Backtest Booster: A self-adaptive solution for accelerating trading strategies using JIT

Yuran Fang
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
y7fang@uwaterloo.ca

Shan Xue
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
s2xue@uwaterloo.ca

***Abstract*—This self-adaptive system (SAS) serves as a prototype to explore the possibility of accelerating the execution time of trading strategies using a Just-In-Time compiler (JIT). The system offers a solution using architecture-based adaptation, with the assist of Executable Runtime Megamodels (EUREMA). Through experiment, the system has effectively accelerated the execution time of certain trading strategies, meanwhile using reasonable amount of memory.**

Keywords Self-adaptive system, Just-in-time compilation, Execution time acceleration, Architecture based adaptation, Runtime models

I. Introduction

A. *What is Backtesting?*

Over the years, trading bots are become more and more commonly used for traders due to its ability to complete a settlement in a very short amount of time with very high precision. Therefore, the lucrativeness and robustness of trading strategy has become increasingly important for traders to evaluate. However, it will be very costly if the trading strategies are tested through real time stock values. Backtesting is one common way for traders to evaluate the viability of their trading strategy before it is executed in real time stock market.

Backtesting utilizes the historical data of a certain stock, the trading strategy will be run against the stock historical data, and it will generate results on how the trading strategy performs. This simulation process allows the traders to analyze its risk and profitability of the trading strategy [1].

An Ideal form of backtesting chooses sample stock data from a certain time-period that is relevant to the trading strategy and reflects a variety of market conditions. It is also very important that the stock data should represent a company that still have significant relevancy in today's market. The stock of companies which are backrupt, sold, or even liquidated may not yield results that have enough relevancy for evaluations [1].

A backtest should consider all trading costs, even if the costs are insignificant. The summation of different trading costs over a period of backtesting can greatly performance results of the trading strategy and the evaluations later on [1].

In all, this project aims to speed up the execution time of the backtesting strategy meanwhile not affecting the performance of the trading strategy. The reasoning of this will be further elaborated in the section of methodology.

B. *What is Crossover Strategy?*

The system will be solely focusing on accelerating the execution time of the crossover strategy.

The crossover strategy is a very common trading strategy for the trader to use during backtesting. Crossover strategy involves a short-term moving average and long-term moving average. The short-term moving average means the average of stock price values over a short period of time, and long-term moving average means the average stock price values over a long period of time. The "short-term" or "long-term" refer to the length of time the average price values are calculated over. Those two terms can also be referred to as "fast period" and "slow period" [2].

Once the short-term moving average and long-term moving average overlaps with each other. It's called a "crossover". Once the "crossover" event happens, it acts as an "indicator" to predict the future trend of the stock price value. There are three types of commonly used "indicator" for crossover strategy: simple moving average indicator, weighted moving average indicator, and exponential moving average indicator.

C. *What is Backtrader ?*

Backtrader is a commonly used python package for traders to write their own trading strategy and run their strategies as simulations over historical stock data. There are several python classes of the backtrader's package that are very essential for this self-adaptive system. For the following descriptions, "bt" is the acronym for the backtrader.

- **bt.Feeds:** bt. Feeds offers a service for the traders to upload the historical stock data and specify the time-period that the traders want to test on. The trader can either upload the .csv file of a stock by themselves, or they can use a specific interface to retrieve the historical stock data from the yahooFiance platform.
- **bt.Strategy:** For each strategy that the traders implement, it needs to be encapsulated within the bt.Strategy class. This class includes the algorithmic operations that will perform the buy and sell actions over different historical stock data.
- **bt.Cerebo:** This is simulator class which can take in the bt.Strategy and the historical stock data from bt.Feeds as inputs. The simulator will set a commission at first, and then the simulator will run over the historical stock data. Eventually, the

simulator can also generate different results for the traders to further evaluate the performance of their strategies. For optimizing trading strategies, the Cerebo can include an optimizer module which can apply different parameters that can affect on the performance of the trading strategy. As mentioned earlier, the crossover strategy needs to specify the slow period and fast period. In backtrader, the traders can specify a list of slow and fast periods. The backtrader allows the traders to run their crossover strategy through all combinations of fast and slow periods and find some combinations of fast and slow periods that might be lucrative in the real market. Furthermore, the backtrader package also allows the traders to specify the type of indicators without writing the indicator's logic manually.

There are two other optimization parameters that can be added to Cerebo simulator, "optreturn" and "optdata". These two parameters are the build-in parameter which backtrader implements with aim of optimizing the execution time of the strategy and usage of memory.

"optdata" will load the historical stock data beforehand the execution of the trading strategy, instead of allowing the strategy to run along with reading the historical stock data. This parameter will decrease the execution time of the strategy meanwhile possibly leads the results of the strategy less accurate.

"optreturn" will selectively return the results that are possibly the most "relevant" to the performance of strategy. In other words, not all results that are generated by the trading strategy will be returned. This parameter will also possibly lead to inaccurate results when evaluating the performance of the trading strategy.

These two parameters will be set to "true" by default, when using backtrader. However, if the traders want to set these two paramters to "false", they need to specify those two parameters as "no-optreturn" and "no-optdata".

D. *What is Just-In-Time (JIT) Compilation?*

In recent decades, due to the development of computer programming language, many modern programming languages now support just-in-time compilation such as Java, Python, JavaScript, etc. The purpose of using just-in-time compilation is to speed up the execution of program. For a programming language that support just-in-time compilation, there exists a just-in-time compiler can translate the frequently executed part of the program into efficient machine code. It is a well-known fact that, the speed of execution of machine code is faster than the execution which in default interpreted mode. For example, the execution speed of a C program is faster than the similar program written in Python, without loss of generality.

There are three types of program executions [3]. The first type is the program executing on top of the operating systems, also can be seen as native execution mode, and there are programming languages such as C and C++. The second type is that executing source code by the interpreters, and such programming languages are Python and JavaScript. The third type is that the executing compiled artifacts on the virtual machines, such programming languages are Java and C#. Interpreters and virtual machines are called interpreted mode. The program execution in native execution mode is faster than the program executed in interpreted mode. The reason is that the interpreted mode contains multiple additional layers than native execution mode. Therefore, the just-in-time compilation is used to translate the frequently executed code to machine code for operation systems, then to realize the better performance on speed during runtime.

There are two ways to do the just-in-time compiler, one is method-based just-in-time compiler, another one is trace-based just-in-time compiler [3]. As commonly accepted, the hot code is a snippet of codes that executed frequently. If there exists a method that need to be executed many times, then it is a hot method. Moreover, if there exists a code path that will be executed multiple times, then it is a hot path. For method-based JIT compiling, the hot method will be entirely transformed to machine code. For example, Hotspot uses the method-based just-in-time compiling, it is an implementation of Oracle of the Java Virtual Machine. The trace-based JIT compiling, the hot path will be transformed to machine code. PyPy uses the trace-based JIT compiling.

E. *What is PyPy?*

Different programming languages have different implementations. Java has its implementation for just-in-time compiler, and the implementation provides just-in-time configuration parameter. Moreover, PyPy is an implementation of the Python programming language [4]. PyPy often runs faster than the standard execution of CPython because of the usage of just-in-time compiler. PyPy provides a list of just-in-time configuration parameters. These configuration parameters can indicate the approach to translate the executed code. For instance, *threshold* is a configuration parameter of PyPy, and it is used to define the number of times a loop has to be run before it can be considered as the frequently executed code that need to be translated by just-in-time compiler. The default value of *threshold* is 1039, which means after 1039 times a loop has to run for it to become hot. These constants are tuned carefully by PyPy team.

Furthermore, for another example, the official document of *trace_eagerness* is described as number of times a guard has to fail before we start compiling a bridge [4]. In order to understand the guard and bridge, we need to explain these terms in the context of JIT compilation. In JIT compiler of PyPy, a guard is used to ensure assumptions made during optimization remain valid. If a guard check fails, the JIT compiler has to abandon the optimized code path and fall back to a more general, unoptimized interpretation. A bridge is a compiled code path that starts from where a guard failed and continues to the end of the loop or function. It is a way for the JIT compiler to solve this kind of situations, where the initial assumptions no longer hold. Therefore, the parameter *trace_eagerness* refers to a threshold to decide when to compile a new code path. If a guard fails often enough, the JIT compiler of PyPy might decide to compile the new code path to optimize it as well. This new path is called a bridge [5].

In this report, we will explain these configuration parameters with piece of codes to show the usage in PyPy and its performance.

```
def loop(n):
  exp = 2:
    for I in range(n):
```

```
4.        exp = exp ** 2
```
Code snippet 1: a function does exponential calculation.

Given the above code snippet, we can write a Python3 program to compare the performance between Python default implementation and PyPy. In the case of *loop(30)*, by using the Python default implementation, the result of execution time is around *7029 milliseconds*, and its memory usage is around *530 MiB*. Moreover, if we use PyPy and set a JIT configuration parameter *threshold=22*, then the result of execution time is around *2786 milliseconds*, and its memory usage is around *1213 MiB*. From the outcomes, we know that PyPy speed up the execution time, and it is faster than Python default implementation. However, PyPy uses more memory. For this code snippet, the memory usage of PyPy is about 2 times greater than Python default implementation.

The default value of threshold is set to 1000. In the context of this self-adaptive system, since a "lightweight" crossover strategy can be run over the historical stock data of 50 days with 10 fast and slow periods combinations, a "lightweight" crossover strategy can include around 50*10 = 500 iterations. Therefore, in our adaptation plan the threshold is always set to 500 for the PyPy execution.

II. UNCERTAINTY & KNOWLEDGE BASE

A. *Data preprocessing*

The self-adaptive system takes advantage of the knowledge base which includes the data that describe the execution of certain strategy. We have come up with several data elements, or parameters, that can be used to factorize a certain strategy execution. Table 1 will describe the data elements that will be used to describe the strategy used by the traders.

Element Name	Description	Data Type
Indicator Type	The indicator used by the traders: It can be either be simple moving average indicator, weighted moving average, and exponential moving average. These three indicators will be encoded as "1", "2", and "3".	Integer
Time Span(in days)	Time period between two dates of the a certain stock. For example, if the strategy aims to simulate the stock histrocial data between 2020-9-21 to 2021-9-21, the time span will be 365 days, and the element will be 365.	Integer
Fast period lower bound	The lower bound of the fast period, for example it can be 5 days, and the element will be 5.	Integer
Fast period upper bound	The upper bound of the fast period, for example it can be 10 days, and the element will be 10. The fast period lower bound and upper bound will describe the range of fast periods. In this case it will be (5, 10).	Integer
Slow period lower bound	Same idea as fast period. Slow period lower bound usually is greater than fast period upper bound.	Integer
Slow period upper bound	Same idea as fast period.	Integer
Slow period steps	Same idea as fast period.	Integer
Number of combinations of fast and slow periods.	The number of all possible combinations of fast and slow periods for the given fast and slow period range. If the fast period range includes 4 values and slow period range includes 3 values, the total possible combinaitons are 4x3 = 12. The element will be 12	Integer
no-optreturn applied	If the trader applies the parameter, "no-optreturn", it will be marked as 1, otherwise it will be 0.	Integer
no-optdata applied	If the trader applies the parameter, "no-optdata", it will be marked as 1, otherwise it will be 0.	Integer

Table 1: Data that describes the trading strategy.

B. *Uncertainty*

As mentioned earlier, the self-adaptive system solely focuses on accelerating crossover strategies. In other words, we assume that the strategies which will be used by the traders are all crossover strategies. The previous section has explained how the crossover strategies can be factorized into the elements in Table 1. Moreover, the value of these elements is completely up to the trader's input of their crossover strategies. Therefore, the uncertainty is the specific that the traders will simulate, and the uncertainty can be factorized into the elements of Table 1.

C. *Machine Learning Model*

The self-adaptive system utilizes a feed-forward machine learning model to predict the following two values listed in Table2.

Predicted value name	Description	Data type
Execution time (in seconds)	The time it takes to execute the strategy	Float(decimal)
Memory used (in megabytes)	The maximum amount of memory used while executing the strategy.	Integer

Table 2: Values predicted by the machine learning model.

When executing the strategy, the self-adaptive system can either pick to use the original CPython interpreter, or the PyPy JIT complier. To predict the two values listed in Table 2, the machine learning model does not only take in the elements of Table 1 as input but also the information of whether the strategy is executed using the CPython interpreter or the PyPy JIT compiler. Therefore, we add another element as the machine learning model's input, where the execution using CPython is encoded as 0 and the execution using PyPy JIT compiler as 1.

In the introduction section, it has shown that PyPy will generally use more memory, and it takes less time to execute the strategy. When using both CPython interpreter and PyPy JIT compiler for the exact same strategy, the machine learning model should be able to make the prediction with major difference between the two execution methods.

Considering the great difference that the execution method will make, the machine learning model's structure is described in Table 3.

Layer Name	Layer Type	Dimension (# of neurons)	Previous Layer
Input_Layer_1	Input layer	11	None
Input_Layer_2	Input layer	1	None
Dense_Layer_1	Hidden layer	20	Input_Layer_1
Dense_Layer_2	Hidden layer	10	Input_Layer_2
Dense_Layer_3	Hidden layer	10	Dense_Layer_1
Dense_Layer_4	Hidden layer	5	Dense_Layer_2
Outpu_Layer	Output layer	2	Dense_Layer_3, Dense_Layer_4

Table 3: Machine Learning Model Structure

All the layers of the machine learning model are fully connected layers. The machine learning model has two input layers, one takes the input of all elements in Table 1, the other one takes in the encoded value of execution method (1 or 0). Each input layers will be processed by two hidden layers respectively, and the output neurons of the second hidden layers for both input layers will be concatenated and forward into the output layer, which have two output neurons.

Before the actual adaptation begins, the machine learning is trained as a base model to make reasonable predictions. The base model requires two groups of data as features. One is the elements of Table 1. The other is the encoded value of execution method, 0 or 1. The labels should be the value of execution time and memory usage after the actual execution. To train model, 1000 random crossover strategies are generated, and they will be factorized and executed with execution method randomly picked. Eventually, the actual execution time and memory usage will be documented. Through this workflow, the dataset of training data with 1000 data entries can be formed, and the base model, which will be referred as cycle_0_model, will be trained. Figure 1 illustrates the entire workflow for training the cycle_0_model.

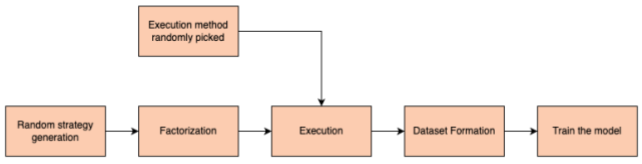

Figure 1: Workflow of training cycle_0_model

The model is trained over 100 epochs, where the loss function of the predicted values and actual values is Mean Absolute Errors (MAE). Since the output has two dimensions, the MAE is calculated using the following formula respectively for execution time and memory usage:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Where:

- n is the number of observations of the dataset.
- Y_i is the actual value of either the execution time or memory usage.
- $Y^\wedge_i$ is the predicted value of the execution time or memory usage.

The eventual MAE is the average of the MAE for execution time and the MAE for memory usage.

D. *Utility Function*

The utility function is responsible for providing a reference for the self-adaptive system to decide which adaptation plan to use. The utility function is given as follows:

$$Utility = 80 * e^{-P\_exec\_time} + 20 * e^{-P\_memory\_usage}$$

Where:

- P_exec_time represents the value of execution time in seconds
- P_memory_usage represents the value of memory usage in megabytes.

Because both P_exec_time and P_memory_usage are positive, and e^{-x} ranges between 0 and 1 for a positive number, the value of utility will range from 0 to 100.

III. Methodology

The self-adaptive system has two types of adaptation cycles.

- For each strategy provided by the traders, the system needs to provide an adaptation plan, which is the execution method that makes a great trade-off between the execution time and memory usage. This adaptation cycle will be referred as “general adaptation cycle”.
- When the monitor module of the runtime model can gather 50 pieces of data of actual executions, the model will be re-trained using the newly collected data and evaluated if it’s worthy to update model. This adaptation cycle will be referred as “model adaptation cycle”.

The two types of adaptation cycle will be further elaborated within the section of architecture-based adaptation and runtime model.

A. *Architecture Based Adaptation*

On a micro level, for each strategy entered by the traders, the traders’ strategy will be factorized into the elements that is included in Table 1. The factorization will be formed as one part of the input, the other part of the input is the encoded value of the execution method, 0 or 1. However, instead of selecting one encoded value, both values will be used by the current machine learning model, along with the factorized input from the strategy, to predict the values of execution time and memory usage. After the execution time and memory usage are predicted for both execution methods, the predicted values are entered respectively to the utility function.

Comparing both outputs of utility function, the execution method that helps to generate the higher utility function, will be selected as the adaptation plan. Finally, the selected execution method will be used to execute the strategy provided by the traders. Figure 2 shows this workflow from the micro perspective.

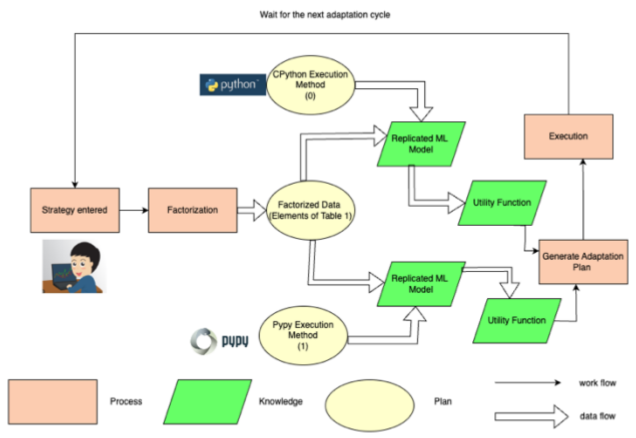

Figure 2: Micro workflow for the general adaptation cycle

On a macro level, the self-adaptive system takes advantage of the three-layer architecture model [6].

The goal management layer consists the current machine learning model for the general adaptation cycle. For a certain strategy, the machine learning model will predict the execution time and memory usage using either execution methods. The predicted values will be later entered into the utility function. From the outputs of the utility function, the comparison will be used to select the adaptaion goal and the corresponding adaptation plan. Assuming that the output from the prediction values of using CPython interpreter is the value of "utility 0" and the output from the prediction values of using Pypy JIT compiler is the value of "utility 1". Here are the two adaptation goals from comparing the two outputs from utlity function:

- Both adaptaion goals focus on the self-optimization property.
- Adaptation Goal 1: If utility 0 >= utility 1, the execution method should focuses on using less memory.
- Adaptation Goal 2: If utlity 0 < utility 1, the exuection method should focuses on saving more time for execution.

The change management layer actualizes the entire MAPE-K feedback loop. Through the MAPE-K feedback loop, a managing system is built to sense the input of strategy from traders, eventually generate the concerete adaptation plan, and actuate the execution in the managed system using the corresponding adaptation plan. The adaptation plans are listed as follows:

- Adaptation plan 1: Executing the strategy using CPython interpreter. This plan will picked when the current adaptation goal is the adaptation goal 1.
- Adptation plan 2: Executing the strategy using Pypy JIT compiler. This plan will picked when the current adaptation goal is the adaptation goal 2.

To actualize the MAPE-K feedback loop, here are reponsibility of each module:

- Monitor module: It's responsible to complete the data pre-processing and data collection. Specifically, the monitor module factorizes the strategy into the elements listed in Table 1. The factorized data is passed to the analyzer module. Moreover, after the actual execution is done, the factorized data for the strategy, the execution method used, and the actual execution time with memory used will be documented.
- Analyzer module: It's responsible to use the machine learning model to predict the execution time and memory usage from the factorized data of the strategy with different the encoded values for execution method. Furthermore, the predicted values will be passed to the utility function of the goal management layer.
- Planner module: It's responsible to take the adaptation goal from the goal management layer, and select the corresponding adaptation plan, and pass it to the executor module.
- The executor module is responsible to run the actual command based on the adaptation plan, and pass the encoded value of execution method selected and the actual execution time and memory usage to the monitor module.
- The knowledge module is the current machine learning model used for this general adaptation cycle.

The component control layer consists of the actual strategy provided by the traders, and also the running environment that the strategy is planned to be run on. The component change layer will eventually actualize the adaptation plan.

To sum up, figure 3 generalizes the entire genral adaptation cycle's work flow from the three-layer architecture perspective.

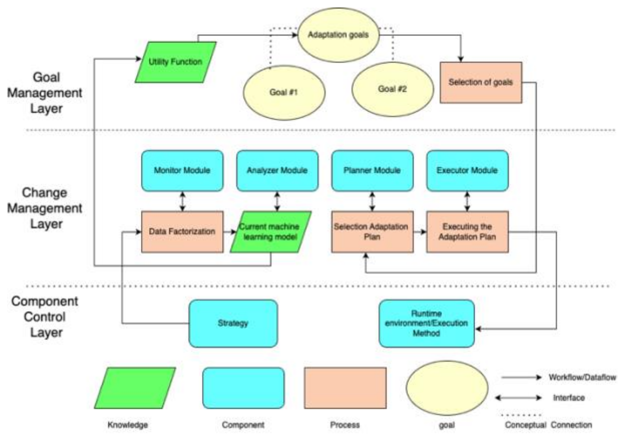

Figure 3: Macro workflow for the general adaptation cycle

B. *Runtime Models(EUREMA)*

The self-adaptive system takes advantage of the Executable Runtime Megamodel to actualize the model adaptation cycle. The Megamodel consists of the machine learning models used for the current and previous model adaptation cycles, and it also consists of the models to monitor, evaluate, change, and execute the machine learning model that should be used by the current model adaptation cycle.

The model adaptation cycle repeats for every 50 general adaptation cycles. In other words, after 50 general adaptation cycles, the self-adaptive system needs to update the weights of its current model. To complete this model adaptation cycle, the megamodel needs to have the monitor model, the evaluate model, the change model, and the execute model. The responsibilities of these models are listed as follows:

- The monitor model is responsible to collect the data from 50 general adaptation cycles, or 50 executions of different strategies. These data will form a new training dataset and testing dataset for the current machine learning model to furtherly train to update its weights. Specifically, the dataset is split into the training dataset with 40 pieces of data from different executions, and the testing dataset with 10 pieces of data from different executions.
- The evaluation model is responsible to update the weights of the current machine learning model, and then it will check the performance of the newly trained machine learning model and the current machine learning model. Specifically, a replication of the current machine learning model will be made, and the replicated model will be furtherly trained by the training dataset provided by the monitoring model. After the training, the newly trained model will predict the execution time and memory usage using the testing dataset and eventually calculating the MAE. Meanwhile, the original machine learning model will also predict the execution time and memory usage using the testing dataset and calculating the MAE. The values of the MAEs calculated from the current machine learning model and the newly trained machine learning model will be passed to the change model.
- The change model is responsible for making the decision that whether the newly trained machine learning model should replace the current machine learning model to be used as the knowledge module in the general adaptation cycle. Specifically, if the MAE of the newly trained machine learning model is less than the MAE of the current machine model by 5%, the current machine learning model will be replaced by the newly trained machine learning model. "Improvement" is the mathematical term that will be used to describe whether the current machine learning model should be replaced or not. Let MAE1 be the MAE for the current machine leaning model and MAE2 be the MAE for the newly trained machine learning model:

$$Improvement = \frac{MAE1 - MAE2}{MAE1}$$

  If the improvement is greater than 0.05, or 5%, the current machine learning model will be replaced by the newly trained machine learning model.
- The execution model is responsible to load the newly trained model if the current machine learning model needs to be replaced. Otherwise, the current machine learning model will be kept for usage until the next model adaptation cycle begins.

The workflow of the runtime models, or model adaptation cycle, updates the knowledge module of the MAPE-K feedback loop of the general adaptation cycle, and the updated knowledge module allows the prediction of the execution time and memory usage to be more accurate. Furthermore, the updated knowledge module improves the self-optimization property of the self-adaptive system. Figure 4 generalizes the workflow the EUREMA runtime models.

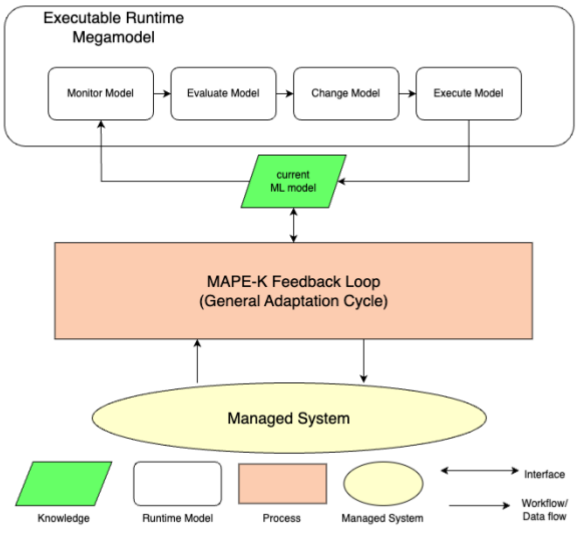

Figure 4: Workflow of EUREMA runtime models

IV. Results

For the experiment session, as mentioned in the introduction section, the self-adaptive system will only focus on the crossover strategy. We have implemented a strategy generator which the strategies will be factorized into the elements in Table 1. For every strategy generated, a general adaptation cycle will be completed, and for every 50 adaptation cycles, a model adaptation will be completed. The experiment lasts for 5 model adaptation cycles, and therefore, 250 general adaptation cycles.

A. *Results of Model Adaptation Cycle*

For every 50 general adaptation cycles, or 50 executions of strategies, a model adaptation cycle will be completed. The experiment begins with the cycle_0_model, and Figure 5 shows the MAEs of the current machine learning model and the newly trained machine learning model.

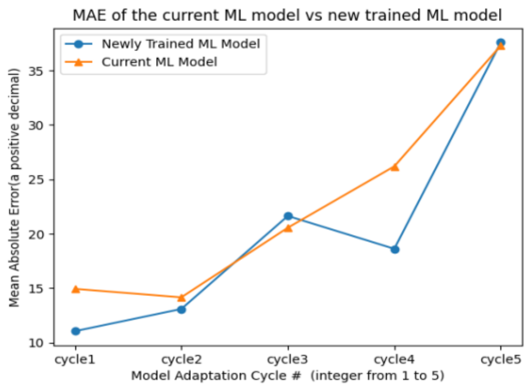

Figure 5: MAE of current ML model and newly trained ML model over 5 model adaptation cycles.

As mentioned in the methodology section, we have introduced the mathematical term "improvement", and through calculating the value of "improvement", the change model of the runtime models will decide if the newly trained machine learning model should replace the current machine learning model. Figure 6 shows the results of "improvement" over 5 model adaptation cycles.

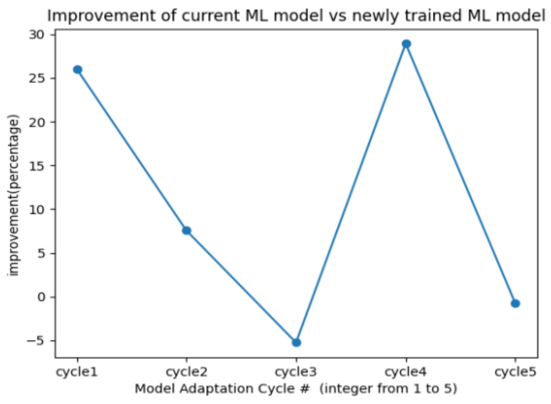

Figure 6: Improvement over 5 model adaptation cycles

From the 5 model adaptation cycles, we can see that the value of "improvement" is greater than 5% occurs in cycle 1, cycle2, and cycle4. Therefore, after these three cycles, the machine learning model has updated its weights. In other cycles, the machine learning model has kept unchanged. Table 4 offers a more detailed look for which machine learning model is used for each model adaptation cycle.

Cycle #	Current ML Model	Updated (Yes/No)	New ML Model
1	Cycle_0_model	Yes	Cycle_1_model
2	Cycle_1_model	Yes	Cycle_2_model
3	Cycle_2_model	No	Cycle_2_model
4	Cycle_2_model	Yes	Cycle_4_model
5	Cycle_4_model	No	Cycle_4_model

Table 4: The machine learning used for each model adaptation cycle.

In Table 4, the column "current ML model" represents the machine learning model used during the model adaptation cycle, and the column "new ML model" represents the machine learning model that will be used for the next model adaptation cycle.

B. *Results of General Adaptation Cycle*

There is total 250 general adaptation cycles that have been completed during the experiment session. In other words, there are 250 randomly generated strategies that are executed. In the fifth model adaptation cycle, we have run the same 50 randomly generated strategies without using any adaptation plan. In other words, we have run these 50 randomly generated strategies simply using the CPython interpreter, and we have also documented the execution time and memory usage. We have selected the results of 7 randomly generated strategies. Figure 7 shows that the comparison of the execution time of these 7 executions of strategies with or without using the adaptation plan.

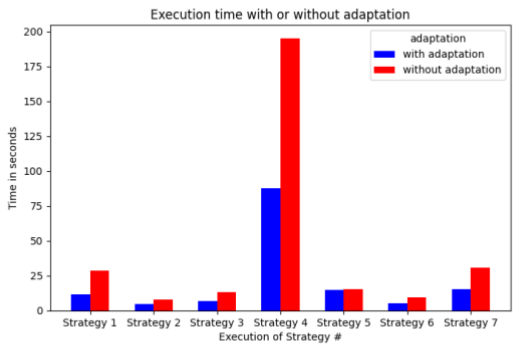

Figure 7: the execution time of 7 random strategies with or without adaptation

Furthermore, we have documented which adaptation plan has been used for these 7 randomly generated strategies. Table 5 shows the adaptation plan that has been used for each strategy.

Strategy #	Adaptation plan #
Strategy 1	Adaptation Plan 2
Strategy 2	Adaptation Plan 2
Strategy 3	Adaptation Plan 2
Strategy 4	Adaptation Plan 2
Strategy 5	Adaptation Plan 1
Strategy 6	Adaptation Plan 2
Strategy 7	Adaptation Plan 2

Table 5: The randomly generated strategies with corresponding adaptation plan

From Table 5, we can tell that only strategy 5 has used the adaptation plan 1, and rest of the other strategies have used adaptation plan 2 for execution. From Figure 7, we can tell that the execution time for all executions using the adaptation 2, or the Pypy JIT compiler, has shown shorter execution time comparing to the executions without any adaptation applied. These comparison results are consistent with our previous experiment in the introduction session that using the Pypy JIT complier can save more time for execution.

For the memory usage, figure 8 shows the comparison of the memory usage of these 7 executions of strategies wih or without using the adaptation plan.

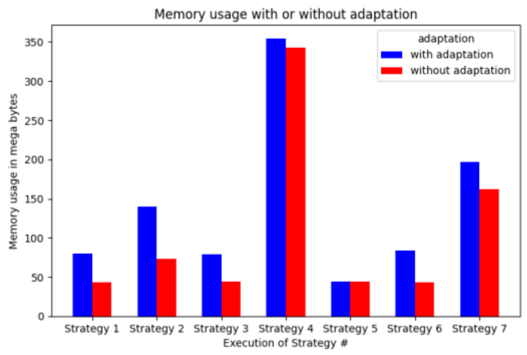

Figure 8: the memory usage of 7 random strategies with or without adaptation

From figure 8, we can tell that except for strategy 5, which has applied adaptation plan 1, the rest of the strategies using the adaptation have shown significantly greater memory usage. These comparison results are consistent with our experiment in the introduction section that the PyPy JIT complier will use more memory for execution.

Eventually, figure 9 shows the comparisons of the values of the output of the utility function from actual execution time and memory usage using the adaptation and not using the adaptation.

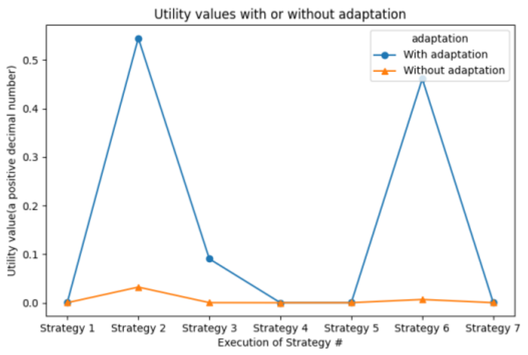

Figure 9: the utility value of 7 random strategies with or without adaptation

From figure 9, we can tell that for strategy 2, strategy 3, and strategy 6, the utility values of using the adaptation is obviously greater than the utility values that are not using the adaptation. The utility values for the rest of the strategies with ot without the adaptation are relatively closer to each other. Table 6 shows a more detailed look for the utility values for each strategy with or without the adaptation.

Strategy #	Utility Value(with adaptation)	Utility Value(without adaptation)
Strategy 1	0.000807	2.74e-11
Strategy 2	0.544	0.0321
Strategy 3	0.0903	0.00014
Strategy 4	6.45e-37	1.776e-83
Strategy 5	2.7414e-05	1.8100e-05
Strategy 6	0.4610	0.0066
Strategy 7	1.4812e-05	2.9917e-12

Table 6: Utility values for each strategy with or without the adaptation

As we can tell from the Table 6, even though the utlity values for strategy 5 with or without the adaptation are very close, all the utility values using the adaptation are greater than the utlity values that does not use the adaptation.

V. Conclusion & Future work

A. *Conclusion*

From results documented in Table 6, we can derive the conclusions from these two perspectives:

- Consistency: as mentioned in the methodology section, the goal management layer uses the utility function to get the utility values from the predicted execution time and the memory usage. The goal management layer will select adaptation goal 2 if the utility value derived from the predicted execution time and memory usage using the PyPy JIT compiler is greater than using the CPython interpreter. Once adaptation goal 2 is selected, the corresponding adaptation plan 2 will be generated. From Table 6, for strategies that use adaptation plan 2, the utility values of using the adaptation are all greater than the utility values not using the adaptation respectively. Therefore, the relationships of utility values derived from the predicted values are consistent with the relationships of the utility values derived from the actual execution time and memory usage.
- Effectiveness: from Table 6, all the utility values using the adaptation are greater than the utility values not using the adaptation. Therefore, the self-adaptation system has offered a solution that has effectively reduced the execution time for trading strategies, meanwhile considering the trade-off of between execution time and memory usage.

The solution offered by the prototype self-adaptive system has shown consistency and effectiveness. Moreover, there is still plenty of room for improvements for this current prototype.

B. *Strategy Factorization*

The self-adaptive system will only be able to optimize the execution time of crossover strategy. The data factorization process will only be applicable to only this type of trading strategy. However, with the advancement of automatic trading, there are plenty of other types of trading strategies. Other methods of factorization process are needed to gather the information of other types of trading strategies. Moreover, the method should not only effectively convert the trading strategies into some useful factors, but also the execution time it takes to complete the factorization process.

C. *Machine Learning Model & Runtime Model Strategy*

The runtime strategy using in this prototype also has many limitations. Firstly, all the machine learning models which are used in the adaptation process have the same topology. However, different types of strategies maybe applicable to different topologies for predictions. Secondly, the strategy of updating the weights will only consider the dataset generated from the previous model adaptation cycle. There are other ways of gathering the data or different loss function to be applied that can make the prediction more accurate. Lastly, the PyPy JIT compiler's parameters are all fixed in the adaptation plan. Different values can be specified to parameters such as threshold, and the model can be trained to predict the most optimal set of parameters for the PyPy JIT compiler.

VI. References

[1] J. Chen, "Backtesting: Definition, How It Works, and Downsides," Investopedia, 18-Aug-2021. [Online]. Available: https://www.investopedia.com/terms/b/backtesting.asp.

[2] J. Chen, "What Is a Crossover in Technical Analysis, Examples," Investopedia, 21-Apr-2022. [Online]. Available: https://www.investopedia.com/terms/c/crossover.asp

[3] Li, Yangguang & Jiang, Zhen. (2019). Assessing and optimizing the performance impact of the just-in-time configuration parameters - a case study on PyPy. Empirical Software Engineering. 24. 10.1007/s10664-019-09691-z.

[4] *What is pypy?¶* (no date) *What is PyPy? - PyPy documentation*. Available at: https://doc.pypy.org/en/latest/introduction.html

[5] Plangger, R., & Krall, A. (2016). Vectorization in PyPy's Tracing Just-In-Time Compiler. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems* (pp. 67–76). Association for Computing Machinery.

[6] Danny Weyns, An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective, Wiley-IEEE Press, 2020