pyEff: New tool for code efficiency measurement

Guillermo Calvo Álvarez[1(✉)], Jesús Enrique Sierra [2(✉)]

[1,2] Department of Civil Engineering, University of Burgos, 09006 Burgos, Spain
[1] gca0011@alu.ubu.es,[2] jesierra@ubu.es

Abstract. Nowadays most of modelling and simulation techniques are based on a computational approach. These models must be implemented in some programing language to be interpreted by the machine in charge of the execution of the model. The measurement of the code efficiency is a key point to optimize the resources of the execution. In this work we present an approach to evaluate the efficiency of code. The code is transformed to an operation matrix what accounts the number of operations of each type. Then this matrix is weighted by a translation matrix which collects the set of instructions of a processor. The approach is tested by developing a new tool capable of measuring efficiency of Python code. The tool can make comparisons between different files in order to identify which code has the highest efficiency. The tool is tested with some examples to check the validity of the technique.

Keywords: Modelling, code efficiency, computational time.

1 Introduction

Nowadays most of modelling and simulation techniques are based on a computational approach. These models must be implemented in some programing language to be interpreted by the machine in charge of the execution of the model. The measurement of the code efficiency is a key point to optimize the resources of the execution. A typical application of the models is the design of controllers of systems and processes. For example in [1-4] different applications of artificial neural networks in modelling and control are proposed. The measurement and the tuning of the code efficiency plays a major role to run models on limited resources hardware platforms such as the low-cost ones and to deploy Greensoft policies [5].

Knowing the efficiency of a computer program is something of vital importance, however the most common way of calculating it today is not a very reliable method. Efficiency is measured by using the time it takes to achieve an objective, but this time is usually conditioned by factors that are beyond the control of those who perform the measurement. Factors such as the load of the CPU at the time the analysis is performed can produce very different efficiency measurements for the same code.

This approach raises a question, is there any way to find the efficiency of a program without being affected by factors external to the code?, by calculating the number of operations that the program uses in the execution, this would get an invariable measure and it could be used to calculate a theoretical efficiency.

Therefore, the aim of the tool presented in this paper is to perform an efficiency analysis calculating the aforementioned theoretical efficiency. Furthermore, it allows to simulate different environments in which the codes could be executed in order to appreciate the change they would have in the efficiency, depending for example on the type of processor with which they executed the code of a program. Also, it is able to compare programs that have the same purpose and thus discover which is more efficient, and for which, it would be better to use.

Some previous works have studied the measurement of the computation time.For example, Rosendahl describes a system to derive upper bounds for the computation time using abstract interpretation, and tests this approach with Lisp [6]. Other family of papers are focused on a related problem, the development of tools and methods to measure the energy efficiency of the software. In this line, a survey of techniques for improving energy efficiency in embedded computing systems is presented in [7]. A survey focused on GPUs is exposed in [8]. Some metrics on Energy-Efficiency of Software are developed in [9].

The paper is organized as follows. The process in which the theoretical computational time is obtained from a given code is explained in section 2. The section 3 describes the tool, its composition and main functions. Results are presented and discussed in section 4. The document ends with the conclusions and future works.

2 From code to computational time

The transformation of the code into a measure of efficiency goes through several stages:

1. Change the high level code to Bytecode through the compiler.
2. Interpret translates and executes the ByteCode.
3. Meanwhile the interpret execute the byteCode, the Operations Matrix is obtained.
4. Weighting through the Operations matrix within the theoretical processors to obtain the computational time.

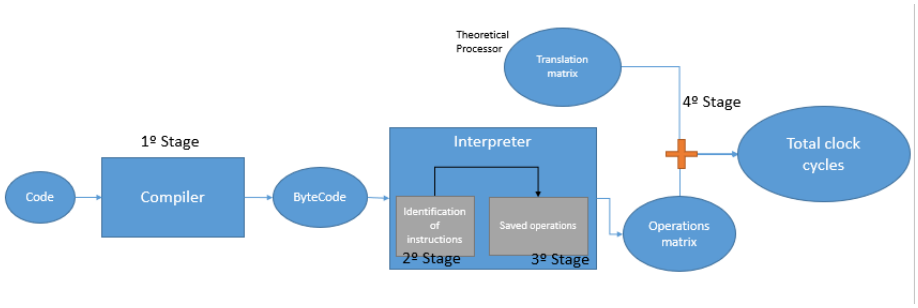

Fig. 1. Flow diagram to transform the high-level code to computational time

In the first stage as it can be seen in the figure 1, the compiler is in charge of transforming the high level code (figure 2) into another type of intermediate level code that is understandable by the interpreter, this intermediate level code is called Bytecode (figure 3).

```
x=10
y=8
z=x*y
```

Fig. 2. Example of a high-level code

```
1           0 LOAD_CONST               0 (10)
            3 STORE_NAME               0 (x)

2           6 LOAD_CONST               1 (8)
            9 STORE_NAME               1 (y)

3          12 LOAD_NAME                0 (x)
           15 LOAD_NAME                1 (y)
           18 BINARY_MULTIPLY
           19 STORE_NAME               2 (z)
           22 LOAD_CONST               2 (None)
           25 RETURN_VALUE
```

Fig. 3. Example of the ByteCode that comes out after the compiler's translation process

Then in stage two, once the ByteCode is obtained, the interpreter is able to translate the instructions that come in it, and execute them while it analyzes the ByteCode. While the interpreter does this process, at the same time he is managing the operations that are found in the code.

The matrix of operations M_O is treated, as its name indicates, of a matrix in which the number of operations that the interpreter is recognizing in execution time is stored. This matrix is formed by an index composed by the type of operation found and the type of the first of the operators, in this way each tuple of the matrix is collecting the number of times that a certain type of operation is found with that type of operator, and to know the specific cell where the count should be registered, each column of the matrix represents the type of value that the second operator can have. In order to better understand this concept, an example of the basic structure that the operations matrix can take is shown in Table 1.

Table 1. Example of basic structure of the operations matrix

	INT	STR	BOOL	COMPLEX
[ADD, int]	12	5	Null	Null
[ADD, str]	Null	Null	Null	Null
[ADD, bool]	Null	Null	7	Null
[MULTIPLY,int]	Null	9	Null	Null
[MULTIPLY,str]	Null	Null	Null	Null

In the table 1 is possible to see how there are four types of operations, 12 sums between integers, 5 sums between an integer and a string, 7 sums between a Boolean and a multiplication between an integer and a string.

Once this matrix is obtained, the total number of total operations executed in the execution time can be drawn, but to be able to talk about efficiency, first we must weight these figures to one uniform unit that allow us to compare them, in the case of this work it has been decided to choose cycles of clock. But whatever other unit which allow us to weight the cost of each operation could have been selected. To extract the quantity with which we must weigh each operation, there is a translation matrix M_T .

The translation matrix has exactly the same structure as the operations matrix, but in this case instead of saving the number of operations that the interpreter is detecting, it saves a unit that serves to differentiate the different types of operations according to their cost, the more cost an operation has, the worse it is when talking about efficiency. In this case the chosen unit has been clock cycles that an operation of a specific type would take to execute. This matrix is generated external to the tool and is filled with the values that each one desires, according to the type of environment that you want to simulate. The external files in which the matrices are located are the so-called theoretical processors. This name is due to the fact that it is theoretically intended to calculate the clock cycles of the operations according to the features that a given processor could provide, and in this way the processors can be simulated by adjusting the parameters of the matrices as desired.

And finally in the fourth stage, after obtaining the necessary values to calculate the clock cycles that the code takes to execute, the following formula is applied:

$$\boldsymbol{Tcloc} = \sum_{(i,j)}^{(N,M)} M_O\,(i,j) M_T(i,j) \tag{1}$$

Where N and M are the number of rows and columns of M_O. Must be paid attention at the size of M_T must be bigger than the size of M_O.

Although this tool has been developed entirely in Python, and currently can only analyze .py files exclusively, one of the main ideas is that later it can be used in different environments, for different types of languages and can perform a wider variety of analysis, in order to better adapt to the different demands that an in-depth analysis of the efficiency of a code may require, in a more demanding environment. It was decided to start with this type of language, because today it is widely used in different fields such as robotics, thanks to the great variety of libraries it has. And it would be very interesting to see the results that could be obtained in all these fields.

3 Description of the tool

The internal structure of the program is depicted in the figure 4. As it may be observed in the figure 4, it is a view controller model. In this case the controller and the view are implemented in such a way that they share some components, hence they share a box in the image, but even so the concept of the model is typical in which the controller acts as an intermediary. In the tool, the controller acts as an intermediary between the

graphical interface, which allows interacting and shows the results of the efficiency analysis, and the interpreter, called ByteRun, which analyzes and executes the code.

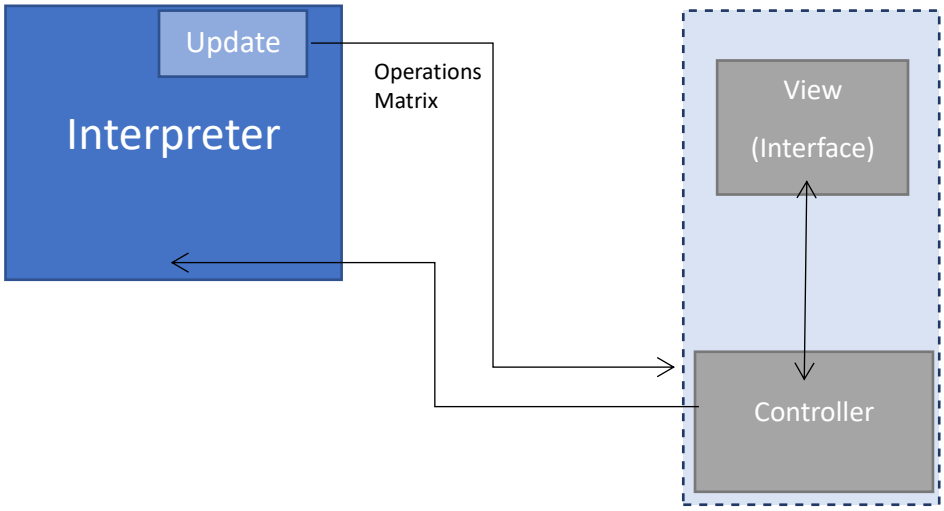

Fig. 4. View of the internal structure of the tool

Byterun [10] is a Python interpreter made in Python which has its advantages and disadvantages, the main drawback is its speed, it is much slower than other types of interpreters such as CPython (A python interpreter made in C), but on the other side as advantage is an interpreter easy to understand its operation and root of this is easier to implement changes in this when needed, which is the case of this tool where you had to add a way in which the interpreter will identify and save the operations in some way. That is why, for the development of this tool, byteRun was chosen as the interpreter.

In the figure 4, it is shown how the interpreter has a prominent part within it, this refers to the part that has been implemented within the ByteRun for this Project in particular, and it is a series of functions that compare and save in the operations matrix the instructions that are detected as operations. Once the values of the matrix are calculated, the controller passes the values selected from the view of the translation matrix, in order to proceed to calculate the efficiency. Once it has finished and it has the total clock cycles, it returns them to the controller to distribute them to the interface according to the user's request.

Turning to the topic of functionalities that have the tool, we must highlight two main functional (figure 5):

- Individual analysis
- Multiple / Comparative analysis

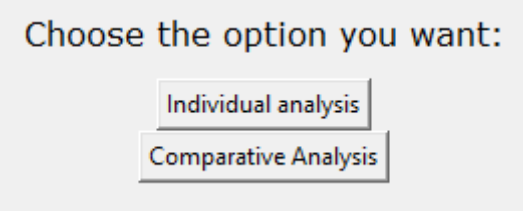

Fig. 5. View of the main window

The individual analysis, as its name indicates, is intended to measure the efficiency of a single file. You can choose the file you want, as long as it is type.py. After selecting the file, the tool invokes the interpreter and translates and saves the operations that arise when executing the ByteCode that has been previously transformed by the compiler.
Once finished the analysis, it shows a button, that after pressing it opens a window with a graph with the percentages of operations that it has found in execution time. Up to this point the weights have not yet been applied to the operations, to do this, in the upper part of the window there are checkboxes Where the theoretical processors can be selected (CSV type files that contains the translation matrix $M_T$where weights are given to each type of operation). Once you have selected the processor you want, you can also change the types of operation that you do not want to show on the chart. Once the parameters have been configured according to the priorities required by the user, pressing the “Show results” button causes the graph to change, showing the percentages of the desired operations with the weight corresponding to the values proposed by the theoretical processor. At that moment it shows you the percentage in clock cycles that each operation uses (figure 6). This functionality is designed to debug a code, since thanks to the percentages of the result it is possible to identify which operations consume the most resources

The Multiple / Comparative analysis shares a series of characteristics with the individual analysis, such as how to choose the theoretical processor to weigh the operations or how to choose the files, but the purpose of this function is different. To make the Multiple / Comparative analysis it is mandatory to choose more than one file, since this functionality is based on the comparison of several files.

Once selected, as in the individual analysis, the interpreter makes the analysis of the Bytecode through of the interpreter, in this case that of each of the selected files, and save the operations obtained from each, once analyzed you can access a window where at the beginning only a couple of buttons shows a "listbox" and a series of checkboxes. The listbox, as already explained, lets you choose a theoretical processor, the checkboxes show all the operations among all the analyzed files, all these are marked, but they can be removed to show only the desired operations. When selecting the selected parameters, click on the “Show results” button to show a comparative graph between the total clock cycles of the selected operations of each file

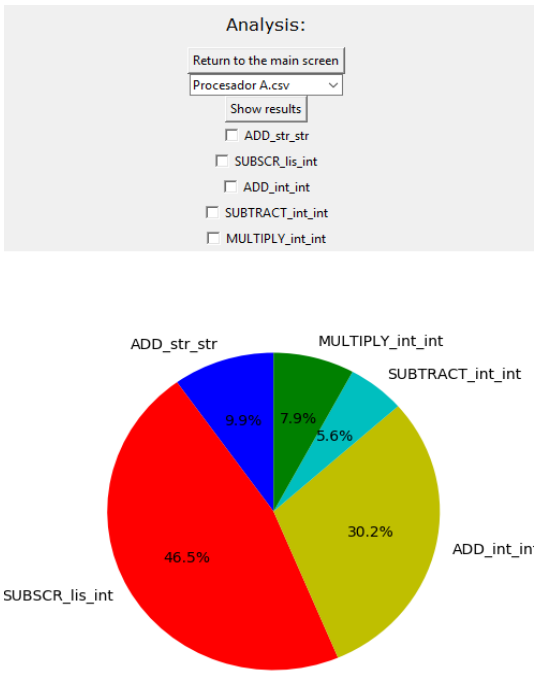

Fig. 6. Window where you can parameterize and observe the results of the individual analysis

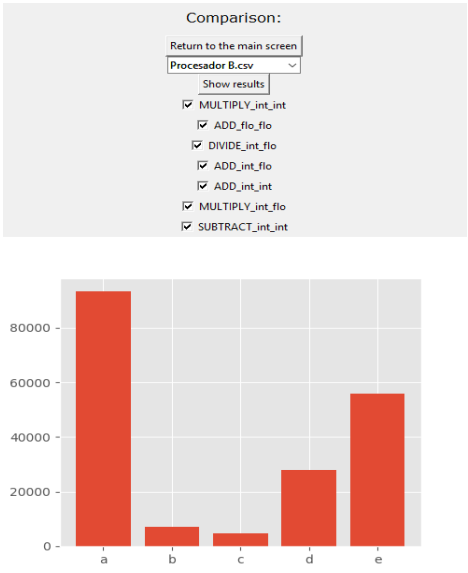

Fig. 7. Window where you can parameterize and observe the results of the multifile analysis

This functionality can be extremely useful if we have several codes with the same objective and we want to know which one responds with a better efficiency according to the scope or processor that executes it.

4 Discussion of results

To be able to operate the tool in a practical environment, we will simulate a multiple analysis of a Python code, combining the values of some parameters to discover with which parameters the code responds with better efficiency. The code used in the example is shown in the figure 8.

```
tEnd=100;ts=0.1
KP=2;KD=8;KI=1
t=0;e=0;eSum=0;eOld=0
while t < tEnd:
    t=t+ts
    e=signal-ref
    out=KP*e + KD*(e-eOld)/ts + KI*eSum*ts
    eOld=e
    eSum+=e
```

Fig. 8. View of the high-level code that will be processed in this example

Once the files we want to analyze have been selected, the compiler is called to translate the codes to ByteCode (figure 9).

When the ByteCode is already achieved, it is when the interpreter begins to perform its functions and generates the contents of the operations matrix (table 2).

Table 2. View of how the operations matrix of one of the files looks after saving all the operations

	INT	STR	BOOL	FLOAT
[ADD, int]	2001	0	0	1002
[ADD, flo]	0	0	0	1001
[MULTIPLY,int]	3003	0	0	1001
[SUBTRACT,int]	2002	0	0	0
[DIVIDE,int]	0	0	0	1001

In the previous table one of the operation matrices has been displayed in a summarized way, only the tuples that contain some value are shown. It should be noted that there would be n Matrices, one for each file analyzed.

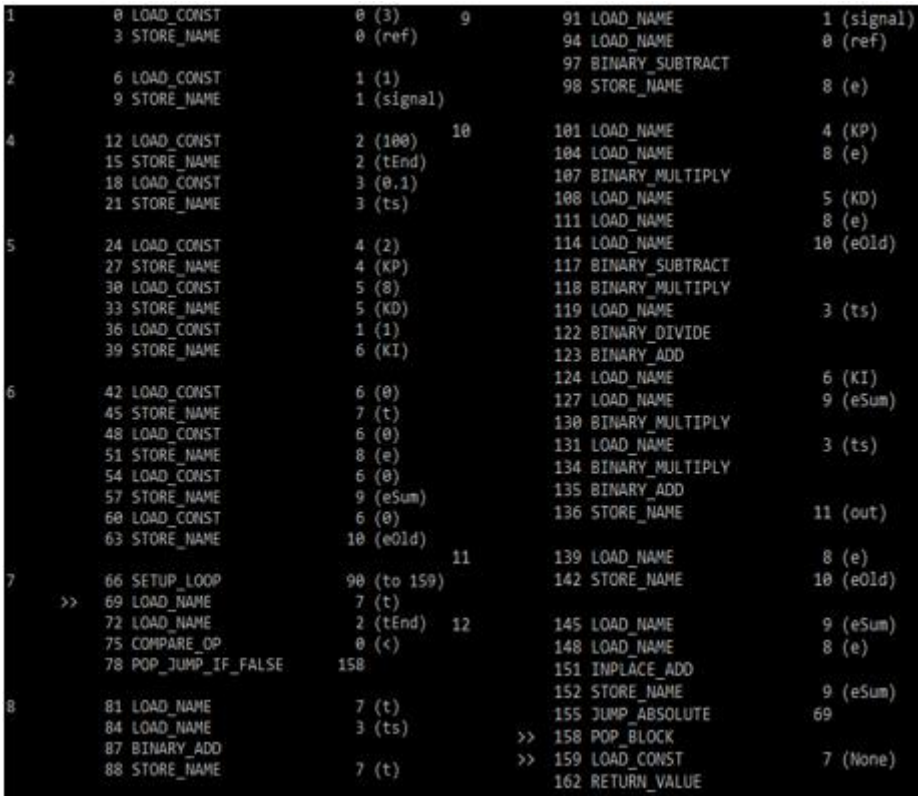

Fig. 9. View of the byteCode resulting from one of the files

After calculating the operations matrix, we have to choose the desired processor to weight the operations contained in the operations matrix. Here we choose according to the field in which we want to see the code work. And for that we have to know what values there are within the translation matrix (table 3).

Table 3. Example of basic structure of the translation matrix

	INT	STR	BOOL	FLOAT
[ADD, int]	1	4	2	3
[ADD, flo]	2	2	2	2
[MULTIPLY,int]	1	2	3	2
[SUBTRACT,int]	1	4	2	2
[DIVIDE,int]	2	5	3	3

As in the view of the operations matrix, only the tuples that have a relevant value for the example we are executing are shown in this view. In a total view of the translation matrix there are many more rows.

To check the validity of the technique, the tool is executed in multifile analysis, each file has different tEnd and ts values for the code of the figure 8. The results are shown in the table 4.

Table 4. Execution of the tool for different parameters

File	tEnd	ts	Clock cycles
A	500	0.015	466676
B	100	0.015	93338
C	400	0.1	56000
D	100	0.1	14014
E	**100**	**0.5**	**2800**
F	100	0.3	4676
G	200	0.5	6000
H	100	0.2	7000
I	100	0.05	28014
J	100	0.025	56000
K	500	0.025	280014

With all this data the tool makes its analysis and shows us the results of the figure:

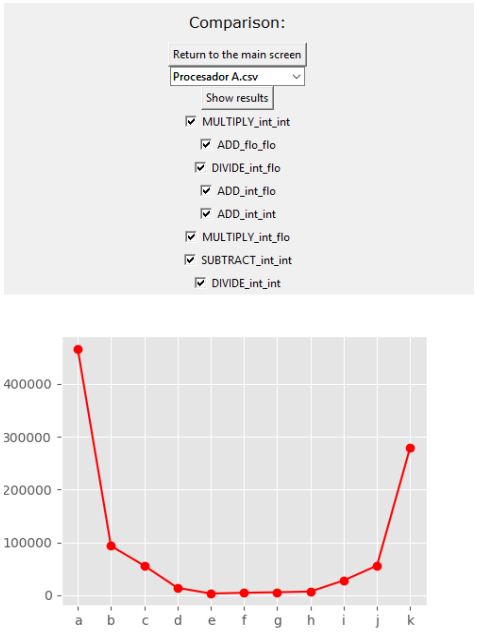

Fig. 10. View of the results of the efficiency analysis

As it may be seen in the figure 10 and the table 4, the best results are obtained by the file e, with parameters tEnd=100 and ts=0.5.

5 Conclusions and future works

As it has been seen throughout the article, this tool can be very useful in different fields and it be adapted to different user requirements. We can extract the necessary information to make an efficiency analysis with enough precision, and without affecting the results due to undesired factors, such as the load of the processor.

At the moment this tool is only available for codes in Python language, in the future the number of supported languages will be extended. Other planned work is to adapt the tool to perform parameters analysis. This feature will vary automatically a set of parameters selected by the user, and it will identify the values which offers better efficiency for the code.

References

1. Sierra, J. E., & Santos, M. (2018). Modelling engineering systems using analytical and neural techniques: Hybridization. Neurocomputing, 271, 70-83.
2. San Juan, V., Santos, M., & Andújar, J. M. (2018). Intelligent UAV Map Generation and Discrete Path Planning for Search and Rescue Operations. Complexity, 2018.
3. Santos, M., Lopez, V., & Morata, F. (2010, November). Intelligent fuzzy controller of a quadrotor. In Intelligent Systems and Knowledge Engineering (ISKE), 2010 International Conference on (pp. 141-146). IEEE.
4. Sierra, J. E., & Santos, M. (2019). Wind and Payload Disturbance Rejection Control Based on Adaptive Neural Estimators: Application on Quadrotors. Complexity, 2019.
5. Naumann, S., Dick, M., Kern, E., & Johann, T. (2011). The greensoft model: A reference model for green and sustainable software and its engineering. Sustainable Computing: Informatics and Systems, 1(4), 294-304.
6. Rosendahl, M. (1989, September). Automatic complexity analysis. In Fpca (Vol. 89, pp. 144-156).
7. Mittal, S. (2014). A survey of techniques for improving energy efficiency in embedded computing systems. arXiv preprint arXiv:1401.0765.
8. Mittal, S., & Vetter, J. S. (2015). A survey of methods for analyzing and improving GPU energy efficiency. ACM Computing Surveys (CSUR), 47(2), 19.
9. Johann, T., Dick, M., Naumann, S., & Kern, E. (2012, June). How to measure energy-efficiency of software: Metrics and measurement results. In Proceedings of the First International Workshop on Green and Sustainable Software (pp. 51-54). IEEE Press.
10. byteRUN 2019. https://github.com/nedbat/byterun