Using crytpocurrencies and LispTick to evaluate trade signing accuracy

Cédric Joulain
cedric.joulain@kereon-intelligence.com
cedric.joulain@gmail.com
https://kereon.lisptick.org

January 3, 2022

Abstract

This paper present a new tool called LispTick, processing any timeserie with a pure streaming approach. To show the full potential, we are implementing well known finance algorithms and use them as benchmark. This has the double advantage of showing the speed and versatility of our solution and validating implemented algorithms with real data from crypto-markets. Those algorithms are Tick Test, Mid Test, EMO (Ellis, Michaely and O'Hara) and Lee-Ready, they infer trade direction using intraday data. Each trade signing algorithm, inferring if trade is market buy or market sell using intraday data, is implemented with LispTick and validated with Bitstamp data. Bitstamp is a market place where you can trade some main cryptocurrencies and where real trade direction is part of the intraday data. Comparing the real direction with the inferred one shows accuracy for each method.

1 Introduction

Bitstamp is a European based cryptocurrency marketplace. Its web-socket API allows to retrieve lots of information for each trade and quote rarely available on standard market places. For example, for each trade we have timestamp, in μs since january 2019, price, amount... But we also have id, buy-order-id, sell-order-id and more important for our study: type, which is buy or sell. So having the real direction of the trade allows us to compute an accuracy score for each inferring method we will implement later.

LispTick is an extension of Lisp language with new embedded types like time series, time, durations... With a powerful internal synchronization mechanism you can add, subtract, multiply or do what you want with several input time series to create a new output time serie. Lisp is there just as an interface to describe the algorithm, this code message is then parsed by a service written in Go, Google open source programming language. Then and only then, the algorithm is evaluated as a stream by pure Go. This streaming approach allows LispTick to handle any size of time serie, even with billions of points. It also allows to work on tiny machines like Raspberry Pi or even smaller like Onion Omega.

All the following algorithms implemented with LispTick are validated against Bitstamp data. But they can be applied on any source, including classical exchanges like Eurex, Nasdaq, EuroNext, LSE...

2 Used data and definitions

Each method is implemented with 4 input arguments and result is a time series of signs., -1 for sell, 1 for buy and 0 for undefined.

Input arguments are:

source the data source, here Bitstamp.

code the instrument code, here "BTC" "ETH" "XRP" "LTC" "BCH" "EUR" "BTCEUR" "ETHEUR" "XRPEUR" "LTCEUR" "BCHEUR".

start the beginning date, here 2019-01-08 .

stop the ending date included, here 2021-12-31 .

Output sign is:

-1 for sell.

0 for unsigned, trade price exactly matching mid price in mid test for example.

1 for buy.

The full number of trades, bids and asks (bbo) for all currencies and the given period are:

trades: 96,374,510

bids: 2,727,299,774

asks: 2,727,299,774

For each, method we are counting how many trades are signed, name “cover”, how many trades have a 0 sign, name “no sign” and how many trade signs are correct, named “right sign”. Then we are able to compute percentage score using:

$$score = \frac{number\ of\ right\ sign}{number\ of\ trades}$$

For each, method we also try several different delays to move quotes in time. This is not done for Tick Test as it is only based on previous trade price and not on bid/ask.

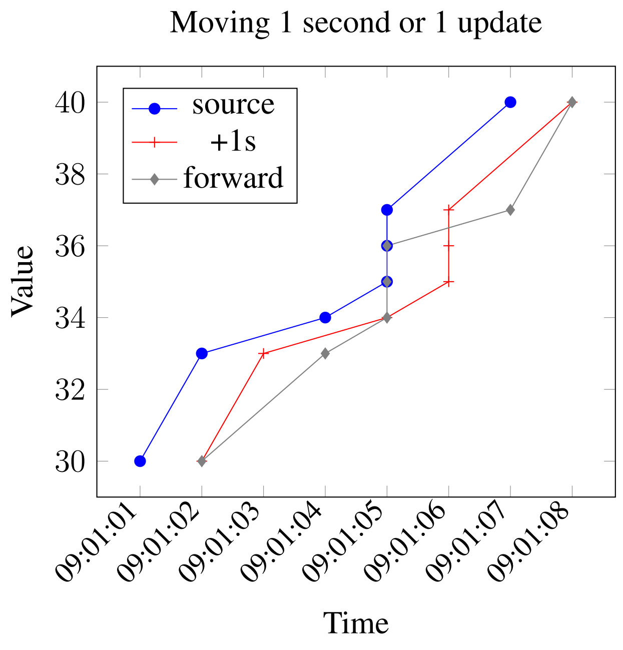

Figure 1: Different ways of moving in time

There are two different ways of moving quotes in time, adding the same delay to all timestamps or using previous or next update time.

As you can see in Figure 1 adding 1 second to each point will move uniformly all the time series.

Another moving option is to move forward, meaning using next timestamp for current value, move seems less uniform but it has the advantage to adapt to the underlying instrument liquidity. Moving by an absolute amount of time will not have the same impact on time-series depending on instrument liquidity. For a very liquid instrument, 1 second can be huge, for an unliquid one it's small. Using previous or next update time is intrinsically linked to instrument liquidity.

We test for each model delays from -2 seconds to 2 seconds, with 100 milliseconds step. We also test several update moves, backward (use previous update time), forward, 2x forward and 3x forward. Base and 0s being the same as original time series without any change.

All the Listings that follow are the exact LispTick code used to compute results. All words in bold are reserved LispTick keywords, others are user defined.

3 Tick Test

The tick test infers the direction of a trade by comparing its price to the price of the preceding trade(s). The test classifies each trade into four categories: an uptick, a downtick, a zero-uptick, and a zero-downtick. A trade is an uptick (downtick) if the price is higher (lower) than the price of the previous trade. When the price is the same as the previous trade (a zero tick), if the last

Listing 1: Tick Test LispTick implementation

```
(defn tick-test[source code start stop]
  (*
    (clockref
      (one
        (timeserie @trade-price source code start stop))
      0) ;reference value for sign when no delta price available
    (keep
      (sign
        (delta
          (timeserie @trade-price source code start stop)))
      != 0)))
```

model	cover	no sign	right sign	score
tick-test	96,357,953	16,557	73,935,691	76.72%

Table 1: Tick Test accuracy

price change was a downtick, then the trade is a zero-downtick. A trade is classified as buy, 1 for our time series, if it occurs on an uptick or zero-uptick; otherwise it is classified as a sell, -1 for our time series.

3.1 Implementation

LispTick implementation is quite compact, see Listing 1. We create a time series of not null sign from the delta of each price. We then multiply by a time series of 1 at each trade timestamp. Doing so each missing sign will be replaced by previous sign multiplied by 1...

3.2 Accuracy

Score obtained in Table 1 is in line with other studies like [3], where accuracy is estimated around 72%.

4 Mid Test

Mid test is based on middle price, $mid = \frac{best\,ask + best\,bid}{2}$, if trade price is higher (lower) this is a buy (sell) trade. Trade with price exactly matching mid can't be signed.

4.1 Implementation

LispTick implementation is also straightforward is this case, see Listing 2.

The only trick is to define trade price timeseries as a **`clockref`** for the subtraction so there will be only one update per trade and not one update for every new trade and mid.

We don't move quotes in time as user defined function `move-in-time` in Listing 2 does nothing.

Listing 2: Mid Test LispTick implementation

```
;Use synchronous quotes (unchanged)
(defn move-in-time[ts]
  ts)

(defn base-mid[source code start stop]
  (/
    (+
      (timeserie @bid-price source code start stop)
      (timeserie @ask-price source code start stop))
    2))

(defn mid-test[source code start stop]
  (sign
    (-
      (clockref ;only one point per trade
        (timeserie @trade-price source code start stop)
        0) ;reference value for sign when no quote available
      ;ask previous day quotes to match all trades
      (move-in-time
        (base-mid source code start stop)))))
```

4.2 Accuracy

LispTick computed results can be seen in Figure 2 for delay in number of updates and lag in seconds. In Figure 3, we can see scores split by currency. EUR have a different behavior perhaps because it's not a crypto and, on this place, it is less liquid, have far lower volatility with slighly higher spread.

Concerning quantity of covered trades, as starting quotes are 1 day in past coverage is always 100% of trades. We always have enough information to put a mid in-front of each trade price.

Around 1.4‰ of trades have a zero sign, labeled "no sign", meaning trade price was exactly equal to mid price. This percentage decrease as we move forward, i.e. as we use quotes from the past.

Best scores are obtained by using synchronous quotes, or with a small lag in future (100ms) when we use a time as delay.

5 Lee-Ready

Lee-Ready procedure [2], is based on both trade prices and quotes. First it uses what we call a "Side Test", if trade price is higher than (lower than) best ask (best bid) it is a buy (a sell). For trade with price between bid and ask it uses Tick Test result.

5.1 Side Test Implementation

Side Test LispTick implementation is done in 3 parts see Listing 3:

1st `ask-side` a time series of 1 when trade price is higher or equal to ask price and 0 other ways.

Listing 3: Side Test LispTick implementation

```
;Use synchronous quotes (unchanged)
(defn move-in-time[ts]
  ts)

(defn ask-side[source code start stop]
  (sign
    (+
      1 ; only prices < ask will have 0 sign, others 1
      (sign
        (-
          (clockref
            (timeserie @trade-price source code start stop)
            0) ;reference value for sign when no quote available
          (move-in-time
            (timeserie @ask-price source code start stop)))))))

(defn bid-side[source code start stop]
  (sign
    (+
      -1 ; only prices > bid will have 0 sign, others -1
      (sign
        (-
          (clockref
            (timeserie @trade-price source code start stop)
            0) ;reference value for sign when no quote available
          (move-in-time
            (timeserie @bid-price source code start stop)))))))

(defn side-test[source code start stop]
  (+
    (ask-side source code start stop)
    (bid-side source code start stop)))
```

model	cover	no sign	right sign	score
bwd	62,773,900	33,600,610	56,329,823	58.45%
base	87,782,179	8,592,331	85,734,343	88.96%
fwd	80,921,674	15,452,836	77,587,598	80.51%
2fwd	77,928,644	18,445,866	74,057,901	76.84%
3fwd	75,933,643	20,440,867	71,669,681	74.37%

Table 2: Side Test accuracy per delay

2nd `bid-side` a time series of -1 when trade price is lower or equal to bid price and 0 other ways.

3rd Then we sum both, so trade between bid and ask will keep a 0 sign.

Ask (bid) side is done by computing sign of $trade\,price - ask(bid)$ then we add 1 (-1) and once again we only keep sign.

So `ask-side` gives:

$trade > ask \rightarrow sign(sign(trade - ask) + 1) = sign(1 + 1) = 1$

$trade = ask \rightarrow sign(sign(trade - ask) + 1) = sign(0 + 1) = 1$

$trade < ask \rightarrow sign(sign(trade - ask) + 1) = sign(-1 + 1) = 0$

And `bid-side` gives:

$trade > bid \rightarrow sign(sign(trade - bid) + 1) = sign(1 - 1) = 0$

$trade = bid \rightarrow sign(sign(trade - bid) + 1) = sign(0 - 1) = -1$

$trade < bid \rightarrow sign(sign(trade - bid) + 1) = sign(-1 - 1) = -1$

Once again trade price time series are used as **`clockref`** to get one and only one point per trade update.

5.2 Side Test Accuracy

As expected, see Table 2, Side Test score is smaller than Mid Test as all trades between bid and ask are unsigned. The best score is also obtained using synchronous quotes (base).

5.3 Lee-Ready Implementation

Lee-Ready LispTick full implementation, see Listing 4, uses previously defined functions `tick-test` and `side-test`. We split `side-test` in two parts, unsigned $= 0$ and signed $! = 0$. We keep signed part as it is and we **`merge`** it with unsigned part replaced by `tick-test`. To do so, we sum unsigned trades with `tick-test` using **`clockref`** to get one and only one update per unsigned `side-test` and not an update for every `tick-test`.

Listing 4: Lee-Ready LispTick implementation

```
(defn lee-ready[source code start stop]
  (merge
    (keep
      (side-test source code start stop)
      != 0)
    (+
      (clockref
        (keep
          (side-test source code start stop)
          = 0))
      (tick-test source code start stop))))
```

Listing 5: Perfect Side Test LispTick implementation

```
;Use synchronous quotes (unchanged)
(defn move-in-time[ts]
  ts)

; -1 if trade = bid, 1 if trade = ask, 0 for all others
(defn perfect-side-test[source code start stop]
  (+ ;sign from ask plus sign from bid
    (+ 1
      (* -1 ; 1 if trade = ask 0 else
        (abs
          (sign
            (-
              (clockref ;only at trade time
                (timeserie @trade-price source code start stop)
                0) ;sign when no ask available
              (move-in-time
                (timeserie @ask-price source code start stop)))))))
    (+ -1 ; -1 if trade = bid 0 else
      (abs
        (sign
          (-
            (clockref ;only at trade time
              (timeserie @trade-price source code start stop)
              0) ;sign when no bid available
            (move-in-time
              (timeserie @bid-price source code start stop))))))))
```

Listing 6: EMO LispTick implementation

```
;Ellis, Michaely, and O'Hara
(defn emo-[source code start stop]
  (merge
    (keep ;use perfect-side if sign different from 0
      (perfect-side-test source code start stop)
      != 0)
    (+ ;use tick-test when perfect-side sign is 0
      (clockref
        (keep
          (perfect-side-test source code start stop)
          = 0))
      (tick-test source code start stop))))
```

5.4 Ellis, Michaely and O'Hara (EMO) Implementation

Ellis, Michaely and O'Hara procedure [1], is also based on both trade prices and quotes. First it uses what we call a "Perfect Side Test", if trade price is equal to best ask (best bid) it is a buy (a sell). For trade with price not perfectly equal to bid or ask it uses Tick Test result.

EMO LispTick full implementation, see Listings 5& 6, uses previously defined functions `tick-test` and `perfect-side-test`. We split `perfect-side-test` in two parts, unsigned $= 0$ and signed $! = 0$. We keep signed part as it is and we **`merge`** it with unsigned part replaced by `tick-test`. To do so we sum unsigned trades with `tick-test` using **`clockref`** to get one and only one update per unsigned `perfect-side-test` and not an update for every `tick-test`.

5.5 Performance as a function of lag

Lee and Ready [2] observe that the current quote is often not synchronized to the trades, so they suggest to compare the trade price to the quote 5 seconds before. Doing this with LispTick is as easy as adding 5s to the time series, meaning adding 5s to the timestamp of the time series.

This 5 seconds lag from Lee-Ready was purely empirical, so we tried several lags to find the best for our data. In our case the best lag is 0ms, so no lag, for all. Bitstamp trades and quotes seem to be very well synchronized, that was not the case before January 2019 where best lags were -100ms for Lee-Ready and Mid-Test, and -200ms for EMO.

Our results for Lee-Ready score in Figure 2 are in line with Ioane Muni Toke study [4] Section 5, our graphs have an opposite direction as we add time instead of subtracting it. We have the same shape, performance drops quite fast on one side and more progressively on the other. Absolute maximum is more in line with recent studies than older ones.

As LispTick can move time series by a certain number of updates we also try several update lags, see Figure 2.

6 LispTick more in-depth

We present here more technical information about LispTick. Thanks to streaming, LispTick is very frugal with memory resources: it works on a Raspberry Pi Zero, a small single-board computer costing 17€. It's so frugal that we are able to make it work on an Onion Omega2, a $20 64MB nano-computer.

6.1 Computing Score

In Listing 7, you can see how score computation is done calling `right-sign`. Inferred sign, `side` function being `tick-test`, `mid-test`, `EMO` or `lee-ready`, is multiplied by the real sign coming from source time series **`@sign`**. Then we keep result of multiplication only when value > 0, so only when signs are identical and not equal to 0. Final result is the length (**`len`**) of this time series, the score of this method will by this number divided by full number of trades.

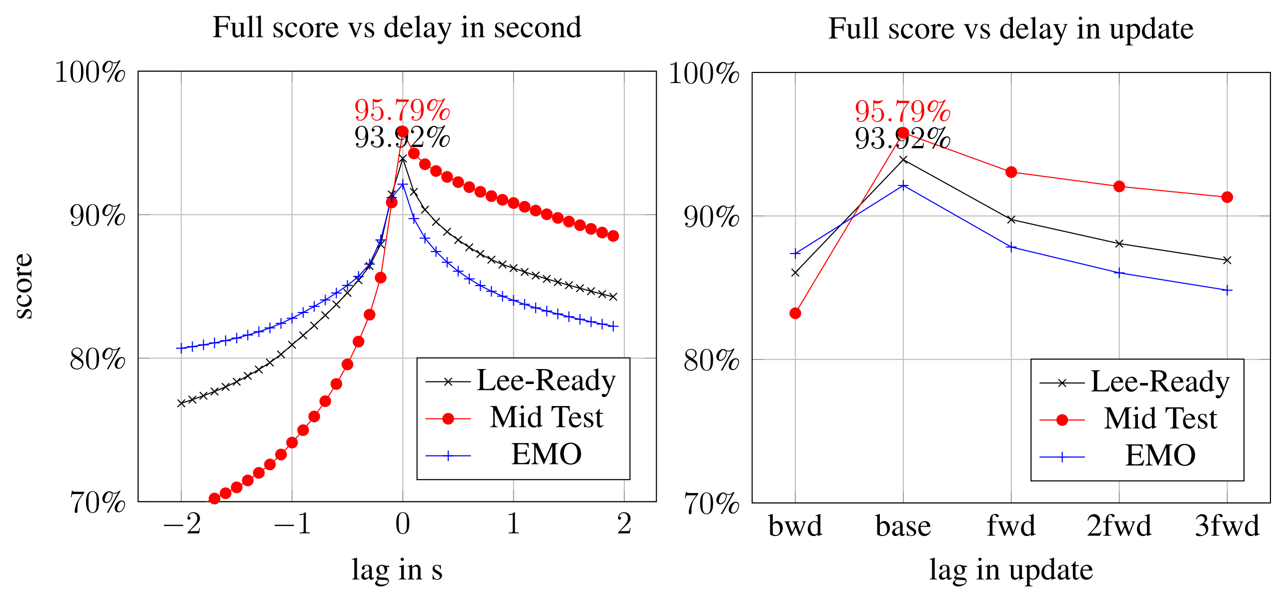

Figure 2: Lee-Ready vs Mid Test vs EMO

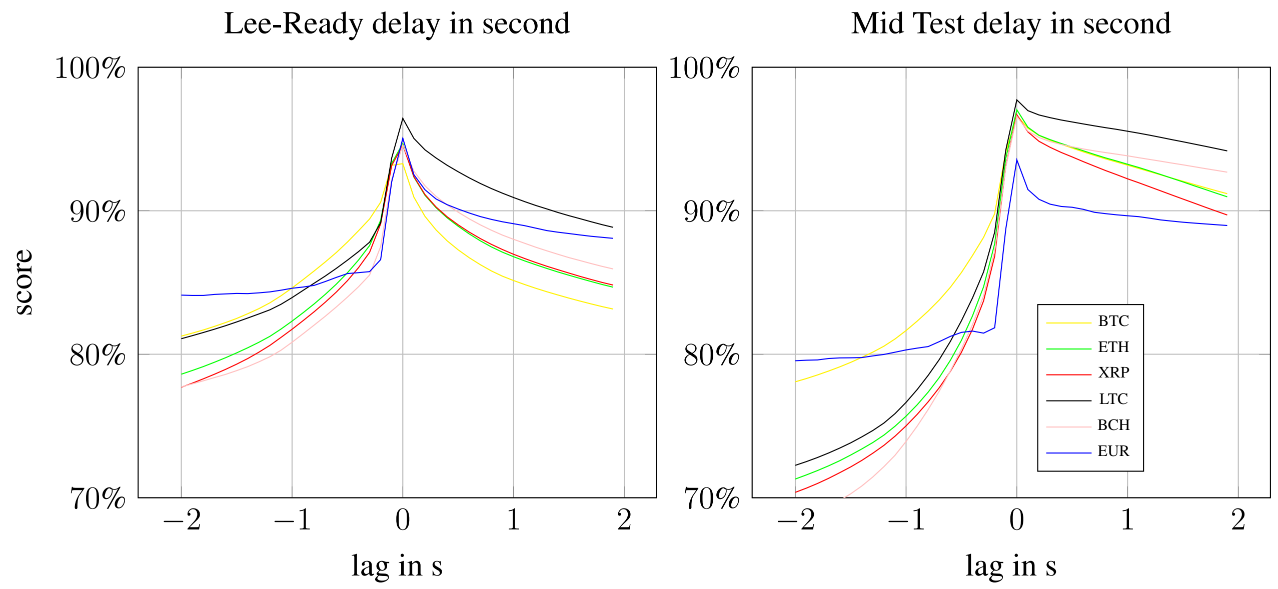

Figure 3: Lee-Ready & Mid Test score for each currency

Listing 7: Score LispTick implementation

```
;sign stats for one code one date
(defn sign-stats[source lag code dt]
  (hist
    (*
      (clockref
        (side source lag code dt dt))
      (* -1 (timeserie @"sign" source code dt dt)))))

;simple function giving size of timeserie
(defn nbr[field d code]
  (len
    (timeserie field "bitstamp" code d)))
```

Listing 8: Parallel score LispTick implementation

```
(map-reduce nbr-right sum dates codes)
```

Similarly number of unsigned trades is computed by counting number of signs $= 0$ by the function `no-sign`.

6.2 map-reduce

To benefit from the increasing number of cores in our today processors LispTick has implicit and explicit parallelization mechanisms.

In our case, we compute score day by day and sum each day using all cores, to do so we use **`map-reduce`** and a function to compute one day score, see Listing 8&7.

How it works: first argument of **`map-reduce`** is the function that will be called several times, here `nbr-right` counting number of rightly sign for one day one code see Listing 8. Then we define what to do with each value, here **`sum`** all values. At the end, we have the arguments to iterate on: all the dates and all the codes.

7 Benchmarks

To get numbers for our benchmark we had to do several rounds, looping several times over data and doing the same computation. Each loop is mentioned afterward as round, we keep the value form the fastest round.

The first round was slower as data was not in RAM OS file cache.

In Table 3, you can see number of trades signed per second depending on method and computer. In all cases, data is in OS cache as we forget 1st run time. Those are best results when cpus are efficiently cooled. Some CPUs may suffer thermal throttling as LispTick maximize all cores usage. After several seconds, frequency will decrease about 20% so same decrease will apply to number of signed trades.

The AMD Ryzen Threadripper 1950X is a 64-bit microprocessor with 16 cores and 32 threads (SMT enabled) and a frequency of 3.0Ghz. Benchmarks are 20 to 30% slower when disabling SMT, Simultaneous multithreading,

model	Tick-Test	Mid-Test	Lee Ready
amd1950X	12,400,000	2,070,000	895,000
i9-7900X	10,900,000	1,640,000	730,000
i7-11800H	9,230,000	1,370,000	616,000
Ryzen 7 2700X	8,990,000	1,410,000	622,000
Xeon Gold 5218R	6,170,000	1,010,000	468,000
Apple M1	6,010,000	927,000	408,000
E3-1270v6	4,950,000	713,000	308,000
2xE5530	4,720,000	713,000	305,000
Celeron J3455	1,410,000	221,000	99,200
i3-7100U	1,160,000	197,000	82,800
RPi-4	764,000	118,000	52,300
RPi-3	225,000	32,800	13,600
RPi-Zero 2W	176,000	23,700	10,300

Table 3: Signing speed, in trades per second, the more the better

The Intel Core i9-7900X is a 64-bit microprocessor with 10 cores and 20 threads (Hyper-Threading enabled) and a frequency of 4.0Ghz. Benchmarks are 30% slower when disabling Hyper-Threading.

The Intel Core i7-11800H is a laptop 64-bit microprocessor with 8 cores and 16 threads (Hyper-Threading enabled) and a frequency of 2.3Ghz with a 4.6Ghz boost.

The Intel Xeon Gold 5218R is a 64-bit server microprocessor with 8 cores and a frequency of 2.1Ghz. This is a VPS courtesy of Datacampus.fr. Servers are submerged into mineral oil, a far more sustainable cooling solution.

The AMD Ryzen 7 2700X is a 64-bit microprocessor with 8 cores and 16 threads (SMT enabled) and a frequency of 3.7Ghz.

The Apple M1 is a MacBook Air laptop 64-bit microprocessor with 8 cores and a frequency of 2.4Ghz.

The Intel Xeon E3-1270 v6 is a 64-bit server microprocessor with 4 cores and 8 threads with a frequency of 3.8Ghz.

The Intel Xeon 2xE5530 is a 64-bit bi-microprocessors server with 2x4 cores and 2x8 threads with a frequency of 2.4Ghz.

The Intel Celeron J3455 is a Synology DS918+ NAS 64-bit microprocessor with 4 cores and 4 threads and a frequency of 1.5Ghz. This shows how easy it is to install LispTick, even directly on a NAS.

The Intel Core i3-7100U is a laptop 64-bit microprocessor with 2 cores and 4 threads and a frequency of 2.4Ghz.

The RPi-4 ARM Cortex-A72 is a 64-bit microprocessor with 4 cores and a frequency of 1.5Ghz.

The RPi-3 ARM Cortex-A53 is a 64-bit microprocessor with 4 cores and a frequency of 1.2Ghz.

The RPi-Zero 2W ARM Cortex-A53 is a 64-bit microprocessor with 4 cores and a frequency of 1Ghz.

7.1 Thermal Throttling

Most of our machines suffer from thermal throttling, with a loss of about 20% when CPU was hot. Only server processors and Ryzen 7 did not suffer any slowness.

We had a major issue with the Intel i9, it was unable to sustain full speed and if the weather was hot, even with water cooling, we had a full reset!

7.2 Linux vs Windows

We were able to test LispTick on two different OS with the exact same machine, an Asus TUF F17-FX706HM-TUF766HM. A dual boot with Windows and Ubuntu allowed us to compare speed and OS file cache policy in real. The processor is the Intel i7-11800H, the best result with Windows was -10% slower than with Linux.

Another very interesting fact is how file cache is handled by both OS. On Linux, wherever you data is: USB stick, HDD, SSD, after only "one round" all data is cached in RAM and the speed is maximum. On Windows the behavior is quite different.

With data on a USB stick we never reached full speed but only a third of it. To get the maximum speed, i.e. -10% less than Linux, we had to copy all the data on the SSD and wait for several rounds to reach the maximum.

7.3 Dockerization

All our benchmarks are done with a single standalone executable. By curiosity and to bench cloud platform where the only possible access was container, we tried to Docker our benchmark.

We first benchmark docker vs no docker with a CPUs intense task. Results are inline with what we can see in literacy, there is no noticeable difference in speed.

We then test our LispTick benchmark with a Docker version. On the exact same machine, the Docker version is always slower. Differences are between -8% for a 4 cores RPi400 to -50% for a 16 cores AMD Threadripper. Speed loss seems to be linked with the number of cores. 8 cores machines, like Apple M1 2021 or Ryzen 7, lose around -25% and a 10 cores machine like Intel i9 lose -37%.

We did not explore further this behavior as our tool is very easily installable on any machine without Docker. But we guess that those discrepancies in performance are coming from the different levels of caches: file OS, CPUs...

8 Conclusion

With a tool like LispTick, it is very simple to implement and test all methods. The accuracy found empirically for Lee-Ready is in line with previous theoretical studies. But compared to a simple Mid-Test, when trade and quotes are recorded with an accurate timestamp, accuracy is smaller and computing complexity increases. Tick test is very fast but is only 77% accurate. Ellis, Michaely and O'Hara method is as complex and as accurate as Lee-Ready but less sensitive to time lag.

When Bitstamp timestamps were only precise to second Lee-Ready was marginaly more accurate, but this as changed with μs.

We will try to have access to other sources of data than Bitstamp to refine results. We are currently looking at conventional (not crypto) marketplaces with embedded sign if exists. It will be straightforward to test, just change the data source and instrument code!

References

[1] Katrina Ellis, Roni Michaely, and Maureen O'Hara. The accuracy of trade classification rules: Evidence from nasdaq. *Journal of Financial and Quantitative Analysis*, 35(4):529–551, 2000.

[2] Charles Lee and Mark Ready. Inferring trade direction from intraday data. *Journal of Finance*, 46:733–747, 01 1991.

[3] Marcelo Perlin, Chris Brooks, and Alfonso Dufour. On the performance of the tick test. *The Quarterly Review of Economics and Finance*, 54(1):42 – 50, 2014.

[4] Ioane Muni Toke. Reconstruction of order flows using aggregated data. *Market Microstructure and Liquidity*, 02(02):1650007, 2016.