) as the static kernels acting as momentum filters running in parallel with the partial reconfiguration module. Table 5 presents the DMAC and TMAC evaluation results using the partial reconfiguration solution on FPGA. Every X number of cycles (X corresponding to the market entry points from a particular trading window e.g. 5 minutes trading window) we have

the option to perform a switch, which in our case it happens if the market conditions explained in Section 5 are met. As expected, Table 5 identifies that more frequent switches decrease our system's speedup, because for every switch to be performed during the run-time reconfiguration, we lose $70ms$ (see formula (7)).

Table 5: PRTS - FPGA performance results for 864M Market Data Entries

Switch Frequency (min)	0.5	1	5	15	30
# of switches	374	192	58	24	10
CPU Time (s)	48.12	45.76	42.88	41.01	40.88
FPGA Time (s)	31.90	19.15	9.81	7.43	6.42
PRTS Speedup	**1.51**	**2.39**	**4.37**	**5.52**	**6.36**
PRTS Returns ($M)	**1.428**	**2.903**	**5.890**	**7.965**	**12.361**

Offline - Full Reconfiguration Trading Strategies on FPGA. *Table 6* draws the same conclusion as the PRTS one, that more frequent switches decrease our system's speedup. This method shows a noticeable decrease in the speedup due to the fact that each time we perform a switch we need to unload the old FPGA design and reload a new one. All our designs depend on the use

Table 6: FRTS - FPGA performance results for 864M Market Data Entries

Switch Frequency (min)	0.5	1	5	15	30
# of switches	374	192	58	24	10
CPU Time (s)	55.34	49.12	46.02	45.38	42.29
FPGA Time (s)	607.02	314.03	98.28	44.62	21.18
FRTS Speedup	**0.09**	**0.16**	**0.47**	**1.02**	**2.00**
FRTS Returns ($M)	**0.781**	**1.290**	**4.203**	**6.992**	**8.834**

of DRAM: in the FRTS case we notice an additional DRAM average load time of 1.61s (see Formula (6)), for the 864M data entries we need to transfer each time. After ignoring the communication latency over PCIe/Infiniband which is of the order of milliseconds, we obtain an average total transfer+load time of 1.66s.

PRTS vs FRTS in a Trading Simulated Environment. Table 7 shows returns for both PR and FR approches when trying to simulate a continuously streaming algorithm. We notice that when we lose access to the data corresponding to the switch time period, we seem to be decreasing our overall returns, as well as encounter losses at times. However, the PRTS returns are higher than the FRTS ones which shows that simply switching between trading strategies at different times is not good enough, but by introducing momentum filters (as in the PRTS approach) we better account for the financial markets condition changes and avoid under-performance of one particular selected trading strategy.

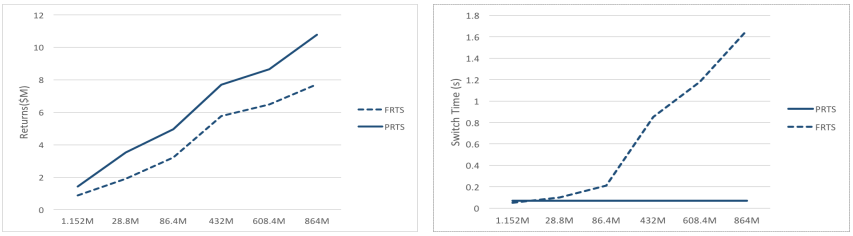
Table 7: PRTS vs FRTS return results for 864M Market Data Entries

Switch Frequency (min)	0.5	1	5	15	30
PRTS # of switches	302	240	78	44	25
PRTS Returns ($M)	**0.712**	**2.108**	**5.264**	**7.112**	**10.780**
FRTS # of switches	402	287	101	53	17
FRTS Returns ($M)	**-5.611**	**-2.889**	**1.017**	**3.859**	**7.513**

Figure 4(i) shows the different returns for both PRTS and FRTS solutions, when we account for the data loss that would appear during the switch time.

This graph presents a 30 minute trading strategied switch frequency over different market data entries (i.e: 28.8M). The "real-time" simulation of our trading strategy shows that when losing access to the data corresponding to the switch time period, returns decrease as data become less reliable and more volatile.

Figure 4(ii) shows the different switch times corresponding to each of the respective number of market entries, when running both PRTS as well as FRTS, using all the implemented trading strategies from the strategy kernel library. We notice PR solution regardless the number of market entries, while it increases with the increase of the market data points in the case of the FR implementation.



(i) Returns($\$M$) vs Market Data(M) (ii) Switch Time(s) vs Market Data(M)

Fig. 4: Returns and Switch Frequency for FRTS and PRTS

SRTS vs PRTS vs FRTS Returns for Historical FX Market Data. We verify the applicability of our proposed approach by evaluating Historical GBP/USD tick-data from the Foreign Exchange Market which corresponds to time-periods from 2003/2004. Our historical tick data include transaction costs. Table 8 shows returns for all static, partial-reconfiguration as well as full-reconfiguration approach when trying to simulate a continuously streaming algorithm using historical market data. We can notice a slight decrease in the return levels from 2003 and 2004, being very much in accordance with the greater FX market efficiency in 2004 compared to 2003 (i.e a growth in electronic high-frequency trading occurred during the 2003 - 2008 period).

Table 8: SRTS vs PRTS vs FRTS - March (10-14) 2003, 2004

Switch Frequency (min)	0.5	1	5	15	30
2003					
SRTS Returns ($\$M$)	**0.412**	**0.402**	**0.899**	**0.594**	**0.656**
PRTS Returns ($\$M$)	**0.335**	**0.484**	**0.805**	**0.661**	**0.579**
FRTS Returns ($\$M$)	**0.147**	**0.388**	**0.689**	**0.570**	**0.601**
2004					
SRTS Returns ($\$M$)	**0.360**	**0.321**	**0.680**	**0.694**	**0.557**
PRTS Returns ($\$M$)	**0.272**	**0.301**	**0.760**	**0.612**	**0.514**
FRTS Returns ($\$M$)	**0.082**	**0.258**	**0.516**	**0.432**	**0.491**

7 Conclusion and Future Work

Our study aims to provide the first framework for developing and comparing multiple trading strategies for FPGA designs. Our tool offers the user multiple solutions for running their trading strategies. Three architecture types are supported: static, partial-reconfiguration and full-reconfiguration. Our approach offers alternative solutions when a static design becomes too large because of too

many different trading strategies or the trading strategies themselves are very complex and occupy a significant amount of resources [8]. If the resources of a given device run out, a larger FPGA would be needed, but if not available, our framework offers the user a low-cost, resource and performance efficient solution.

We show that FPGAs can effectively accelerate a system based on multiple trend-following trading strategies which come as an initial library for our framework. Our SRTS design achieves 11 times speedup, the PRTS design achieves 7 times speedup, while the FRTS design achieves up to 2 times speedup, when compared to the corresponding multi-threaded C++11 implementation running on a six-core Intel Xeon CPU X5650 processor. After testing our tool on historical FX data, we show that trading strategies supported by the proposed design are reliable and, if further exploited, can increase profitability from high frequency FX markets trading. Thus, applying different trading strategies based on different market regimes would help the modeling process better reflect the reality.

Opportunities for further work include adding support for varying data representation and evaluating speedup/returns improvements on more recent financial market data. We could include multiple copies of trading strategies on-chip so that one could start processing without waiting for a previous computation to finish. We also aim to enhance the trading strategies kernel library, implementing additional effective strategies on the FPGA, as well as developing macroeconomic and news factors for stock and fixed income trading. We further plan to include designs to optimally detect regime change, such as those based on permutation entropy [9]. In the future, we will also make our framework available as open source, to allow developers to add their custom strategies to the library.

Acknowledgments The support of UK EPSRC (EP/I012036/1, EP/L00058X/1, EP/L016796/1 and EP/N031768/1), the European Union Horizon 2020 Research and Innovation Programme under grant agreement number 671653, the China Scholarship Council, the Maxeler University Programme, Altera, Intel and Xilinx is gratefully acknowledged.

References

1. S. Wray et al., *Exploring algorithmic trading in reconfigurable hardware*, ASAP, 2010.
2. Altera, *FPGA Run-Time Reconfiguration: Two Approaches - White Paper*, `ftp://ftp.bittware.com/documents/fpga-run-time-reconfiguration.pdf`.
3. M. Driver, Foreign Exchange: A Practical Guide to the FX Markets, CreateSpace, 2012.
4. I. Aldridge, *High-Frequency Trading: A Practical Guide to Algorithmic Strategies and Trading Systems (Wiley Trading)*, John Wiley & Sons; 2nd Edition, 2013.
5. Maxeler Technologies, *MaxCompiler-WhitePaper*, `https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf`, 2001.
6. C. Leber et al., *High Frequency Trading Acceleration Using FPGAs*, FPL, 2011.
7. M. Mastinu, *Design flow to support dynamic partial reconfiguration on Maxeler architectures*, Politecnico di Milano, 2012.
8. A. I. Funie et al., *"Reconfigurable acceleration of fitness evaluation in trading strategies"*, ASAP, 2015.
9. C. Guo et al., *"Pipelined Reconfigurable Accelerator for Ordinal Pattern Encoding"*, ASAP, 2014.