Stream-Based Representation and Incremental Optimization of Technical Market Indicators

Stream-Based Representation and Incremental Optimization of Technical Market Indicators

Konstantin Bakanov
EEECS
Queen's University Belfast
Belfast, UK
kbakanov01@qub.ac.uk

Ivor Spence
EEECS
Queen's University Belfast
Belfast, UK
i.spence@qub.ac.uk

Hans Vandierendonck
EEECS
Queen's University Belfast
Belfast, UK
h.vandierendonck@qub.ac.uk

***Abstract*—Technical market indicators are used to measure the trends of financial markets. In practice they are conventionally expressed using a non-formal notation or a DSL specific to a certain development platform, which poorly correlates between individual trades and the high-level formulas operating on those trades and leaving very little room for optimization. In this paper we propose a formal, mathematically based notation for expressing technical market indicators, which represents trades as streams of data. We argue that this notation is more accurate and open to optimizations. We express three technical indicators from the ground up, demonstrate our optimization approach, and implement the indicators using Click router runtime. Finally, we benchmark various configurations and versions of the implemented indicators, running in kernel space as well as user space, and discuss the findings.**

***Index Terms*—domain-specific languages; technical market indicators; in-kernel processing; click router; data streams; high-frequency trading**

I. Introduction

Financial markets are positioned at the heart of any modern economy. In the last 30 years the financial markets have transitioned into high performance systems with high degrees of automation. Along with this transition the amount of trading activity has risen dramatically, which is why now more than ever the traders have to rely on special algorithms to help them with their day-to-day tasks. One class of these algorithms is commonly known as Technical Market Indicators. They are used to analyze the state of the market, enabling the traders to predict its future behaviour and to form a potentially profitable trading strategy.

Technical indicators themselves are mathematical formulas applied to a series of past trades. Indicators vary in their function: they can be used to operate on one stock or on many stocks (market-wide indicators), they can be used to measure momentum, sentiment, trend [1] and other properties of the security[1] in question. Numerous books and guides have been written on that topic [2]–[4], which describe indicators, evaluate their past performances and give recommendations for their future application. However indicators suffer from a universal problem: their description is ambiguous because it is not rigorous, the notation is either too vague and textual (as can be seen in many examples from [4]) or too confined to a particular technical analysis tool (as can be seen from various implementations of Vortex indicator in [5]). The runtime is most often a closed source implementation, prohibiting the application of any sort of optimization or transformation to the indicators.

We address these problems by developing an open stream oriented DSL, executed in an open source runtime. In Section II-B we introduce our notation and describe the basic principles using TRIX indicator as an example. Then building upon the extensive research of Yanhong A. Liu [6] in Section II-C we suggest a way for an automatic transformation of unoptimized indicators into an optimized incremental implementation. To demonstrate the versatility of our notation in Section III we express two additional indicators: DMI and Vortex indicator. Finally, in Section IV we manually implement unoptimized and optimized versions of indicators using the Click modular router [7] and evaluate their performance in a High-Frequency Trading-like (HFT) environment.

The contribution of this paper is as follows: we propose an innovative precise and unambiguous stream based representation of technical market indicators, we apply an incrementalization technique to our notation in order to derive an exact and efficient implementation, we propose an efficient runtime system for execution of indicators, which we evaluate and benchmark with the help of the aforementioned indicators.

II. Notation

A. Technical Market Indicators

Whenever a transaction happens on a financial exchange an electronic message (a trade) is broadcast to all participants over the network. The trade contains such information as the name of the security, which has been transacted, the price, volume and time of the transaction. The trading software on the recipient's side accumulates those trades, forming a stream. That stream is broken up into equally sized time intervals (see Figure 1). Depending on the origination timestamp each trade belongs to some particular interval. Some intervals may contain many trades, some may contain very few. For each time interval the opening, closing, highest and lowest trade prices are computed (also called a "candlestick" [8]) along with the total volume (sometimes). Technical Indicators take in a stream of those High/Low/Open/Close/Volume tuples

[1]"Stock" and "security" mean the same thing for the purpose of this paper.

and perform some computations on them. The value of an indicator is recalculated on each update (if it changes relevant High/Low/Open/Close values) and upon the start of a new interval.

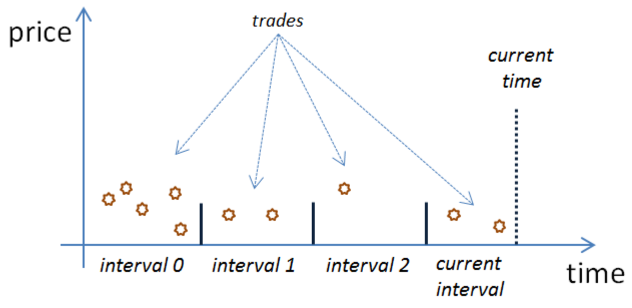

Fig. 1: Trades occurring in an interval.

B. Stream Based Functional Notation

The purpose of our notation is to allow any indicator developer to express new indicators easily and precisely, which is why we chose a purely functional notation as it is simple and easy to optimize. Since the incoming data arrives as a stream of messages we have built our notation around the stream-oriented programming. First, the indicator developer needs to identify the functions making up an indicator and the streams connecting them. This permits the construction of an interconnect diagram, where functions, identified by names, are connected by arrows (streams). This approach is borrowed from our runtime (which is underpinned by Click router framework). It is also referred to as Click language. Let us continue our explanation using a very simple TRIX indicator [9] as an example.

It is usually calculated as follows: the closing prices of a particular stock are smoothed 3 times with exponentially weighted moving average (EWMA) and then a difference is taken between the current period's triple smoothed closing price and previous period's triple smoothed closing price, divided by the previous period's triple smoothed closing price.

First, we need to produce an interconnect diagram, which looks as follows (without parameters and declarations):

```
SOURCE.C -> ewma1 -> ewma2 -> ewma3
     -> TRIX -> DISPLAY
```

where
`SOURCE.C` - is the source of closing prices.
`ewma1, ewma2, ewma3` - are the EWMA smoothing functions.
`TRIX` - is the function doing the calculation.
`DISPLAY` - is a notional function that displays the final result.

After producing this interconnect diagram we can express each function using our functional notation. In order to streamline and simplify the writing of new indicators we have built our notation around a number of principles:

- Each function takes in one or more streams of data and produces exactly one output value. It is the job of the runtime to turn that value into a stream, which will serve as an input to the subsequent function.
- Each function treats the incoming stream as a completely static object. I.e. the developer does not need to worry about new intervals being added as the time moves forward. The function is only concerned with calculating the output value once. Again, it is the job of the runtime to turn that static calculation into a dynamic one.

Using the above principles the EWMA function will be represented as follows:

```
EWMA : ℝ*,ℝ → ℝ
EWMA (S, α) = if (#S = 1)
            then head(S)
            else (1 - α) * head(S) +
              α* EWMA(tail(S),α)
```

where
The first line is the function signature. From the signature we can see that the function takes in a stream of ℝeal values (the asterisk denotes a stream) and a single ℝeal value. The output is a single ℝeal value.
$head(S)$ returns the last element of a stream.
$tail(S)$ returns the tail of a stream, i.e. all elements except the head.
$\#S$ is the length of the stream S.

TRIX will look as follows:

$$\texttt{TRIX: } \mathbb{R}^* \rightarrow \mathbb{R}$$
$$\texttt{TRIX(S)} = \frac{\texttt{head(S) - head(tail(S))}}{\texttt{head(tail(S))}}$$

where
The first line is once again the signature, from which we can see that the function takes in a stream of ℝeal values. The output is a single ℝeal value.
$head(S)$ is the last element in a stream.
$head(tail(S))$ is the penultimate element in a stream.

In the following section we will explain how to turn that static representation into a dynamic one.

C. Indicators' Optimizations (Incrementalization)

Another key contribution of our paper is the transformation of the static functional notation into dynamic incremental format. We heavily base our ideas upon the substantial work on incrementalization by Yanhong A. Liu, which culminated in her book [6]. Our reasoning and the rules are the adaptation of [10] applied to our functional notation. It is made of a number of steps:

1) We find the minimum increment operation.
2) We apply the minimum increment operation to the function in question.
3) We simplify the resulting function and detect what additional variables besides an increment are needed to

compute the next value without resorting to the original stream.

4) We then derive three functions:
   a) We modify the original function so that it returns not only the return value, but also the additional variables needed for an incremental computation.
   b) We derive an incremental function, which computes the next value only by using the increment and the additional variables.
   c) We extend the incremental function to return the additional values needed by the next computation.

Using TRIX as an example the minimum increment operation (Step 1) is the addition of the new sequence[2] element n to the sequence S. We designate this operation with prepend $\triangleleft$ operator as described by [11] when applied to sequences.

We now write the TRIX function with an incremented input and see what is the difference compared to the unincremented input (Step 2). This allows us to find a set of additional values that should be returned by the unincremented function so that the incremented function would be able to work.

$$\texttt{TRIX:}\ \mathbb{R}^* \rightarrow \mathbb{R}$$
$$\texttt{TRIX(n} \triangleleft \texttt{S)} = \frac{\texttt{head(n} \triangleleft \texttt{S)} - \texttt{head(tail(n} \triangleleft \texttt{S))}}{\texttt{head(tail(n} \triangleleft \texttt{S))}}$$

We now do some simplification (Step 3): $(head(tail(n \triangleleft S))$ is the same as $head(S)$, which means we need to store the last element from our previous computation.

$head(n \triangleleft S)$ is the same as n which is our increment input. Therefore all we need to store between incremental computations is the last element of S.

Let us now introduce the notion of the extended, optimized and optimized extended functions, which correspond to Step 4 in our transformation rules.

The whole pipeline from the start of operation is shown in Figure 2. The initial stream of values is passed to the extended (_EXT) function, which is essentially the original function that returns auxiliary values in addition to the result itself. When applied to TRIX indicator it will look as follows (where S is the input stream of values):

$$\texttt{TRIX\_EXT:}\ \mathbb{R}^* \rightarrow <\mathbb{R},\mathbb{R}>$$
$$\texttt{TRIX\_EXT(S) = <TRIX(S), head(S)>}$$

The return value of the extended function is always a tuple (denoted by $<>$). The first element of the tuple is the actual result value (the upward arrow in Figure 2), which is returned to the user or passed along through the runtime. The remaining elements in the tuple are the auxiliary values which are stored by the runtime and then used in the subsequent calls. For TRIX indicator the auxiliary value is the last value of S.

An optimized function (with the suffix _OPT) is the function that takes in the increment, the auxiliary values (returned by extended function) and computes the next result. When applied to TRIX indicator it will look as follows (where n is the incremental input, i.e. a new element in the stream, and p is the previous value):

$$\texttt{TRIX\_OPT:}\ \mathbb{R},\mathbb{R} \rightarrow \mathbb{R}$$
$$\texttt{TRIX\_OPT(n, p) = (n-p)/p}$$

However, since we also need the auxiliary values for the next call, the optimized function must also be extended to return those auxiliary values. Such function has the suffix _OPT_EXT and its return value is the same as the one of the extended function. Just as with the extended function, the auxiliary values are cached and then supplied as parameters in the next invocation of the _OPT_EXT function. When applied to TRIX it looks as follows:

$$\texttt{TRIX\_OPT\_EXT:}\ \mathbb{R},\mathbb{R} \rightarrow <\mathbb{R},\mathbb{R}>$$
$$\texttt{TRIX\_OPT\_EXT(n, p)=<TRIX\_OPT(n, p), n>}$$

Note, that the optimized extended function will always take in first the same parameters as the original function (except that instead of an input sequence we get an increment), followed by the auxiliary parameters cached by the runtime.

In this paper we only describe the rules for indicators' transformation (i.e. the derivation of the _EXT and _OPT_EXT functions). It is beyond the scope of our research to develop a tool for this. We rest upon [12] to conclude that such tool is feasible and possible, albeit with the limitations outlined therein as well. In particular, [12] states that in order for the process to be fully automatable the analyses employed in the process (program analyses for dependencies, types, aliases, costs and simplification for equality reasosoning) need to be automatable themselves and that finding an incremental operation needs to be solvable (decidable). In our case case the incremental operation is already known and is an addition of a new element to the sequence. The simplification for equality reasoning and program analysis are also straightforward as we are always dealing with the same data structure - a sequence and an increment. By resolving selector operations on a sequence (e.g. $head(S)$, $head(tail(S))))$ we can always find out the exact elements of a sequence that a given function depends on. By comparing the resolved elements of a sequence with an increment we can establish whether this given function depends on the sequence at all or it only needs an increment. To demonstrate our approach we provide a few examples (EWMA, DMI and Vortex) to serve as a guide where automation can work, and an example (SUM) where our approach does not work automatically. Overall and in general, as [12] states, whether a given function can be automatically transformed, depends on that individual function and whether the automatable analyses can be successfully applied to such function. In cases, where the entire process cannot be fully automated, some analyses can still be useful in aiding the algorithm designer to produce an optimized function.

In the rest of this document we will refer to the original, baseline functions as naïve and to the optimized ones as optimized.

[2]We use terms "stream" and "sequence" interchangeably.

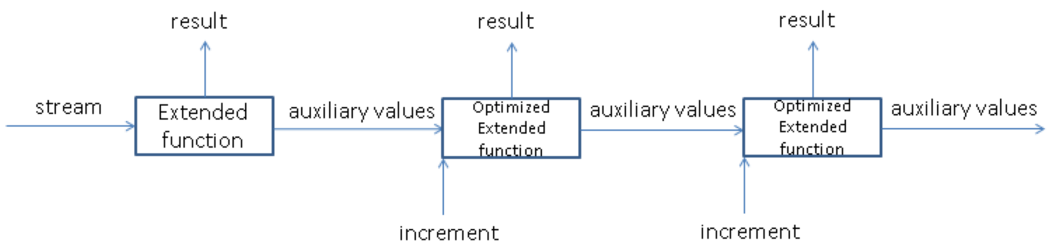

Fig. 2: Extended and Optimized Extended Functions

D. EWMA and SUM

Let us now apply the same transformation rules to EWMA:

```
EWMA : $\mathbb{R}^*, \mathbb{R} \to \mathbb{R}$
EWMA (n ◁ S, $\alpha$) = if(#(n ◁ S) = 1)
    then head(n ◁ S)
         else(1 - $\alpha$)*head(n ◁ S) + $\alpha$
            *EWMA(tail(n ◁ S), $\alpha$)
```

We now do the simplification:
$head(n \triangleleft S)$ is the same as the increment n. $EWMA(tail(n \triangleleft S), \alpha)$ is the same as $EWMA(S, \alpha)$, which is the previous value of EWMA.

We also know that whenever an optimized function is invoked, the stream S will already contain at least one element, meaning that we can eliminate the *then* clause from the optimized function. This means that all we need to store is the previous value of EWMA. The extended, optimized and optimized extended functions will look as follows (where n is the incremental input, e is the previous value of EWMA):

```
EWMA_EXT: $\mathbb{R}^*, \mathbb{R} \to <\mathbb{R}, \mathbb{R}>$
EWMA_EXT(S, $\alpha$)=<EWMA(S, $\alpha$), EWMA(S,
     $\alpha$)>

EWMA_OPT: $\mathbb{R}, \mathbb{R}, \mathbb{R} \to \mathbb{R}$
EWMA_OPT(n, $\alpha$, e)=(1-$\alpha$)*n+$\alpha$*e

EWMA_OPT_EXT: $\mathbb{R}, \mathbb{R}, \mathbb{R} \to <\mathbb{R}, \mathbb{R}>$
EWMA_OPT_EXT(n, $\alpha$, e)=<EWMA_OPT(n,$\alpha$
    , e), EWMA_OPT(n,$\alpha$, e)>
```

Another frequently used function is the SUM. Its conventional version expressed in our functional notation will look as follows:

```
SUM: $\mathbb{R}^*, \mathbb{N} \to \mathbb{R}$
SUM(S,j) = if(j = 0) then 0
         else head(S) + sum(tail(S)
             , j - 1)
```

where j is the number of elements to sum up.
If we apply our optimization rules here, we will see that these fail to produce the optimized version of the above function. This is due to the fact that an optimized form would involve a sliding window sum calculation (as it requires j last elements of the stream to be computed), which is beyond the scope of our work. For cases like this a manual optimization needs to be done.

III. Defining Indicators

In order to demonstrate our functional notation we use it to express 2 more indicators besides TRIX: DMI and Vortex indicators. We have chosen these indicators because they are complex in terms of their structure, they share a common element, and at the same time they are sufficiently diverse. First, we introduce the indicators, provide their interconnect structure and then provide a table with all functions. The implementation details and benchmarks are discussed in Section IV.

A. DMI

This indicator is described in [13] as follows (verbatim excerpt):

> "The Directional Movement Index (DMI) is a unique filtered momentum indicator ... DMI is a rather complex trend-following indicator."

DMI refers to a number of inter-related indicators: a positive directional indicator, a negative directional indicator and their combinations, such as ADX. In this paper we will concentrate on expressing ADX. First we need to calculate Positive and Negative Directional Movements (PDM and NDM) and the True Range (TR). These are then smoothed and fed into Positive and Negative Directional Indicators (PDI and NDI), which are in turn fed to Directional Movement (DX), the smoothed output of which happens to be the ADX. The simplified interconnect diagram looks as follows:

```
SOURCE -> PDM -> EWMA_PDM -> [0]PDI
SOURCE -> NDM -> EWMA_NDM -> [0]NDI
SOURCE -> TR -> EWMA_TR

EWMA_TR[0]-> [1]PDI -> [0]DX
EWMA_TR[1]-> [1]NDI -> [1]DX

DX -> EWMA_DX -> DISPLAY
```

where
[0] prefix in front of an element, e.g. $[0]PDI$, is the

TABLE I: DMI and Vortex Indicators. S_H, S_L, S_C all denote a corresponding stream of high, low and closing prices. H_n and H_p denote new and previous high prices, the same applies to C_n, C_p (closing prices) and L_n, L_p (low prices).

DMI		Vortex
P, N: $\mathbb{R}^* \to \mathbb{R}$	MAX: $\mathbb{R},\mathbb{R} \to \mathbb{R}$	VMU,VMD: $\mathbb{R}^*,\mathbb{R}^* \to \mathbb{R}$
P(S)=head(S)-head(tail(S))	MAX(a,b)=if (a>b)	VMU(S_H,S_L)=abs(head(S_H)-
N(S)=head(tail(S))-head(S)	then a	head(tail(S_L)))
	else b	VMD(S_H,S_L)=abs(head(S_L)-
PDM, NDM: $\mathbb{R}^*,\mathbb{R}^* \to \mathbb{R}$	TR: $\mathbb{R}^*,\mathbb{R}^*,\mathbb{R}^* \to \mathbb{R}$	head(tail(S_H)))
PDM(S_H,S_L)=if (P(S_H) < N(S_L))	TR(S_C,S_H,S_L)=MAX(head(tail(S_C)) - head(S_L),	
then 0	MAX(head(S_H)-head(S_L),	VIU,VID: $\mathbb{R}^*,\mathbb{R}^* \to \mathbb{R}$
else if(P(S_H) < 0)	head(S_H)-head(tail(S_C))))	VIU(S_{VMU},S_{TR})=$\frac{\text{head}(S_{VMU})}{\text{head}(S_{TR})}$
then 0	PDI, NDI: $\mathbb{R}^*,\mathbb{R}^* \to \mathbb{R}$	VID(S_{VMD},S_{TR})=$\frac{\text{head}(S_{VMD})}{\text{head}(S_{TR})}$
else P(S_H)	PDI(S_{PDM},S_{TR}) $= \frac{\text{head}(S_{PDM})}{\text{head}(S_{TR})}$	
NDM(S_H,S_L)=if (N(S_L) < P(S_H))	NDI(S_{NDM},S_{TR}) $= \frac{\text{head}(S_{NDM})}{\text{head}(S_{TR})}$	
then 0	$\text{DX}(S_{PDI}, S_{NDI}) = 100*$ $\frac{\text{abs}(\text{head}(S_{PDI}) - \text{head}(S_{NDI}))}{\text{head}(S_{PDI}) + \text{head}(S_{NDI})}$	
else if(N(S_L) < 0)		
then 0		
else N(S_L)		

ports mechanism within the Click language, which we employ to write the interconnect diagrams. It allows for multiple streams to "feed" into a single function. We can see that PDM and TR both feed into PDI at the ports 0 and 1 respectively.
Similarly the suffix port, e.g. $EWMA\_TR[1]$ allows to duplicate the outgoing stream. In our experimental setup we used a separate `Tee` element for this purpose (which splits a given stream into multiple ones), but we use suffix here as a shorthand.

The naïve versions of the DMI elements are provided in Table I. Optimized ones are omitted as their derivation is trivial.

B. The Vortex Indicator

Vortex Indicator [5] is similar to DMI in that it is made of two interrelated indicators: VIU (Vortex Indicator Up) and VID (Vortex Indicator Down). First we need to calculate Vortex Movements Up and Down (VMU and VMD) and True Range (TR). Then the results are fed into the summation function, with the typical value of 21 intervals. Then the outputs of summations are fed into the final indicators of VIU and VID.

The interconnect structure will look as follows:

```
SOURCE -> VMU -> SUM_A -> [0]VIU
SOURCE -> VMD -> SUM_B -> [0]VID

SOURCE -> TR -> SUM_C

SUM_C[0] -> [1]VIU
SUM_C[1] -> [1]VID
```

The naïve functions are described in Table I. The optimized ones are omitted as their derivation is trivial.

IV. Runtime

A. Click Modular Router

The Click modular router [7] is a software router, which was designed for researching and rapid prototyping of various routing configurations, and which we extended for computing technical market indicators. It can also execute in kernel space as well as in user space, which can be used to reduce "wire-to-processing" latency.

The Click's runtime is built around the concept of elements (processing functions), that connect to each other using data streams. A notation called Click language is used to connect the elements together via the means of a configuration file [7].

In our work we use Click runtime as a proof of concept to demonstrate that, in principle, our notation can be used to process market data streams at HFT speed (low microseconds), that our optimizing transformation can have a positive effect on latency and that our modular approach allows for other efficient optimizations: multithreaded and element (module) sharing.

The indicators are made of multiple Click elements. We have created a base class, which acts as a runtime to our indicators (by maintaining the cache and creating an abstraction layer), and from which each element making up an indicator must inherit. On top of this base class we have carefully mapped naïve, _EXT (extended) and _OPT_EXT (optimized extended) functions onto C++ code, so that it corresponds directly to our notation. Each element can work in two modes: naïve and optimized. In the naïve mode only the naïve function is invoked by the runtime. In the optimized mode during the startup phase the extended function is invoked, which in turn calls the naïve function. After the startup is finished, only the optimized extended function is called.

In front of indicators we have a special class, named the Trade Processor (TP), which is a single Click element. The input to TP is trade messages, coming from the network card

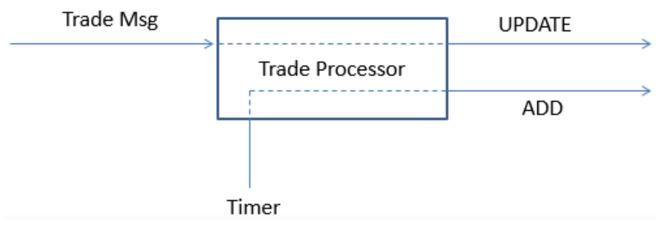

Fig. 3: Trade Processor

and carrying such information as the security name, the size and the price of a transaction. The output are messages of 2 types: ADD and UPDATE (see Figure 3). As can be seen from the picture the ADD messages are generated by the timer inside a TP, whereas UPDATE messages are caused by incoming trades. ADD messages signify the start of the new interval (as shown in Figure 1) and cause the cache to be updated.

V. Benchmarks Setup

Typically traders in the High-Frequency Trading environment are mostly concerned with latency, therefore this is what we will be measuring here. For our experiments we use TRIX, DMI and Vortex indicators.

In our setup we have two machines: one is used to run indicators and another one is used to stream the trades as UDP messages. The machine hosting the runtime has `Intel(R) Core(TM) i7 CPU 930` processor running at 2.8 GHz.

The trades originate from the market data playback file kindly provided to us by Fidessa (a company participating in this project). We use the starting 15 minutes worth of the playback file, which translates to around 230 relevant messages, reaching our indicator. For statistical accuracy we repeat the tests 3 times and take the average.

A. Indicators Evaluation

Firstly, we want to determine whether it is worthwhile using Click as a runtime for our indicators. Due to the lack of publicly available standard high-frequency trading libraries (as these are usually a trading secret) we had to develop our own hand coded implementation of each one of the indicators, representing a typical implementation that an HFT developer would write. The latency of our implementation is comparable to that of commercial HFT systems [14] - low microseconds. We have only executed that in the userspace as this is a typical use-case for a typical HFT developer and it serves as a baseline that we compare against.

In our first test we compared the latency of each indicator in hand coded form vs. the latency in Click form running in userspace vs. the latency in Click form running in kernel space (all Click-based indicators are running in optimized mode here). The results are presented in Figure 5. The latency measured is from before the packet reaches the network stack (i.e. right after it comes off the network card driver) and until after the final result is calculated.

Then, we decided to evaluate the latency of naïve implementation of indicators vs. the optimized implementation. We

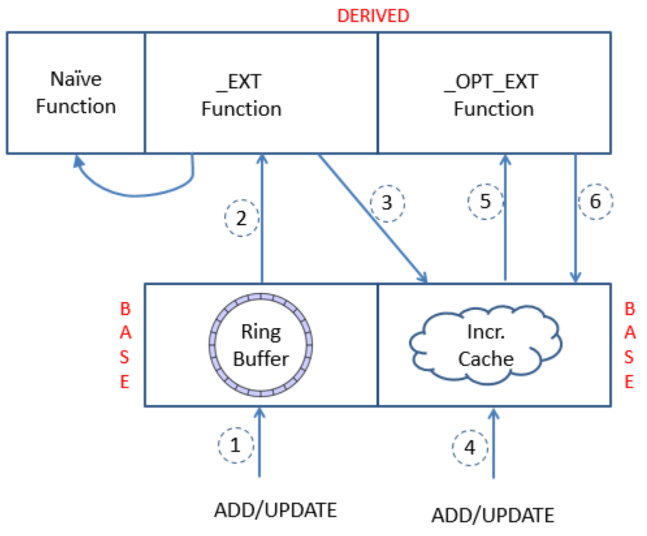

Fig. 4: Indicator cache lifecycle: (1) In the startup mode an Add/Update message is added to the ring buffer. (2) IndicatorBase passes the ring buffer to extended function, which in turn calls the naïve function. (3) Extended function returns cache values needed for incremental computation. (4) During normal operation an Add/Update message is passed directly to the optimized extended function together with (5) cached values. (6) The return values of an optimized extended function contain the updates to the cache.

run these tests in kernel space. We also measure latency, but this time within the indicator (i.e. inside Click): from after the UPDATE message had been formed by TradeProcessor and until after the final value has been calculated. The results are presented in Figure 6.

The results of the tests show that Click-based implementation of our indicators running in kernel space compares favourably with our hand coded baseline. Furthermore, our optimizing transformation does have a positive, but limited effect. After profiling individual elements, we have discovered that cache manipulation (storage and update) can be consuming a significant amount of CPU resources. Therefore for indicators that do not need a lot of historical values in their naïve form, such as DMI, the speedup of the optimized version is not that great. It is most effective where the cost of managing the cache is small and where the decrease in the amount of computation as a result of optimization is relatively large (for example due to reduced number of recursive calls).

B. Multithreaded Optimization

In these tests we are taking advantage of the stream-based nature of our indicators, which makes it easy to isolate and map specific chain links to different threads. This might be especially useful if the user wishes to calculate multiple versions of the same indicator at the same time. For example, TRIX indicator, which we use for these tests, is mostly made of interconnecting EWMA elements. Each EWMA element has a certain fixed α value. Multithreading makes it possible

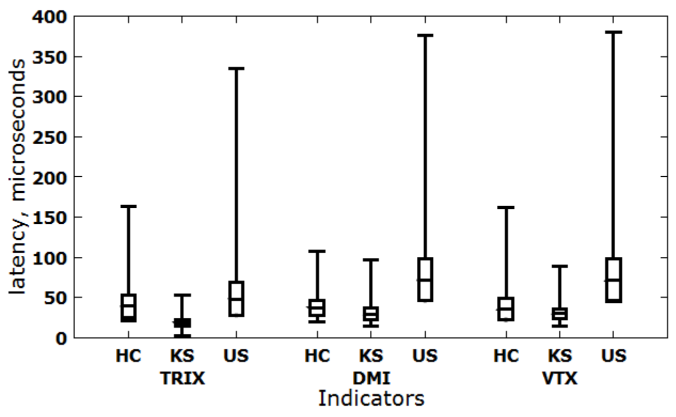

Fig. 5: Execution time of TRIX, DMI and Vortex (VTX) indicators. HC - Hand Coded, KS - Kernel Space, US - User Space. The candlesticks indicate the minimum observed latency, half of a standard deviation below the mean, the mean observed latency, half of a standard deviation above the mean and the maximum latency.

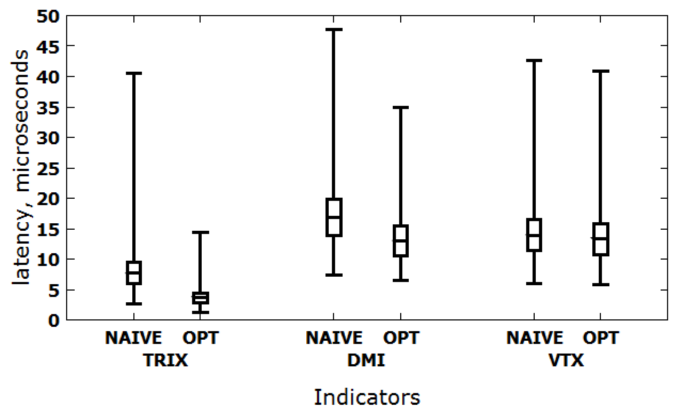

Fig. 6: Execution time of the naïve and optimized versions of TRIX, DMI and Vortex (VTX) indicators. The candlesticks indicate the minimum observed latency, half of a standard deviation below the mean, the mean observed latency, half of a standard deviation above the mean and the maximum latency.

to create a number of TRIX indicators each with a different value of α, but all processing the same UPDATE message simultaneously such as shown in Figure 7.

As with previous tests we are interested in measuring latency (from after the TradeProcessor and until the end of calculation). The testing method was as follows: first, we process a number of instances of TRIX in the same thread and we measure the total time taken to calculate all indicators. Then we try mapping TRIX indicators to different threads and measure the average time that it takes to calculate all indicators. In this experiment we configured Click to run in kernel mode and each kernel thread is set to "greedy" mode, which means that it doesn't sleep for extended periods of time and utilizes nearly 100% of CPU. We mapped threads to tasks using `StaticThreadSched` element. We used standard Click elements `Queue` and `Unqueue` to pass messages between threads. The CPU on the test machine has a total of 8 cores. The thread that handles the reading of incoming trades, their processing in the TradeProcessor and enqueueing internal messages is referred to as "producer" thread.

Initially, we instantiated eight TRIX indicators, as shown in Figure 7, which we then ran using one, two, four, six and seven threads . Since we had more elements than threads we mapped threads to elements in a circular fashion. The results of the experiments are shown in Table II.

TABLE II: Latency per processing of 8 TRIX indicators.

Num Threads	Latency (microseconds)
1	16
2	13
4	10
6	6
7	202

As can be seen from the table, the average latency decrease is proportional to the number of threads involved. However, when running with 7 threads, the latency suddenly spikes and general OS applications become less responsive and often "hang". We assumed that this happens, because there is not enough resources left for the OS tasks. To verify our assumption we created another test with 16 indicators. We then ran it in the following configurations:

- Single thread as in Figure 7. We named this test **1T**.
- Six threads and fifteen queues. Once again we mapped threads to queues using our circular mapping method, i.e. thread 0 would read messages from network card and process indicators 6 and 12 as well as the last one, thread 1 would process indicators 1, 7, 13, thread 2 would process indicators 2, 8, 14 and so on. We named this test **6T-15Q-OLD**.
- Six threads, but this time we designated one thread exclusively as a "producer" thread and did not use it for any other purposes. The other threads were mapped to indicators once again in a circular fashion. We named this test **6T-16Q-NEW**.
- Also six threads, but we created a different more hierarchical topology as shown in Figure 8: we reduced the number of queues down to five and "connected" groups of indicators to the same queue. We put four indicators in the first group and three indicators in other groups (sixteen in total). Each thread was mapped to one queue and the "producer" thread was used exclusively for receiving messages and enqueueing them to the queues. We named this test **6T-5Q-NS**.

The results are presented in Table III.

As can be seen from the results by decreasing the amount of work that the "producer" thread has to do we managed to balance the workload using the new topology such that the OS tasks no longer have a detrimental effect on the latency. We then performed two more tests using the same topology with 20 and 30 indicators. We found that the latency for 20 indicators was still on track, but 30 indicators' setup once again interfered with OS functioning and caused a massive spike in latency.

TABLE III: Latency per processing of 16 TRIX indicators.

Configuration	Latency (microseconds)
1T	34
6T-15Q-OLD	1213
6T-16Q-NEW	72
6T-5Q-NS	9

C. Element Sharing

As can be seen from the interconnect diagrams (Sections III-B and III-A) the Vortex indicator and the DMI indicator both share a common element: the TrueRange. In this experiment we decided to measure how much would we save in terms of latency by sharing that element. First, we benchmarked the calculation of a Vortex and DMI indicators together each with their own respective TrueRange element (from after the UPDATE message has been formed and until the end of calculation of both indicators). The benchmarking was done using Click running in kernel space in a single thread mode. Then we benchmarked the same indicators, but with the TrueRange element being shared between them. We observed the latency decrease of slightly over 8%. Whereas this is not a big saving by itself, it can accumulate when applied to a number of indicators at the same time.

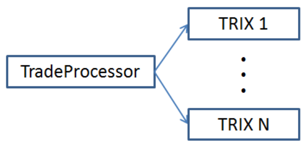

Fig. 7: Multiple TRIX indicators running in a single thread. In multithreaded configurations each indicator is prefaced with a queue.

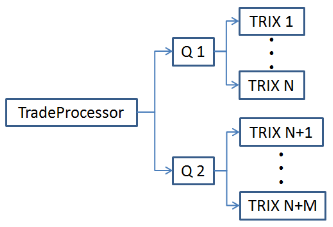

Fig. 8: Multiple TRIX indicators running in multiple threads using hierarchical distribution topology.

VI. Related Work

Whereas we know of no research, which would compare directly with our work, there are a number of publications dedicated to the topic of financial market indicators.

FFTI [15] is an attempt to define a uniform notation for expressing technical market indicators. The authors take an approach whereby they define a number of aggregate functions, such as average, sum, product, min, max etc, and then use those functions with auxiliary parameters to define technical market indicators. They implement their concept as a web based tool.

ChartLingo [16] is a language for expressing technical market indicators on mobile devices. Similar to our notation it uses the notion of datasets (streams), on which the calculations are performed. However, its intended platform are the slow mobile devices, and it does not operate at the low-level of mathematical primitives that our approach suggests.

Then, there is a plethora of proprietary languages developed by various brokerage companies, such as EasyLanguage in TradeStation [17] and MultiCharts [18], AIQ EDS language in AIQ [19], QScript in Wave59 [20] and others.

There are numerous books on the topic of technical analysis, but [3], [4] provide some of the most comprehensive overviews on the topic of technical market indicators, including their description, returns analysis and trading advice.

VII. Conclusion

In this paper we have presented a functional stream based notation and model for the expression of technical market indicators, the incremental optimization of such notation, and the proposed in-kernel runtime based on Click router. We have measured the latency of computing these indicators in various configurations. By evaluating the benchmarks we can conclude that our suggested incremental optimizations do have a positive impact on latency, but depend on the topology of an indicator in question. The multi-threaded optimizations have a very good effect on average latency, but have to be carefully balanced so as not to interfere with the functioning of the OS, when running in kernel mode.

Overall our approach is comparable and sometimes superior in terms of latency to the hand coded versions of indicators (when executing in kernel space), however it also has a number of other advantages: it makes it possible to create new indicators out of the library of the already existing components without requiring any programming skills, it makes it possible to share elements, which has a positive impact on latency, it makes it possible to easily scale multiple indicators across multiple threads. Furthermore, when creating new elements, our notation makes it possible to potentially automatically convert the functional notation into efficient implementation, provided the needed tools are created.

References

[1] R. W. Colby, "Types of Technical Market Indicators: Trend, Momentum, Sentiment," in *The Encyclopedia Of Technical Market Indicators, Second Edition*, 2nd ed. Two Penn Plaza, New York, USA: McGraw-Hill, 2003, ch. 1, pp. 7–8.
[2] B. Johnson, "Institutional trading types," in *Algorithmic Trading and DMA: An introduction to direct access trading strategies*. 4Myeloma Press, February 2010, ch. 1.4, pp. 8–10.
[3] R. J. Bauer and J. R. Dahlquist, *Technical Markets Indicators: Analysis & Performance (Wiley Trading)*, 1st ed. 605 Third Avenue, New York, NY 10158-0012: John Wiley & Sons, Inc., 1999.
[4] R. W. Colby, *The Encyclopedia Of Technical Market Indicators, Second Edition*, 2nd ed. Two Penn Plaza, New York, USA: McGraw-Hill, 2003.
[5] E. Botes and D. Siepman, "The Vortex Indicator," *Technical Analsysis of Stocks & Commodities*, January 2010.
[6] Y. A. Liu, *Systematic Program Design: From Clarity to Efficiency*. New York, NY, USA: Cambridge University Press, 2013.
[7] E. Kohler, "The Click Modular Router," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2001.

[8] R. J. Bauer and J. R. Dahlquist, "The Technical Analysis Controversy," in *Technical Markets Indicators: Analysis & Performance (Wiley Trading)*, 1st ed. 605 Third Avenue, New York, NY 10158-0012: John Wiley & Sons, Inc., 1999, ch. 1, pp. 7–8.

[9] R. W. Colby, "TRIX (triple exponential smoothing of the log of closing price)," in *The Encyclopedia Of Technical Market Indicators, Second Edition*, 2nd ed. Two Penn Plaza, New York, USA: McGraw-Hill, 2003, pp. 702–705.

[10] Y. A. Liu, "Recursion: iterate and incrementalize," in *Systematic Program Design: From Clarity to Efficiency*. Cambridge Universtiy Press, 2013, ch. 4, pp. 83–116.

[11] D. Gries and F. B. Schneider, "A theory of sequences," in *A Logical Approach to Discrete Math*. 185 Fifth Avenue, New York, NY 10010, USA: Springer-Verlag New York Inc., 1993, ch. 13, pp. 251–264.

[12] Y. A. Liu, "Conclusion," in *Systematic Program Design: From Clarity to Efficiency*. Cambridge Universtiy Press, 2013, ch. 7, pp. 187–212.

[13] R. W. Colby, "Directional Movement Index (DMI)," in *The Encyclopedia Of Technical Market Indicators, Second Edition*, 2nd ed. Two Penn Plaza, New York, USA: McGraw-Hill, 2003, pp. 212–217.

[14] C. M. Jones, "What do we know about high-frequency trading?" *SSRN Electronic Journal*, 03 2013.

[15] A. Zubayer, M. Musharraf, and R. Ahmed, "FFTI: Free Form Technical Indicator," in *2011 3rd International Conference on Computer Research and Development (ICCRD)*, vol. 1, 2011, pp. 87–91.

[16] H. Larkin, "A human readable language for stock market technical analysis on mobile devices," in *Proceedings of the 12th International Conference on Advances in Mobile Computing and Multimedia*, ser. MoMM '14. New York, NY, USA: ACM, 2014, pp. 132–136. [Online]. Available: http://doi.acm.org/10.1145/2684103.2684131

[17] "Tradestation," last accessed 18th of Feb., 2019. [Online]. Available: https://www.tradestation.com/

[18] "Multicharts," last accessed 18th of Feb., 2019. [Online]. Available: http://www.multicharts.com

[19] "Aiq systems," last accessed 18th of Feb., 2019. [Online]. Available: http://aiqsystems.com

[20] "Wave59," last accessed 18th of Feb., 2019. [Online]. Available: http://www.wave59.com