What Are Your Programming Language's Energy-Delay Implications?

Stefanos Georgiou
Athens University of Economics and Business
sgeorgiou@aueb.gr

Panos Louridas
Athens University of Economics and Business
louridas@aueb.gr

Maria Kechagia
Delft University of Technology
m.kechagia@tudelft.nl

Diomidis Spinellis
Athens University of Economics and Business
dds@aueb.gr

ABSTRACT

Motivation: Even though many studies examine the energy efficiency of hardware and embedded systems, those that investigate the energy consumption of software applications are still limited, and mostly focused on mobile applications. As modern applications become even more complex and heterogeneous a need arises for methods that can accurately assess their energy consumption.
Goal: Measure the energy consumption and run-time performance of commonly used programming tasks implemented in different programming languages and executed on a variety of platforms to help developers to choose appropriate implementation platforms.
Method: Obtain measurements to calculate the Energy Delay Product, a weighted function that takes into account a task's energy consumption and run-time performance. We perform our tests by calculating the Energy Delay Product of 25 programming tasks, found in the Rosetta Code Repository, which are implemented in 14 programming languages and run on three different computer platforms, a server, a laptop, and an embedded system.
Results: Compiled programming languages are outperforming the interpreted ones for most, but not for all tasks. C, C#, and JavaScript are on average the best performing compiled, semi-compiled, and interpreted programming languages for the Energy Delay Product, and Rust appears to be well-placed for I/O-intensive operations, such as file handling. We also find that a good behaviour, energy-wise, can be the result of clever optimizations and design choices in seemingly unexpected programming languages.

CCS CONCEPTS

• **Hardware → Power estimation and optimization**; • **Software and its engineering →** *Software libraries and repositories*; *Software design tradeoffs*;

KEYWORDS

Programming Languages; Energy-Delay-Product; Energy-Efficiency;

ACM Reference Format:
Stefanos Georgiou, Maria Kechagia, Panos Louridas, and Diomidis Spinellis. 2018. What Are Your Programming Language's Energy-Delay Implications?. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196398.3196414

1 INTRODUCTION

Nowadays, energy consumption[1] mattes more than ever before—given that modern software applications should be able to run on devices with particular characteristics (e.g., regarding their main memory and processor). Although hardware design and utilization is undoubtedly a key factor affecting energy consumption, there is much evidence that software can also significantly influence the energy usage of computer platforms [7, 16, 18].

Today, software practitioners can select from a large pool of programming languages to develop software applications and systems. Each of these programming languages comes with a number of features and characteristics that can affect the energy consumption and run-time performance of programming tasks implemented in such languages. With the advent of cloud computing, data centers, and mobile platforms, the same programming tasks can run on distinct platforms consuming energy in different ways. In this context, there is limited work available that examines the energy and performance implications of particular programming tasks that are written in different languages and run on different platforms.

In this paper, we measure the Energy Delay Product (EDP), a weighted function of the energy consumption and run-time performance product, for a sample of commonly used programming tasks. We do this to identify which *programming language implementations* (i.e., programming tasks developed in particular programming languages) are more efficient. The usage of a metric like EDP can help us to make suggestions regarding the programming languages that should be used for the development of particular programming tasks, which are dependent on the energy or performance requirements of the software systems and applications the tasks

[1] Although in the physical sense energy cannot be consumed, we will use the terms energy "demands", "consumption", and "usage" to refer to the conversion of electrical energy by computers to thermal energy dissipated to the environment. Correspondingly, we will use the term energy "savings", "efficiency", and "optimization" to refer to reduced consumption.

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196414

belong to. At the same time, the EDP metric can help us to investigate how the type of the execution environment, where particular implementations run on, affects their efficiency.

To achieve our goals, we make use of a large publicly available repository that is used for programming chrestomathy,[2] the Rosetta Code Repository [29]. Conducting an empirical study on this data set, we elicit EDP results from small programming tasks implemented across popular programming languages. In our experiments, we include different programming paradigms, such as procedural, object-oriented, and functional. Our results highlight the EDP implications of a particular programming language's implementations. We also analyze the collected results to derive conclusions on the behaviour of different computing platforms (i.e., embedded, laptop, and server system) viz-a-viz EDP.

The paper makes the following contributions:

- a customized and extended data set that can be used as a benchmark for similar studies to ours,
- a set of publicly available tools for measuring the EDP of various programming tasks implemented in different programming languages,
- an empirical study on programming language EDP implications, by using different types of programming tasks and software platforms, and
- a programming language-based ranking catalogue, in the form of heat maps, where developers can find which programming language to pick for particular tasks and platforms; when energy or run-time performance are important.

Our results show that C is the most EDP efficient language when it comes to computational-intensive tasks for almost all of our platforms. Go outperforms most implementations for sorting algorithms, Rust seems to make the most EDP efficient file-I/O operations, JavaScript performs better for regular expression tasks, and C++ is the most efficient for function composition. On average, C, C#, and JavaScript are the most EDP efficient programming languages, while Swift, Java, and R show the weakest performance among the compiled, semi-compiled (i.e., compiled into intermediate code and executed on virtual machines), and interpreted groups, respectively. Our analysis also shows that the average EDP of the same programming language is significantly different, in statistical terms, between the embedded and the laptop platforms. However, we could not detect a statistically significant difference between the embedded and the server or the laptop and server platforms.

2 METHODS

In this section, we provide details regarding our objectives, research questions, and samples. We also present our automated approach for calculating programs' EDP, its input parameters, as well as its threats and limitations.

2.1 Research Questions

Previous studies have claimed that compiled programming languages such as C and C++ are the most energy and performance efficient, while semi-compiled and interpreted languages are the least efficient [1, 10, 33]. We attempt to investigate whether the above statement is true by conducting a large scale empirical study on a sample of 25 programming tasks implemented in 14 programming languages. In addition to that, our focus is to introduce an automatic approach that software practitioners can use to calculate the EDP of a selected programming language for a certain task type, e.g., I/O- or CPU-intensive applications, running on specific computer systems. We define our research questions as follows:

RQ1. *Which programming languages are the most* EDP *efficient and inefficient for particular tasks?*—Our objective here is to rank the selected programming languages based on the EDP of the implemented programming tasks. By answering this research question, we will guide practitioners on which programming languages they should avoid or consider when developing software applications that require to be EDP efficient for particular tasks.

RQ2. *Which types of programming languages are, on average, more* EDP *efficient and inefficient for each of the selected platforms?*—Our goal here is to evaluate the efficiency of different programming languages' families (i.e., compiled, interpreted, and semi-compiled) based on the average EDP of their implementations, when the latter are running on particular software platforms (i.e., server, laptop, and embedded system). By answering this research question, we will determine which types of programming languages can on average produce better EDP results when running on specific platforms.

RQ3. *How much does the* EDP *of each programming language differ among the selected platforms?*—For answering this research question, we plan to examine whether the average EDP of the selected tasks, which are implemented in a particular programming language, differs when these tasks run on different software platforms (i.e., server, laptop, and embedded system). By answering this research question, we will define whether programming tasks implemented in a specific programming language have similar EDP behavior when running on different software platforms.

2.2 Selection of Subject Systems

Data Set. We made use of the Rosetta Code Repository, a publicly available repository for programming chrestomathy. Rosetta Code offers 868 tasks, 204 draft tasks, and has implementations in 675 programming languages [29]. However, not all tasks are developed in all programming languages that we have selected for our experiments. To retrieve the online data set, we used a public Github Repository [3] that contains all the currently developed tasks listed in Rosetta Code's web-page.

Since there is a large number of programming languages' implementations available in Rosetta Code, we consulted the Tiobe [38] website to elicit the most popular languages. Tiobe offers a monthly rating index of programming languages' popularity, by estimating the number of hits for a given search query on the internet. The search query is applied on 25 of the highest ranked search engines (according to Alexa) and it searches for programming language hits across the web that: 1) refer to at least 5,000 hits for the Google search engine, 2) are Turing complete, and 3) have their own Wikipedia page [39]. Taking Tiobe October's 2017 index rating, we

[2] An ancient Greek word meaning "desire to learn".

Table 1: Programming Languages, Compiler and Interpreter Versions, and Run-Time Performance Optimization Flags

Categories	Programming Languages	Compilers & Interpreters Embedded	Laptop	Server	Optimization Flags
Compiled	C	6.3.0	6.4.1	6.4.1	-O3
	C++	6.3.0	6.4.1	6.4.1	-O3
	Go	1.4.3	1.7.6	1.7.6	–
	Rust	1.20.0	1.18.0	1.21.0	-O
	Swift	3.1.1	3.0.2	3.0.2	-O
Semi-Compiled	C#	4.6.2	4.6.2	4.6.2	-optimize+
	VB.NET	4.6.2	4.6.2	4.6.2	-optimize+
	Java	1.8.0	1.8.0	1.8.0	–
Interpeted	JavaScript	9.0.4	8.9.3	8.9.3	–
	Perl	5.24.1	5.24.1	5.24.1	–
	PHP	5.6.30	7.0.25	7.0.25	–
	Python	2.7.23	2.7.13	2.7.13	-O
	R	3.3.3	3.4.2	3.4.2	–
	Ruby	2.4.2	2.4.1	2.4.1	–

selected the top 14 programming languages. We excluded from our initial list programming languages such as assembly, Scratch, Matlab, and Objective Pascal, that have particular limitations, i.e., they are architectural, visually oriented, proprietary, and OS dependent, respectively. On the contrary, we included programming languages such as R, Swift, Go, and Rust, although they were not among the top 14. We have selected R and Swift because they had the next highest popularity in Tiobe. Also, we chose Go and Rust because they had one of the highest rise in ratings in a year and are considered to be promising according to Tiobe [38]. The selected programming language categories (compiled, interpreted, and semi-compiled), along with their names, compilers and interpreters' versions, and compile-time performance optimization flags are shown in Table 1. We selected the highest possible performance optimization flags, since some languages (e.g., Go) by default apply the highest degree of optimizations. Regarding run-time configurations, we did not use or tune any environmental variables.

After retrieving our data set, we had to filter, crop, and modify it to adapt it to our study's needs. For instance, consider that most of the categories found in Rosetta Code, such as arithmetic and string manipulation, offer more than one task. However, we had to use a balanced data set of different types of tasks. Therefore, we developed a script that extracts all the tasks that are implemented in at least half of the selected 14 programming languages. We ended up with the 25 tasks shown in Table 2. Table's 2 first and second columns list the selected categories and the names of the tasks, as they were found in Rosetta Code. The third column explains the tasks and the fourth column provides the inputs that were used for each task; we tried to keep the original inputs as given from the Rosetta Code task Wiki-pages. The fifth column shows the task name abbreviations as they are used in Figure 1, 2a, 2b, and 2c.

Handling the Data Set. To properly use the above-mentioned data set, we had to do several amendments. Initially, we had to add each task to a `for` loop for making the run-time execution of each task to last more than a second—since our power analyzer, i.e., Watts Up Pro [41], has a sampling rate of a second (rate lowest). The `for` loop's iterations among the tasks vary between a thousand to two billion of times. It is important to note that some implementations last significantly longer. For instance, the *exponentiation-operator* for C and R took 4.5 seconds and 109 minutes, respectively, to execute the task two billion of times. In addition, consider that some compilers and interpreters offer aggressive source code optimizations. Then, once they find out that the same function is being repeated multiple of times, they optimize their native code to avoid unnecessary calculations. To handle the above issue and execute the same tasks multiple times, we made the programming tasks' loop variable dependent to enforce different outcomes each time. We also used `volatile` variables (whenever a programming language offered such an option), whose value may change between different accesses. Furthermore, some of the tasks developed in a specific language offered more than one implementation. Here, we chose the one that was most similar with the implementations in the other selected programming languages. For example, we added any required scaffolding, such as a `main` function or libraries that needed to be installed and configured. We also developed from scratch the tasks that were not implemented for all the programming languages of our selection whenever it was possible (e.g., multiple inheritance is not applicable in C#).

To execute all the tasks and collect the results, we developed a number of scripts that are available for public use in our Github Repository [19]. In total, we wrote 2,799 lines of source code in BASH, Python, and Java to compile, execute, collect, filter, and plot our results. Moreover, we wrote 1,373 lines of source code to implement the missing tasks for programming languages including C, C#, JavaScript, Perl, PHP, R, Rust, Swift, and VB.NET.

Experimental Platform. To perform our experiments, we used a Dell Vostro 470 (with 16 GB RAM) server [12], an HP EliteBook 840 G3 Notebook laptop [25], and two Raspberry Pi 3b model [34], with an Intel i7-3770, an Intel i7-6500U, and a quad-core ARM Cortex-A53 micro-processors, respectively.

From hereafter, we refer to the server, laptop, and one of the RPis as Computer Node (CN) and to the Watts Up Pro as WUP. The CNs responsibility is to execute the tasks and to retrieve their execution time using `time` [15]. In addition, we note that we used one of the two RPis (the one is not acting as a CN) to retrieve the energy measurements from WUP's internal memory. We will refer to it as an Energy Monitoring (EM) system. Finally, to extract the collected measurements from WUP's internal memory, we used an open source Linux utility-based interface available in GitHub [5].

2.3 Research Method

Our calculations are based on EDP, an equation introduced by Horowitz et. al [24] and applied as a weighted function by Cameron et. al [6]. The EDP is defined as follows:

$$E \times T^w \tag{1}$$

Using the term E, we denote the total energy consumed by a particular task from the start until its finish time. T is the total execution time of a task. The exponent w denotes weights, and it can take the following values: 1 for *energy efficiency* when energy is of major concern; 2 for *balanced*, when both energy consumption and performance are important; 3 for *performance efficiency*, when performance is most important. Here, we kept the exponent w equal to 1 (to compare energy/performance in equal terms) to

Table 2: Selected Categories, Tasks, Explanation, Input Test, and HeatMap Abbreviations

Categories	Names	Explanation	Input Test	Abbreviations
Arithmetic	*exponentiation-operator*	exponentiates integer and float	$2017^{12}, 19.88^{12}$	exp.-operat.
	numerical-integration	calculates the definite integral by using methods rectangular{left,right,midpoint}, trapezium, and Simpson's for 10^x approximations	$f(x) = \int_0^1 x^3, 10^2$ $f(x) = \int_1^{100} 1/x, 10^3$ $f(x) = \int_0^{5000} x, 5 \times 10^5$ $f(x) = \int_0^{6000} x, 6 \times 10^5$	num.-integ.
Compression	*huffman-coding*	encodes and decodes a string	"huffman example"	huffman
	lzw-compression	encodes and decodes a string	"Rosetta Code"	lzw-compr.
Concurrent	*concurrent-computing*	threads creation and printing	[Enjoy,Rosetta,Code]	conc.-comp.
	synchronous-concurrency	shares data between 2 threads	Random text file	synch.-conc.
Data structures	*array-concatenation*	concats two integer arrays	[1,2,3,4,5], [6,7,8,9,0]	array
	json	serializes and loads JSON in data structure	"foo":1,"bar":["10", "apples"]"	json
File handling	*file-input-output*	reads from A and writes to B	10,000 unique binary files	file-I/O
Recursion	*factorial*	factorial of n	(10!)	factorial
	ackermann-function	examples of a total computable function that is not primitive recursive	$A(m,n) = n+1, A(m,n) = A(m-1,1)$, $A(m,n) = A(m-1, A(m-1,1))$	ackermann
	palindrome-detection	finds if word is palindrome	"saippuakivikauppias"	palindrome
Regular Expression	*regular-expression*	matches a word from a sentence and then replaces a word	"this is a matching string"	regex
Sorting algorithms	{*selection, insertion, merge, bubble,quick*}	sorts an array of 100 random elements	[the same 100 random elements for all cases]	selection, insertion, merge ,bubble, quick
String manipulation	*url-encoding*	encode a string	"http://foo bar/"	url-encode
	url-decoding	decode a string	"http%3A%2F%2 ffoo+bar%2fabcd"	url-decode
Object Oriented	*call-an-object-method*	calls a method from an object		obj-method
	classes	creates an object		classes
	inheritance-multiple	invokes inherited classes methods		inher.-multi.
	inheritance-single	invokes inherited class method		inher.-single
Functional	*function-composition*	pipes a function's result into another	$sin(asin(0.5))$	func.-comp.

address **RQ1** and **RQ3**, and we used all three weights (to see how performance affects the average EDP) to answer **RQ2** in Section 3.

We chose EDP among other energy metrics (e.g., Greenup, Speedup, and Powerup [2]) because normalized EDP can offer fair comparisons among programming implementations, which run on different execution platforms. Contrary to Abdulsalam's et al. study [2], we do not evaluate the efficiency of optimizations, where performance and energy is measured separately.

Before collecting our measurements, at the CN's boot time, we had to ensure a *stable condition* (where the energy usage is stable) before starting to retrieve measurements—we defined a waiting period of five to minutes to avoid adding overhead to our results.) As it is suggested by Hindle [23], we tried to shut down background processes found in modern operating systems, such as disk defragmentation, virus scanning, cron jobs, automatic updates, disk and document indexing, and so on, to minimize possible interference to our measurements. Additionally, to ensure our systems' stable condition, we used the Linux-monitoring sensors tool, i.e., the `lm_sensors` [35] for the server and laptop platforms and `vcgencmd` [17] for the RPi. We used `lm_sensors` and `vcgencmd` to retrieve the systems' temperature. If the systems' temperature was found high compared to its idle temperature, it could be possible that the system was using its fans to cool down (thus consuming additional energy) and the processor's clock speed might scale down (reducing run-time performance) to avoid overheating. On such occasions, our script stalled the execution of the next task's implementation until they reach a stable condition.

Upon reaching a stable condition, our script initiated the execution of the tasks. Before the execution of a task the CN sent an SSH command to the EM device to start retrieving energy measurements from the WUP's internal memory. By the end of the whole use case, the EM has collected and sent the energy measurements to the CN through the SCP utility. When the results were received by the CN, a plotting script depicted them in the form of heat maps.

2.4 Threats to Validity

Internal validity. Internal validity refers to possible issues of our techniques that can lead to false positives and imprecision. Here, we reveal the sources of such problems.

First, we used cabled instead of wireless connection, because the former is more energy efficient. However, the use of different protocols provided by the network connection might cause additional overhead. Also, the laptop we used has irremovable battery and in case of discharge, the power supply immediately starts charging. This may also introduce additional overhead.

Second, WUP offers aggregate sampling and reports a sample per second. This means that the energy consumption of the operations that last less than a second is not reported. Therefore, such cases

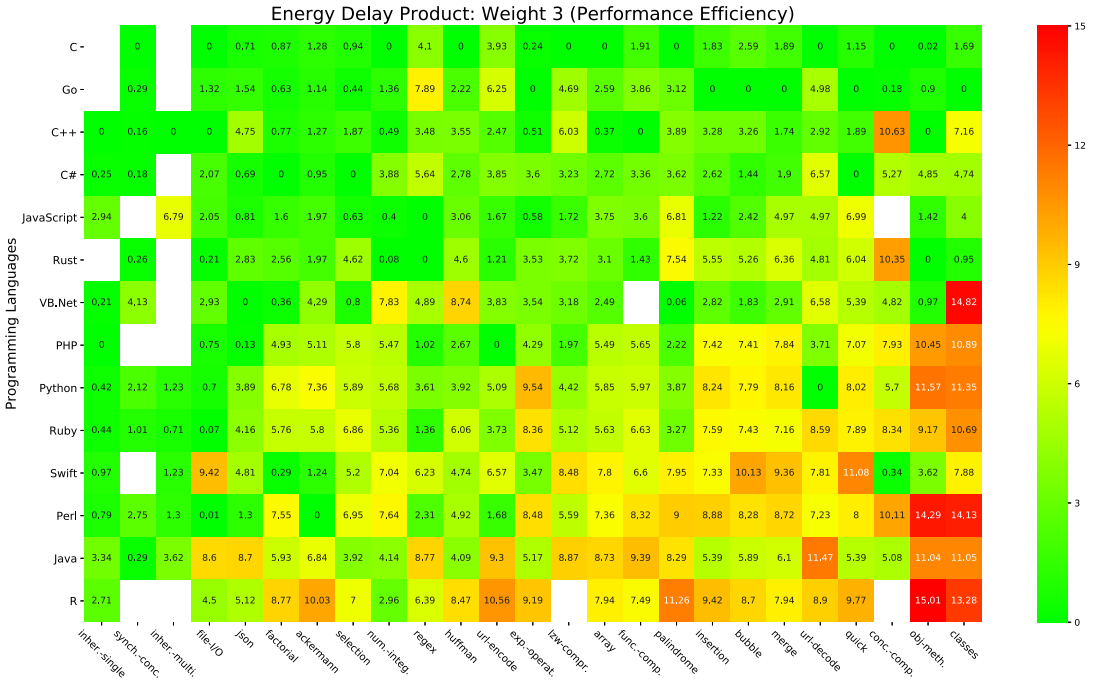


Figure 1: Server's Platform EDP Results

are excluded from our final results. Having full control over the OS' workload and background operations is hard, because, at any time, different daemons may operate. This could affect our calculations, too. Lastly, given that there exist several compiler and run-time versions of the programming languages we used, we cannot precisely calculate their impact on EDP.

Third, in Rosetta Code, not all tasks are implemented in all programming languages. For example, the *classes* and *call-an-object-method* tasks are not applicable to programming languages such as C, Rust, and Go. We have tried to keep the original snippets of Rosetta Code intact and apply only minor changes when needed e.g., adding a `main` function, changing from iterative to recursion, and using `structs`. Consequently, some of the tasks might not reflect their most efficient and optimal implementations, resulting to higher EDP.

External validity. External validity refers to the extent to which the results of our study can be generalized to other programming implementations. Here, we present the limitations of our study.

According to Sahin et al. empirical studies that use real applications (e.g., mobile ones) show different energy consumption results from studies that use micro benchmarks (i.e., traditional desktop software) [36]. This mostly occurs because desktop or data center software are CPU bound, whereas mobile applications are more interactive. In addition, for mobile devices, screen, radios, and sensors—and not the CPU—consume most of the device's battery. Admittedly, since our study's results are based on a benchmark, our findings could be different for real software.

Finally, we have evaluated the EDP of 25 programming implementations written in 14 programming languages, and running on three platforms. Thus, it is currently difficult for us to generalize our arguments for other programming languages and platforms.

3 RESULTS

In this section, we discuss the EDP results for the selected platforms and tasks. We also compare their EDP when the implementations run on different software platforms.

Before plotting the heat maps, we ranked the tasks based on their EDP. For each particular task, we took the lowest EDP value and we used it to normalize the measurements of similar tasks. Some of the resulting values varied a lot among for all the selected platforms. Therefore, we used the base ten logarithm to reduce the influence of extreme values or outliers in the data without removing them from the data set.

Figure 1 illustrates the logarithmic base 10 EDP results for the server platform when performance is more important than energy. The y and x axes show the programming languages and the tasks, respectively. To this end, entries with 0 in Figure 1, correspond to the programming languages that achieved the lowest score of EDP for particular tasks, as $0 = \log 1$ after normalizing. The green color in the heat map shows higher efficiency, in terms of EDP, while red shows lower efficiency. The programming languages' scores for each task are sorted from top to bottom and the tasks from left to right, starting from the lowest to the highest average EDP.

Due to space constraints, we only added one of the nine plotted heat maps. The remaining heat maps can be found in our GitHub repository [20].

3.1 RQ1. Which programming languages are the most EDP efficient and inefficient for particular tasks?

To answer this research question we, initially, we discuss the results of each platform by grouping them into categories. We identify in which categories, in Table 2, specific programming languages show efficient or inefficient results. Then, we present programming languages' EDP implications that cannot be grouped into categories. Note that for the tasks *classes* and *call-an-object-method*, if a non-object-oriented implementation achieved better results, we excluded it and we picked the next one that had a better EDP result.

Table 3 lists results by giving the task's name, the most (min) and least (max) EDP efficient programming language implementations for the relevant task, and the corresponding difference. The difference is shown as the number of times a particular implementation is more EDP efficient than the other, in terms of the base ten logarithm EDP value. The difference between the most and least efficient raw values differed significantly, even close to billions of times in some cases. For instance, if the EDP's raw value difference between the minimum and the maximum is equal to 10,000,000, that would result in a logarithm of 7. We therefore used the base ten logarithm of the following ratio:

$$\log(p/p_{min}) \tag{2}$$

Where p is the measurement and p_{min} is the minimum value of EDP for each task. If $x = \log(p/p_{min})$, the language giving p_{min} is x times more efficient than p's implementation in logarithmic terms, that is, 10^x times more efficient. Finally, we use show in Figure 2 the score of EDP results all programming languages, for each taask, using box plots.

3.1.1 Embedded System Results. **Grouped into Categories.** The collected results, in Table 3 for the embedded system, show that C and Rust provide better EDP results then Perl, Swift, VB.NET, and R for tasks located under the category of arithmetic, compression, and data structures. C also performs more efficiently for the tasks falling under the concurrency category, by being 4.79 times more efficient than Perl. In terms of file-handling, Rust offers the best efficiency against Swift by 4.27 times. JavaScript is the programming language that achieves 3.88 times better EDP results for the regular expression category, compared to Java. For almost all sorting tasks, Go implementations are 3.98 to 6.52 times more EDP efficient compared to Swift, R, and Ruby. For performing functional tasks, C++ is proven to be 7.96 times more efficient than Swift.

Uncategorized Implications. The most EDP efficient implementations for the tasks falling under the recursion category are provided by the Go, VB.NET, and Swift implementations. More specifically, Go performs 4.77 times more efficiently for *ackermann-function* than R, VB.NET outperforms R by 3.66 times for *palindrome-detection*, and Swift outruns PHP by 4.03 times for the *factorial* task. For the *url-encoding* and *decoding* tasks, C and PHP achieve EDP efficiency of 6.36 and 7.38 times more than Java and R, respectively. For the OO tasks, the most efficient implementations vary: 1) C++ outperforms Python 5.62 times for the *call-an-object-method* task; 2) JavaScript outperforms R 5.1 times for the *classes* task; 3) Ruby outperforms JavaScript 7.94 times for the *inheritance-multiple* task; and 4) C++ outperforms JavaScript 3.6 times for the *inheritance-single* task.

3.1.2 Laptop System Results. **Grouped into Categories.** Table 3 shows the results for the laptop platform where C performs 4.1 to 5.81 times more efficiently for the tasks grouped under arithmetic and compression. The file-handling category, which includes I/O operations, performs better in Rust's implementation, which is 4.44 times more EDP efficient than VB.NET's. Recursion category implementations (*ackermann-function*, *palindrome-detection*, and *factorial*) prove to be 4.42 to 5.06 times more EDP efficient when using the .NET framework (for C# and VB.NET) in contrast to R. For the regular expression category, JavaScript's pattern matching and replacing operations performs 4.4 times better than R's. Likewise, for the tasks *classes* and *call-an-object-method*, found under the OO category, JavaScript offers the most EDP efficient implementations by being 6.86 to 7.16 times more efficient than R and Perl, correspondingly. For the remaining tasks under the OO category (multiple and single inheritance), C++ has the best implementations. Also, C++ outperforms Perl for the functional category by being 5.62 times more EDP efficient.

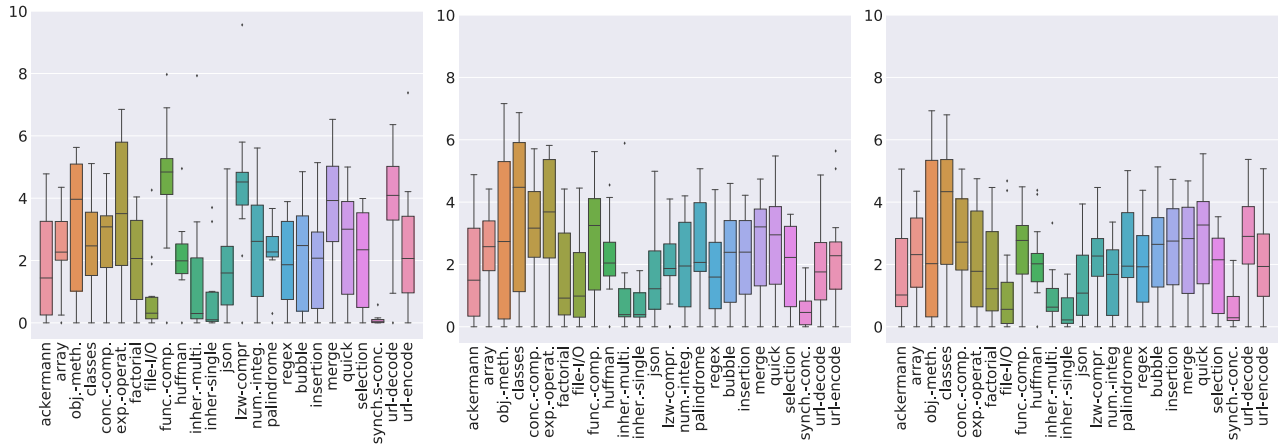
Uncategorized Implications. For the tasks *concurrent-computing* and *synchronous-concurrency*, C and C++ achieve the best efficiency against C++ and Java by being 6 and 1.88 times more efficient, respectively. C and PHP outperform Java by being 4.41 and 4.98 times more EDP efficient for the *array-concatenation* and *json* tasks, correspondingly. For sorting tasks like *insertion* and *selection*, JavaScript outruns R by 4.21 and 3.6 times. In addition, C# shows 4.6 and 5.48 times more EDP efficient results in contrast to R for *bubble* and quick sorting. Also, Go performs 4.74 times more efficiently for *merge* sorting compared to Swift. For the tasks of *url-decoding* and *encoding*, C++ and PHP perform 4.87 and 5.64 times better than R.

3.1.3 Server System Results. **Grouping into Categories.** Table 3 for server system, shows that C is the programming language with the most efficient EDP implementations for the compression, concurrency, and file-handling categories. Regarding the regular expression category, JavaScript offers the best performance by being 4.38 times more efficient than Java. For most of tasks falling under the sorting category, Go performs 4.68 to 5.55 times more efficiently against R and Swift. C++ outperforms JavaScript and Java for OO tasks such as single and multiple inheritance by 1.68 and 3.32 times respectively. In addition, C++ performs 4.48 times better for the functional category compared to Java.

Uncategorized Implications. For the server platform, Go and C achieve the best efficiency for the *exponentiation-operator* and *numerical-integration* tasks, respectively. C and VB.NET outrun Java by being 4.35 and 4.46 times more efficient for the data structures category, correspondingly. C#, Perl, and C achieve, 4.46, 5.06, and 5.01 times better results for the recursion category than R. Moreover, C performs more efficiently for *url-decoding* and PHP for *url-encoding* against Java and R.

Table 3: All System Tasks EDP

Task's Name	Embedded			Laptop			Server		
	Implementations		Logarithmic	Implementations		Logarithmic	Implementations		Logarithmic
	Min	Max	Ratio	Min	Max	Ratio	Min	Max	Ratio
exponentiation-operator	C	R	6.84	**C**	R	5.81	Go	Python	4.74
numerical-integration	Rust	Perl	5.6	**C**	VB.NET	4.2	C	VB.NET	3.35
huffman-coding	C	VB.NET	4.94	**C**	VB.NET	4.54	**C**	VB.NET	4.38
lzw-compression	Rust	Swift	9.56	**C**	Java	4.1	**C**	Java	4.46
concurrent-computing	**C**	Perl	4.79	C	C++	6	**C**	Rust	5.05
synchronous-concurrency	**C**	Perl	0.57	C++	VB.NET	1.88	**C**	VB.NET	2.12
array-concatenation	C	R	4.34	C	Java	4.41	C	Java	4.35
json	Rust	Swift	4.93	PHP	Java	4.9	VB.NET	Java	4.46
file-input-output	**Rust**	Swift	4.27	**Rust**	VB.NET	4.44	**C**	Swift	4.68
factorial	Swift	PHP	4.03	VB.NET	R	4.42	C#	R	4.46
ackermann-function	Go	R	4.77	C#	R	4.88	Perl	R	5.06
palindrome-detection	VB.NET	R	3.66	VB.NET	R	5.06	C	R	5.01
regular-expression	**JavaScript**	Java	3.88	**JavaScript**	Java	4.4	**JavaScript**	Java	4.38
merge-sort	**Go**	R	6.52	Go	Swift	4.74	**Go**	Swift	4.68
insertion-sort	JavaScript	R	5.13	JavaScript	R	4.21	**Go**	R	4.72
quick-sort	**Go**	Swift	5	C#	Swift	5.48	**Go**	Swift	5.55
selection-sort	**Go**	Ruby	3.98	JavaScript	R	3.6	C#	R	3.52
bubble-sort	**Go**	R	4.84	C#	Swift	4.6	**Go**	Swift	5.13
url-decoding	C	Java	6.36	C++	Java	4.87	C	Java	5.36
url-encoding	PHP	R	7.38	PHP	R	5.64	PHP	R	5.06
call-an-object-method	C++	Python	5.62	**JavaScript**	Perl	7.16	C++	R	6.93
classes	JavaScript	R	5.1	**JavaScript**	R	6.86	JavaScript	VB.NET	6.79
inheritance-multiple	Ruby	JavaScript	7.94	**C++**	JavaScript	5.89	**C++**	JavaScript	3.32
inheritance-single	C++	JavaScript	3.6	**C++**	Java	1.79	**C++**	Java	1.68
function-composition	**C++**	Swift	7.96	**C++**	Perl	5.62	**C++**	Java	4.48



(a) Embedded System **(b) Laptop** **(c) Server**

Figure 2: EDP box plots. The points show outliers. The vertical scale is the logarithmic ratio $\log_{10}(p/p_{min})$ where p is the measurement and p_{min} is the minimum of the measurements for that task, corresponding to the most EDP-friendly language.

C is the best in arithmetic, compression and concurrency, while C++, Go, and Rust are the runners-up. R, Perl, Swift, and Java show weak, overall, performance in terms of EDP. *Go is exhibiting best* EDP *results for sorting algorithms, Rust for file-*I/O*, JavaScript for pattern matching and replacing, and C++ for function-composition.*

3.1.4 Range of results. Figure 2 depicts the ranges of the EDP scores for each task that we measure in the three platforms. Apart from a few tasks, our measurements have wide ranges—we note that we are using a logarithmic scale.

For the embedded system, tasks such as *file-input-output*, *inheritance-single*, and *synchronous-concurrency* exhibit smaller EDP scores. This does not happen for the laptop and server platforms.

Our figures also show that outliers do exist for all three platforms. For instance, the embedded system's box plot has outliers for tasks such as *function-composition, huffman-encoding, lzw-compression, palindrome-detection*, and for *url-encoding*. Similarly, for the laptop platform *huffman-coding* and *lzw-compression* indicate that specific programming languages can offer much more gains in terms of EDP.

Table 4: Programming Languages Average Weighted EDP Ranking

Rank	Embedded			Laptop	Server
	$w = 1$	$w = 2$	$w = 3$	$w = 1, 2, 3$	$w = 1, 2, 3$
1	C	C	C	C	C
2	C++	C++	C++	Go	Go
3	Go	Go	Go	C++	C++
4	Rust	Rust	Rust	JavaScript	C#
5	C#	C#	JavaScript	Rust	JavaScript
6	VB.NET	JavaScript	C#	C#	Rust
7	JavaScript	VB.NET	VB.NET	VB.NET	VB.NET
8	PHP	PHP	PHP	PHP	PHP
9	Ruby	Ruby	Ruby	Ruby	Python
10	Python	Python	Python	Swift	Ruby
11	Perl	Perl	Perl	Python	Swift
12	Java	Java	Java	Perl	Perl
13	Swift	Swift	Swift	Java	Java
14	R	R	R	R	R

3.2 RQ2. Which types of programming languages are, on average, more EDP efficient and inefficient for each of the selected platforms?

We provide an EDP ranking for the programming languages—based on all tasks average EDP score—on our platforms. In addition, we discuss how EDP weights influence our results for each platform.

Overall Ranking. Table 4 illustrates for each platform the ranking among the programming languages' average EDP and the influence of their weights; where this is applicable. For all the selected platforms, compiled programming languages such as C, C++, and Go are ranked on top by offering the best EDP implementations on average. Rust is also ranked among the most efficient EDP programming languages for the embedded system, but it drops for the laptop and server platforms. Swift is the only one, from the compiled programming languages, that shows weak performance.

From the semi-compiled programming languages, the .NET framework's implementations (C# and VB.NET) score better against the interpreted languages, but remain less efficient than the compiled languages. Java is ranked as the most inefficient among the semi-compiled languages, while C# as the most efficient.

The interpreted programming languages are the ones offering on average lowest performance, for all platforms, and appear at the bottom ranks in the Table 4. Among them, JavaScript is the one being the most EDP efficient while R is the most inefficient.

Weights impact on EDP. By using different weights in the EDP, we force programming languages with low execution time but higher energy consumption to result in lower EDP score compared to the ones having low energy consumption but higher run-time performance. For instance, with $w = 1$ JavaScript achieves a logarithmic EDP score 3.19 for the *numerical-integration* case while C# achieves 1.68, which makes it more efficient compared to JavaScript. However, when $w = 2$ or $w = 3$, JavaScript's EDP changes to 3.35 and 3.52 while for C# it changes to 3.03 and 4.38, respectively. This denotes that C# is consuming less energy but is slower than JavaScript.

By looking at the results presented in Table 4, we see that only the embedded system's average EDP scores were affected by the changes in the weights. For instance, C#, VB.NET, and JavaScript were influenced after raising the run-time performance to the second and third power. Specifically, this denotes that JavaScript is much faster, on average, compared to C# and VB.NET but more energy demanding—since before raising the run-time performance C# and VB.NET had lower EDP.

Compiled languages are more EDP *efficient compared to the interpreted ones. Among the compiled, semi-compiled, and interpreted languages, the best EDP is obtained by C, C#, and JavaScript, respectively. Raising the* EDP *performance exponent to 2 or 3 affects the ranking of the embedded platform's tasks.*

3.3 RQ3. How much does the EDP of each programming language differ among the selected platforms?

We investigate how much a programming language's average EDP differs across the measurement platforms by using a non-parametric statistical test, the Wilcoxon's signed-rank test. To do that, we developed a script to carry out pairwise statistical analysis for the average EDP that a programming language scored for all the tasks, between two of the platforms each time. Compared to **RQ1** and **RQ2**, we used the raw values instead of logarithms, as the test takes into account the rank of the differences and not their magnitude. Our null hypothesis follows.

Hypothesis H_0: *A programming language's average* EDP, *does not have a statistically important difference between the measurement platforms.*

Let $\mathcal{L}$ be a programming language, and $\mathcal{P}_1, \mathcal{P}_2$, and $\mathcal{P}_3$ (where $\mathcal{P}_1 \neq \mathcal{P}_2 \neq \mathcal{P}_3$) the selected platforms of our experiment. Also, we pick the average EDP for all the tasks and we compare them in pairs such as: $(\mathcal{P}_1, \mathcal{P}_2)$; $(\mathcal{P}_1, \mathcal{P}_3)$; $(\mathcal{P}_2, \mathcal{P}_3)$. Since each platform's results were used twice in a comparison (once for each other platform), we used the Bonferroni correction to counteract the multiple comparison problem. Therefore if the test's p-value for the pair $\mathcal{P}_x$ and $\mathcal{P}_y$ (where $x, y \in \{1, 2, 3\}$ and $x \neq y$) satisfies the condition $p < (0.01/3)$, as there are 3 platforms pairs for each language, the difference between the average of $\mathcal{P}_x$ and $\mathcal{P}_y$ is statistically significant for the $\mathcal{L}$ programming language. Otherwise, if $0.01/3 \leq p < 0.05/3$, the statistical significance of the difference between averages is weaker. If $p \geq 0.05/3$, the null hypothesis cannot be rejected. Table 5 illustrates results after the pairwise statistical test of two platforms at a time for all the programming languages. The collected results show that we can reject the null hypothesis only for two cases and a third with weaker statistical significance (highlighted in figure 5).

Embedded and Laptop. The results illustrate that there is significant difference between the average EDP of the embedded and laptop platforms for C and Swift. In addition, the results show a weaker evidence that the average EDP is different only for a single instance among the programming languages, that is C++, for which the null hypothesis cannot be rejected.

Embedded and Server. For the embedded and server platforms there is no strong statistical evidence for the difference in average EDP.

Table 5: Wilcoxon's Pairwise Sum Ranking

Platforms	C#	C	C++	Go	JavaScript	Java	Perl	PHP	Python	R	Ruby	Rust	Swift	VB.NET
Embedded–Laptop	0.09	0.00	0.01	0.08	0.02	0.56	0.62	0.12	0.15	0.54	0.10	0.05	0.00	0.19
Embedded–Server	0.39	0.07	0.22	0.29	0.20	0.47	0.91	0.39	0.40	0.97	0.35	0.37	0.04	0.25
Laptop–Server	0.19	0.27	0.14	0.43	0.11	0.31	0.47	0.33	0.52	0.36	0.24	0.28	0.07	0.42

Laptop and Server. Between these two platforms we found no strong significant difference for any instance.

There is a significant difference between the average EDP*, in some case, of the embedded and laptop platforms.*

4 DISCUSSION

We investigate the root causes of the results of Section 3 by digging into the source code of the tasks that showed better EDP results. We focus on the programming languages that achieved the most EDP efficient results. Additionally, we explain how the collected results from the heatmaps can be useful for practitioners in developing energy-aware applications.

4.1 Champions

Concurrency. Best efficiency in EDP is achieved in programming languages that rely on libraries implementations for concurrency. For instance, C, that achieved far better EDP, uses OpenMP [26] and libco [8] to execute the tasks found under the concurrency categories; in contrast to Perl's and PHP's thread libraries.

Regular Expressions. JavaScript produces the most EDP efficient results for *regular-expression* tasks, such as pattern matching and replacing. The reason behind this is that the V8 JavaScript engine (build in C++) achieves a speed-up for regular expressions, after the RegEx library is built on top of Irregexp, in combination with CodeStubAssembler [21]. Therefore, we conducted an experiment where we used `nvm` [9], a JavaScript version management for `node.js`, to install a JavaScript version that does not support the above-described libraries and related functionality (versions below 8.5.7). When using the version where the RegEx library does not use Irregexp and CodeStubAssembler, the EDP increased by 331%.

Object-Oriented Features. In addition, JavaScript, in most cases, achieves better results for the *classes* and *call-an-object-method* tasks. However, JavaScript is dynamically typed, i.e., types and type information is not explicit and attributes can be added to and deleted from objects on the fly. That means that object orientation under JavaScript is very different than object orientation in, say, C++, where the focus is on designing polymorphic types. Also, to access the types and properties effectively, V8 engine creates and uses the hidden classes, at run-time, to have an internal representation of the object to improve the property access time. In addition, once a hidden class is created for a particular object, the V8 engine shares the same hidden class among objects created in the same way [22].

File handling. Rust, compared to VB.NET, produces better EDP results for the *file*-I/O operations regarding the embedded and laptop platforms. We performed a small experiment—using the `strace` [14] Linux built-in command—and we found that VB.NET's intermediate code makes 14 times more system calls in total compared to Rust. According to Aggarwal et al. [4], when the system calls between two applications diverge significantly, it is possible that the applications' power usage will differ too. Moreover, the VB.NET implementation for this task takes 89% of its total execution time for `mmap` (creating a new mapping in the virtual address space for the current process files) and `nunmap` (deleting the mapping for the process when it is no longer needed). VB.NET is slower since it executes the `lseek` operation when writing in a file which is not the case for Rust. This might occurs because VB.NET's I/O-buffers are smaller than Rust's and requires more than a single write operation to write all the data in the file. In this way, Rust spends much less time for I/O system calls compared to VB.NET, resulting in faster execution time and thus lower EDP.

Functional Programming Features. C++ is the language exhibiting the best EDP efficiency for the *function-composition* task. The reason behind this is that C++ uses meta-programming through the Standard Template Library to compose functions at compile-time via the help of preprossesor. As a result, C++ has faster execution time and less energy consumption resulting in more efficient EDP compared to the other implementations.

4.2 Applications

We believe that a developer can consult our heat maps and use them as a guideline to develop more EDP efficient applications. For the development of a more complex application—that may combine more that one of the generic tasks evaluated in this study—a developer can choose the language or languages that will provide the best efficiency, in terms of EDP.

In the context of embedded systems, even though we showed in Section 3.2 that Java and Swift, the major programming languages for developing Android and iOS applications, are on average EDP inefficient, developers can use several efficient practices. For instance, when developing Android applications, practitioners can use the Native Development Kit [13] to incorporate native C and C++ code for heavy arithmetic, concurrent, and functional tasks.

In general, each programming language offers a number of different features such as dynamic and static binding, lazy, and eager evaluation, garbage collection, automatic counter references, strong and weak typing, and so on. Moreover, the different platforms' CPU architectures and resources could be also a factor of causing differences in EDP results. We do not know how these might affect the EDP. However, our results show certain programming languages implementations are more beneficial on selected tasks. Therefore, researchers could use these findings to identify which are the factors or features causing these outcomes and use them in developing EDP efficient programming languages.

5 RELATED WORK

To the best of our knowledge, this is the first study that assesses the EDP of commonly used programming tasks—when they are implemented in different programming languages and they run on three platforms (an embedded system, a laptop, and a server)—to provide developers with guidelines about the most efficient programming languages per case. In the following, we present and compare related work's results with our findings.

5.1 Across Programming Languages

Pankratius et al. pursued a controlled comparative experiment on Java and Scala developers, in a multicore environment, to evaluate factors, such as the performance of both languages [31]. One of their important findings refers to the fact that the functional paradigm does not lead to bad performance, but programs that use both functional and imperative styles can have improved performance. Additionally, we took into account object-oriented and functional features from programming languages and we found that the latter result in more efficient EDP.

Pereira et al. conducted an empirical study on 27 programming languages from *The Computer Language Benchmarks Game* and they compared them on energy, time, and memory matters [32]. Even though our methods have similarities with Pereira's et al. study, we, additionally, checked the EDP of our sample's implementations on three different platforms, using implementations from the Rosetta Code Repository. Similarly to us, they found that compiled languages are the fastest, whereas interpreted languages, such as JavaScript and PHP, are the most energy efficient in operations with regular expressions.

Close to our paper is the empirical study that Nanz and Furia conducted on the Rosetta Code Repository to compare the performance of eight popular programming languages, including C, Go, C#, Java, F#, Haskell, Python, and Ruby [30]. Here, we did not measure only performance, but we also used a power analyzer to run programming tasks on 14 different programming languages in order to compare their EDP. This study's findings agree with ours regarding compiled languages that have the best performance.

Finally, Meyerovich and Rabkin performed an empirical study by analyzing 200,000 SourceForge projects and asking almost 13,000 programmers to identify characteristics that lead them to select appropriate programming languages for implementing their software projects [28]. We compared the EDP of programming tasks performed in several programming languages.

5.2 Across Execution Platforms

Abdulsalam et al. conducted experiments on *workstations* and evaluated the energy effect of four memory allocation choices (`malloc`, `new`, `array`, and `vector`). They showed that `malloc` is the most efficient in terms of energy consumption and performance [1]. Chen and Zong worked on *smartphones* and showed, by using the Android Run Time environment (instead of Dalvik), that the energy and performance implications of Java are similar to C and C++ [11]. Finally, Rashid et al. worked on an *embedded system* and compared the energy and performance impact of four sorting algorithms written in three different programming languages (ARM assembly, C/C++, and Java) [33]. They found that Java consumes most energy and performs slowly against C/C++ and assembly. Similarly, we found that Java was slower in comparison with C/C++. In addition, we observed that Go had the lowest EDP for sorting algorithms.

Many empirical studies have assessed the impact of *coding practices* (e.g., the use of `for` loops, getters and setters, static method invocation, views and widgets, and so on) regarding energy consumption. Characteristically, Tonini et al. conducted a study on Android applications and found that the use of `for` loops with specified length and the access of class variables without the use of getters and setters can reduce the amount of the energy that the applications consume [40]. Linares-Vsquez et al. performed an analysis of 55 Android applications from various domains and reported the most energy consuming API methods [27]. For instance, they found that from 60% of the most energy-greedy APIs, 37% were related to the graphical user interface and image manipulation, while the remaining 23% were associated with databases. Finally, Sampson et al. proposed an approach called EnerJ that uses annotations to indicate particular data types that are involved in computations that can consume lower energy [37]. They prove that for a small number of type annotations added to the types of a Java program they can achieve more energy savings.

Contrary to previous works, we compare EDP of small programming tasks implemented in 14 programming languages that can run on three distinct computer platforms. Overall, our results show that compiled programming languages perform far more EDP efficiently compared to interpreted and semi-compiled. Our findings also indicate significant difference between the results of the laptop and the embedded system execution environments.

6 CONCLUSIONS

We examined programming languages' energy consumption and run-time performance implications for certain programming tasks, by using the EDP formula. We also investigated how our results vary among different computer platforms and showed that there is not a single winner for all cases. Based on our findings, we suggest that specific implementations are more EDP efficient than others with respect to the tasks they perform. So, developers can use:

- C for the development of computationally-intensive and concurrent tasks;
- Go for sorting tasks;
- Rust for I/O-intensive tasks, such as file handling;
- JavaScript for *regular-expression* tasks, such as string matching, substitution; and
- C++ for functional programming tasks, such as *function-composition*, at a low level of abstraction.

Regarding our future research directions, we will attempt to: 1) identify which are the programming language features that offer major implications on EDP, 2) dig into generated machine and intermediate code to point out how different compiler implementations or run-time engines can affect EDP, 3) consider more programming task categories, including network access operations and image processing, and 4) increase the variety of our test inputs.

ACKNOWLEDGMENTS

This work is funded by the SENECA project, a Marie Sklodowska-Curie Innovative Training Networks (ITN-EID) 642954 agreement.

REFERENCES

[1] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong. 2014. Program energy efficiency: The impact of language, compiler and implementation choices. In *International Green Computing Conference*. 1–6.

[2] S. Abdulsalam, Z. Zong, Q. Gu, and Meikang Qiu. 2015. Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency. In *6th International Green and Sustainable Computing Conference (IGSC '15)*. 1–8. https://doi.org/10.1109/IGCC.2015.7393699

[3] Acmeism. 2017. RosettaCodeData: RosettaCode Data Project. (nov 2017). Retrieved 2017-10-23 from https://github.com/acmeism/RosettaCodeData

[4] Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. 2014. The Power of System Call Traces: Predicting the Software Energy Consumption Impact of Changes. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering (CASCON '14)*. IBM Corp., Riverton, NJ, USA, 219–233.

[5] Peter Bailey. 2017. watts-up: Watts Up Pro power meter interface utility for Linux. (sep 2017). Retrieved 2017-10-23 from https://github.com/pyrovski/watts-up

[6] K. W. Cameron, Rong Ge, and Xizhou Feng. 2005. High-performance, power-aware distributed computing for scientific applications. *Computer* 38, 11 (Nov. 2005), 40–47. https://doi.org/10.1109/MC.2005.380

[7] Eugenio Capra, Chiara Francalanci, and Sandra A. Slaughter. 2012. Is Software "Green"? Application Development Environments and Energy Efficiency in Open Source Applications. *Inf. Softw. Technol.* 54 (Jan. 2012), 60–71.

[8] Tim Caswell. 2017. (nov 2017). Retrieved 2017-12-28 from https://github.com/creationix/libco

[9] Tim Caswell. 2018. nvm: Node Version Manager - Simple bash script to manage multiple active node.js versions. (jan 2018). Retrieved 2018-01-20 from https://github.com/creationix/nvm

[10] X. Chen and Z. Zong. 2016. Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. 485–492.

[11] X. Chen and Z. Zong. 2016. Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. 485–492. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.77

[12] Dell. 2018. Dell Vostro 470 Specs. (Mar 2018). Retrieved 2018-03-08 from https://www.cnet.com/products/dell-vostro-470-mt-core-i5-3450-3-1-ghz-4-gb-500-gb/specs/

[13] Developer.android. 2018. Android NDK | Android Developers. (2018). Retrieved 2018-01-20 from https://developer.android.com/ndk/index.html

[14] Die.net. 2018. strace(1): trace system calls/signals - Linux man page. (jan 2018). Retrieved 2018-01-11 from https://linux.die.net/man/1/strace

[15] Die.net. 2018. time(1) - Linux man page. (jan 2018). Retrieved January 24, 2018 from https://linux.die.net/man/1/time

[16] K. Eder. 2013. Energy transparency from hardware to software. In *2013 Third Berkeley Symposium on Energy Efficient Electronic Systems (E3S)*. 1–2.

[17] eLinux. 2017. RPI vcgencmd usage - eLinux.org. (aug 2017). Retrieved 2018-01-29 from https://elinux.org/RPI_vcgencmd_usage

[18] M.A. Ferreira, E. Hoekstra, B. Merkus, B. Visser, and J. Visser. 2013. Seflab: A lab for measuring software energy footprints. In *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*. 30–37.

[19] Stefanos Georgiou. 2018. Rosetta_Code_Research_MSR: Exploiting Programming Languages Energy Consumption. (mar 2018). Retrieved 2018-03-16 from https://github.com/stefanos1316/Rosetta_Code_Research_MSR/Scripts

[20] Stefanos Georgiou. 2018. Rosetta_Code_Research_MSR: Exploiting Programming Languages Energy Consumption. (mar 2018). Retrieved 2018-03-16 from https://github.com/stefanos1316/Rosetta_Code_Research_MSR/heatmaps

[21] Jakob Gruber. 2018. Speeding up V8 Regular Expressions. (jan 2018). Retrieved 2018-01-22 from https://v8project.blogspot.com/2017/01/speeding-up-v8-regular-expressions.html

[22] Michael Hablich. 2018. v8: The official mirror of the V8 Git repository. (jan 2018). Retrieved 2018-01-22 from https://github.com/v8/v8/wiki/Design-Elements

[23] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 12–21.

[24] M. Horowitz, T. Indermaur, and R. Gonzalez. 1994. Low-power digital design. In *Proceedings of 1994 IEEE Symposium on Low Power Electronics*. 8–11. https://doi.org/10.1109/LPE.1994.573184

[25] HP. 2018. HP EliteBook 840 G3 Notebook PC| HPÂő United States. (Mar 2018). Retrieved 2018-03-08 from http://www8.hp.com/us/en/products/laptops/product-detail.html?oid=7815294

[26] Tim Lewis. 2017. OpenMP Home. (dec 2017). Retrieved 2017-12-28 from http://www.openmp.org/

[27] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 2–11.

[28] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical Analysis of Programming Language Adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 1–18.

[29] Mike Mol. 2016. Rosetta Code. (Jan 2016). Retrieved 2017-10-23 from http://rosettacode.org/wiki/Rosetta_Code

[30] S. Nanz and C. A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 778–788.

[31] Victor Pankratius, Felix Schmidt, and Gilda Garretón. 2012. Combining Functional and Imperative Programming for Multicore Software: An Empirical Study Evaluating Scala and Java. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 123–133.

[32] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE '17)*. ACM, New York, NY, USA, 256–267. https://doi.org/10.1145/3136014.3136031

[33] Mohammad Rashid, Luca Ardito, and Marco Torchiano. 2015. Energy Consumption Analysis of Algorithms Implementations. *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* 00 (2015), 1–4.

[34] Raspberry.org. 2018. Raspberry Pi 3 Model B. (Mar 2018). Retrieved 2018-03-08 from https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[35] Guenter Roeck. 2018. lm-sensors. (jan 2018). Retrieved 2018-01-20 from https://github.com/groeck/lm-sensors

[36] Cagri Sahin, Lori Pollock, and James Clause. 2016. From Benchmarks to Real Apps. *Journal of Systems and Software* 117, C (July 2016), 307–316. https://doi.org/10.1016/j.jss.2016.03.031

[37] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/1993498.1993518

[38] Tiobe. 2017. TIOBE Index | TIOBE - The Software Quality Company. (Oct 2017). Retrieved 2017-10-23 from https://www.tiobe.com/tiobe-index/

[39] Tiobe. 2018. Programming Languages Definition | TIOBE - The Software Quality Company. (jan 2018). Retrieved 2018-01-20 from https://www.tiobe.com/tiobe-index/programming-languages-definition/

[40] A. R. Tonini, L. M. Fischer, J. C. B. d. Mattos, and L. B. d. Brisolara. 2013. Analysis and Evaluation of the Android Best Practices Impact on the Efficiency of Mobile Applications. In *2013 III Brazilian Symposium on Computing Systems Engineering*. 157–158.

[41] WattsUpMeter. 2017. Watts up? Products: Meters. (Oct 2017). Retrieved 2017-10-23 from https://www.wattsupmeters.com/secure/products.php?pn=0