Analyzing Programming Languages Energy Consumption: An Empirical Study

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

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

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

ABSTRACT

Motivation: Shifting from traditional monolithic approaches to more cutting edge ones such as micro-services—where different services can be implemented and communicate in different programming languages—implies new challenges in terms of energy usage.
Goal: In this preliminary study, we aim to identify energy implications of small, independent tasks developed in different programming languages; compiled, semi-compiled, and interpreted ones.
Method: To achieve our purpose, we collected, refined, compared, and analyzed a number of implemented tasks from Rosetta Code, that is a publicly available repository for programming chrestomathy.
Results: Our analysis shows that among compiled programming languages such as C, C++, Java, and Go offers the highest energy efficiency for all our tested tasks compared to C#, VB.NET, and Rust. Regarding interpreted programming languages PHP, Ruby, and JavaScript exhibit the most energy savings compared to Swift, R, Perl, and Python.

CCS CONCEPTS

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

KEYWORDS

Energy Efficiency, Energy Optimization, Programming Languages

ACM Reference format:
Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. 2017. Analyzing Programming Languages Energy Consumption: An Empirical Study. In *Proceedings of Panhellenic Conference on Informatics, Larrisa, Greece, September 2017 (PCI '17),* 6 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

The increasing demands on services and computational applications from ICT-related products are major factors that contribute to the increase of energy consumption.[1] Recent research conducted by Gelenbe and Caseau [7] and Van Heddeghem et al. [14] indicates a rising trend of the IT sector energy requirements, which are expected to reach 15% of the world's total energy consumption by 2020.

Traditionally, most of the studies, for energy efficiency, have considered energy consumption at hardware level. However, there is much of evidence that software can also alter energy dissipation significantly [2, 5, 6]. Therefore, many conference tracks (*e.g.* GREENS,[2] eEnergy)[3] have identified the energy-efficiency at the software level as an emerging research challenge aiming to reduce energy consumption of a software through software development.

Nowadays, the software development has shifted from traditional monolithic architectures to a more agile one including micro-services. The feature of this approach is the development of independent and reusable small services in a variety of programming languages. However, the energy impact of different tasks implemented in different programming languages is still new and unknown to researchers and developers.

To identify trends and possible gains regarding the reduce of energy consumption during software development, we conducted an empirical study aiming at eliciting energy usage, starting from small tasks implemented in a variety of well-known and most used programming languages. To this end, our results show *which* of the interpreted and compiled programming languages offer more energy efficient implementations for specific tasks. Moreover, we show the negative impact if choosing an inefficient implementation.

The remainder of this paper is organized as follows. Section 2 describes our experimental setup; dataset refining, the software and hardware tools, and our methodology for retrieving results. In Section 3, we present our preliminary results and in Section 4 we discuss potential threats to validity. In Section 5, we list prior work done in the field and compare it with ours. Finally, we conclude in Section 6 and we present future research directions.

2 EXPERIMENTAL SETUP

In this Section, we describe our approach for conducting our experiment and for retrieving measurements. Initially, we provide information about the obtained dataset and the way we selected

[1]Although in the physical sense energy cannot be consumed, we will use the terms energy "consumption", "requirement", and "usage" to refer to the conversion of electrical energy by ICT equipment into thermal energy dissipation to the environment. Correspondingly, we will use the term energy "savings", "reduction", "efficiency", and "optimization" to refer to reduced consumption
[2]http://greens.cs.vu.nl/
[3]http://conferences.sigcomm.org/eenergy/2017/cfp.php

Table 1: Programming Languages, Compilers and Interpreters

	Programming Languages	Compilers and Interpreters version
Compiled	C, C++	gcc version 6.3.1 20161221-
	Go	go version go1.7.5
	Rust	rustc version 1.18.0
Semi-Compiled	VB.NET	mono version 4.4.2.0 (vbnc)
	C#	mono version 4.4.2.0 (mics)
	Java	javac version 1.8.0_131
Interpreted	JavaScript	node version 6.10.3
	Perl	perl version 5.24.1
	PHP	php version 7.0.19
	Python	python version 2.7.13
	R	Rscript version 3.3.3
	Ruby	ruby version 2.3.3p222
	Swift	swift version 3.0.2

our tasks and refine it. Furthermore, we explain our experimental setup, the hardware and software tools we used.

2.1 Dataset

In the context of this study, we used the Rosetta Code,[4] which is a publicly available programming chrestomathy site that offers 851 tasks, 230 draft tasks, and a collection of 658 different programming languages. In general, not all tasks are implemented, and not all tasks are possible to implement in all languages. For our study, we cloned a Github repository[5] that contains all the currently implemented tasks introduced in the Rosetta Code website.

To select popular programming languages, we consulted the website of tiobe,[6] a software quality company. Tiobe uses a search query for index rating of the most popular programming languages around the web on a monthly basis. This query is based on a formula[7] that uses the highest ranked search engines (according to Alexa)[8] and a number or requirements enlisted for the programming languages. We decided to chose the top 15 programming languages enlisted for June 2017. From the current list, we excluded programming languages such as Delphi (not available for Linux OS we are using) and Assembly (different implementations between processor architectures). In contrast, we included Rust in our dataset that is a memory safe programming language and is gaining vast popularity in the web. Therefore, we ended up with 14 programming languages as it is illustrated in Table 1.

To select the examined tasks, we developed a shell script (see Subsection 2.2.2) to identify which of the 851 tasks offer the most implementations for the programming languages of our selection. After launching our script, we obtained around 29 different tasks. For the context of our preliminary study, we chose only nine tasks implemented in the most of the programming languages of our selection. The selected tasks were: *array-concatenation, classes* (creating an object and calling a method to print a variable's value), *url-encoding and decoding, bubble-, quick-, insertion-, merge-, and selection- sorting algorithms.* Moreover, to further refine our dataset we used the following steps:

- Some of the tasks offered more than one implementation for the same programming language. Thus, we had to browse manually through each directory and remove them until we had only one that is consistent with the other implementation. For instance, when most of the implemented tasks used iterative implementation, we removed the ones using recursion.
- The Java file names and their public names where different which resulted in compilation error. Thus, we had to manually change them.
- Some of the implementations did not have main classes, or the same data with other tasks. Therefore, we changed the source code to offer consistency.
- For some programming languages that do not offer the class option such as C and Go, we used `structs`.
- Some of the tasks were relatively small and finished faster than a second which makes it impossible for our power analyzer to capture those results. Therefore, we added all the selected tasks in an iteration loop of a million times.

After applying the above modifications on our dataset, we categorized our programming languages in three main categories, namely, compiled, semi-compiled, and interpreted (see Table 1). For the programming languages which offer a semi-compiled approach such as Java, VB.NET, and C#, we added them under the category of compiled languages for our experiments. In addition, we compared the compiled and semi-compiled implementations while having scenarios with and without compiler optimizations.

2.2 Hardware and Software components

2.2.1 Hardware Components. The physical tools we used comprise: 1) portable personal computer (HP EliteBook 840 G3),[9] 2) real-time electricity usage monitoring tool, and 3) embedded device. The real-time power usage tools we used is the Watts Up Pro (WUP).[10]

In general, there are two venues for retrieving energy consumption from a computer-based system. On the one hand, this is achievable by indirect energy measurements through estimation models or performance counters, core component of software monitoring tools. On the other hand, via direct measurement, hardware power analyzers and sensors. However, each of these approaches has its own pitfalls. The direct approach , *i.e.,* hardware components, offers coarse-grained measurements for the whole systems' energy consumption and low sampling rate. The indirect approach , *i.e.,* software components, suffers from inaccuracy, lack of interoperability, and additional system overhead, while using indirect measurements. In our research, we decided to use a direct approach such as WUP since it does not have software constrains and is relatively cheap to buy.

[4]http://rosettacode.org/wiki/Rosetta_Code
[5]https://github.com/acmeism/RosettaCodeData
[6]https://www.tiobe.com/tiobe-index/
[7]https://www.tiobe.com/tiobe-index/programming-languages-definition/
[8]http://www.alexa.com/

[9]http://www8.hp.com/us/en/products/laptops/product-detail.html?oid=7815294#!tab=specs
[10]https://www.wattsupmeters.com/secure/products.php?pn=0

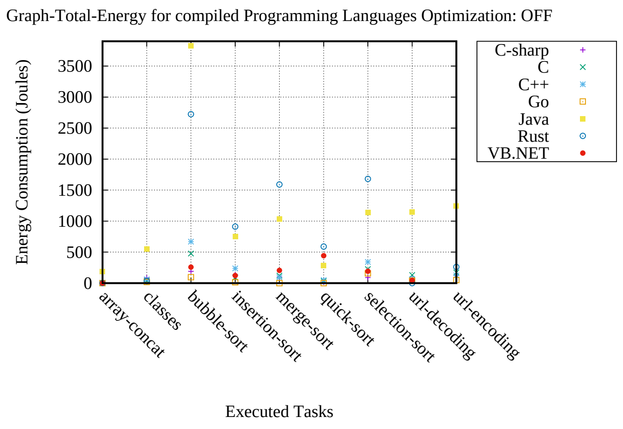

Figure 1: Energy Consumption for Compiled Programming Languages Optimization: OFF

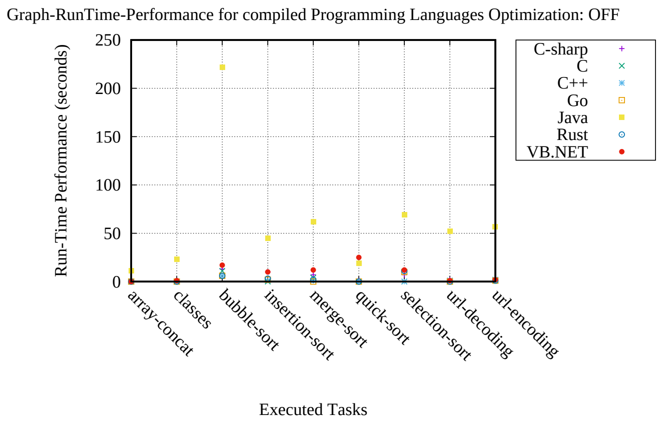

Figure 2: Run-time Performance for Compiled Programming Languages Optimization: OFF

In regards to WUP, it offers accuracy of ±1.5% and as minimum sampling rate of a second. In order to retrieve power-related measurements from the WUP, we used a Linux-based interface utility available in a Git repository.[11] This software helped us to retrieve measurements such as timestamps, watts, volts, amps, and so on through a mini USB interface after we integrated its code in our script that runs all the tasks. In order to avoid additional overhead in our measurements, we connected WUP mini-USB on a Raspberry Pi[12] to retrieve power consumption from our test-bed.

[11]https://github.com/pyrovski/watts-up
[12]https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

2.2.2 Software Components. To extract data, manage, and use our Rosetta Code Repository, we developed a number of shell scripts as enlisted below, which are publicly available on our Git repository.[13]

- **script.cleanAll**: removes the current instance of tasks in the current working directory and copy the new one found from in the parent directory.
- **script.findCommonTasksInLanguages**: provides a list of tasks with the number of existing implementations in different languages.
- **script.createNewDataSet**: filters the Rosetta Code current dataset and removes programming languages and tasks not added as command line arguments.
- **script.fromUpperToLower**: changes the current instance of tasks directories and files from upper to lower case.
- **script.compileTasks**: compiles all tasks found under the tasks' directory and produces error reports if a task fails to compile.
- **script.executeTasksRemotely**: executes all the tasks' implementations found under tasks directory. Moreover, it sends command to WUP, retrieves measurements and stores them on remote host, through *ssh*, in order to start retrieving measurements for each test case.
- **script.createPlottableData**: creates a single file that enlists all the executed tasks with the energy consumption for each implementation. In addition, we used NTP[14] to synchronize both system clocks which helped us to map our results of run-time performance and energy consumption.
- **script.plotGraphs**: after retrieving our data we use this script to plot our graphs. For plotting our graphs we used Gnuplot,[15] an open-source general purpose pipe-oriented plotting tool.

Note that most of the scripts offer the *−help* option that shows a list of available command line arguments and options. In addition, we provide a README.MD file, available in our repository, as a guideline for using our scripts and reproducing the obtained results.

2.3 Retrieving Energy Measurements

As an initial step for our experiment, we shut down background processes, as suggested by Hindle [8], found in modern OS (Operating System) such as disk defragmentation, virus scanning software, CRON jobs, automatic updates, disk indexing, document indexing, RSS feed updates, and so on to minimize possible noise interferences in our measurements. Making the following steps, we reduced our platform's idle power consumption from 8,6 to 5.8 watts on average.

To prevent addition noise in our results, we estimated that is necessary to wait for a short period to reach a *stable condition* (where the WUP shows a stable idle time) which is close to five minutes [3].

After reaching the *stable condition*, we launched our main script *i.e.*, **script.executeTasksRemotely**, that executes all the tasks implemented in different programming languages. Before executing a task, the execution script sends a command to the remote host

[13]https://github.com/stefanos1316/Rosetta-Code-Research
[14]http://www.ntp.org/
[15]http://www.gnuplot.info/

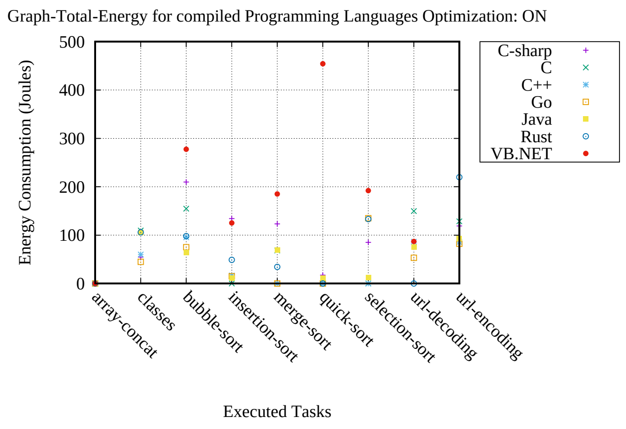

Figure 3: Energy Consumption for Compiled Programming Languages Optimization: On

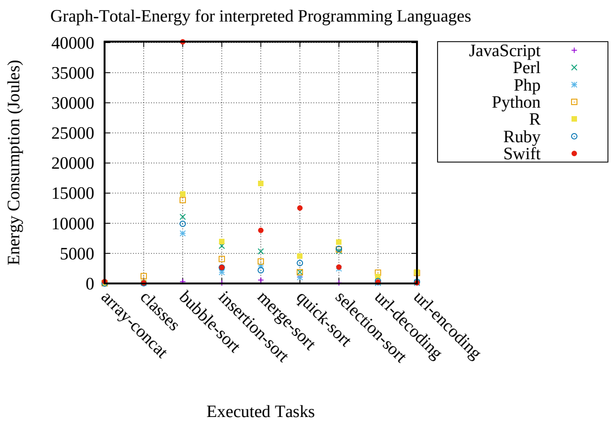

Figure 5: Energy Consumption for Interpreted Programming Languages

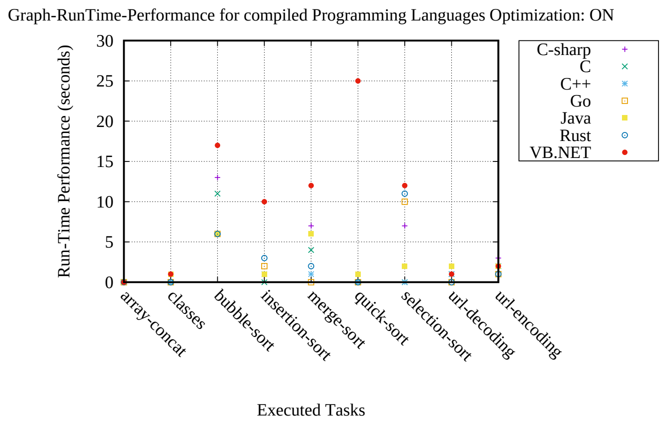

Figure 4: Run-time Performance for Compiled Programming Languages Optimization: On

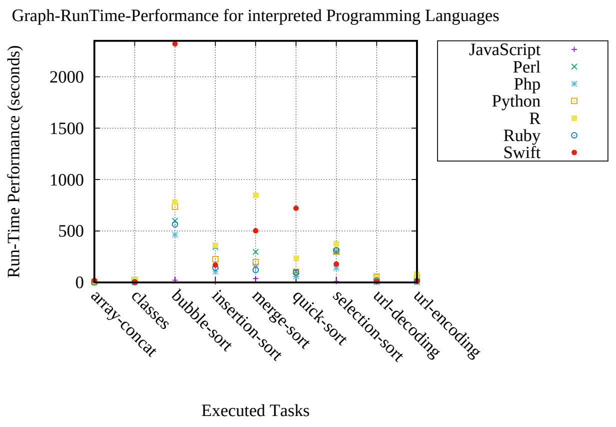

Figure 6: Run-time Performance for Interpreted Programming Languages

,*i.e.*, Raspberry Pi, through a password-less SSH connection to start collecting power consumption measurements from WUP for the currently executing task. In addition, the local host retrieves run-time performance measurements through command time[16] and stores them in timestamped directories which we analyze later. Between each execution of a task, we added a sleep[17] period of three minutes. The time gap exists to ensure that our experimental platform reached a *stable condition* and to avoid unnecessary noise in our measurements. For example, to ensure the platform's CPU is cooled down and the fan is no longer consuming more power.

[16]https://linux.die.net/man/1/time

[17]http://man7.org/linux/man-pages/man3/sleep.3.html

3 RESULTS AND DISCUSSION

For some tasks, the execution time waas less than a second. This fact explains the zeros shown in our graphs since it is impossible for WUP to collect measurements that have duration less than a second of time interval. Figure 1 illustrates the total energy required for each task implementation to execute with no compiler optimization. The results depict that Java and Rust have the highest energy consumption among the compiled programming languages while Go has the lowest. Also, the results in Figure 2 show that energy consumption is directly affected by the run-time performance in all cases except for Rust. In case of Rust, the energy consumption

Table 2: Comparative Results show the percentage of increased energy usage while using the inefficient implementation in compare to the efficient

Tasks Name	Implementations		Comparative
	Efficient	Inefficient	Results
Array-Concat	PHP, Ruby	Swift	888.25%
Classes	PHP	Python	1616.36%
Bubble	JavaScript	Swift	12694%
Insertion	JavaScript	Perl	9430.25%
Merge	JavaScript	R	2894.81%
Quick	PHP	Swift	1212.23%
Selection	JavaScript	R	6657.71%
Url-Decode	PHP	Python	2963.05%
Url-Encode	PHP	R	3239.79%

is kept relatively high the run-time performance is not affected negatively like in Java.

Figure 3 shows the total energy dissipation from the beginning until the end of a task (while making use of the compiler optimizations). The obtained results show the use of compiler optimizations reduces energy usage in most of the cases while it increases it in others. For C#, the energy usage of *quick-, insertion-, bubble- sort* and *url-decode* increased in range of 1% to 10% whereas for *merge-, selection- sort, classes* and *url-encode* reductions were between 3.8% and 36.6%. C's *gcc -O3* achieved energy reductions ranging from 43.36% to 99.6% for the majority of tasks apart from *url-decoding* where the energy increased to 14.7%. In the case of C++, the *g++ -O3* resulted in energy savings between 0.9% to 84.71% for all tasks except for the *url-decoding* that introduced increased energy usage of 33.33%. For Go most of sorting algorithms and *url-decoding* energy usage reductions were ranging from 14.39% to 31.35% while for *insertion-sort* and *url-encoding* increased between 15.38% to 62%. Java was the only programming language with energy reduction for all tasks, in range of 6% to 98.4%. In the case of Rust, energy requirements for all tasks, reduced in range of 15.8% to 97.7% apart from the *classes* task where the energy usage increased to 15.4%. Regarding VB.NET, the compiler optimization had very small impact on energy consumption; less than 10%.

For the interpreted programming languages, we can see that the energy consumption among them is significantly different as depicted in Table 2. The most inefficient case is Swift that consumes 12694% more energy compares to JavaScript. In general, PHP, Ruby, and JavaScript achieved the most energy efficiency compared to Swift, R, Perl, and Python where their implementations contributed to the highest energy consumption. In terms of run-time performance, Figure 6 results show the energy consumed by the interpreted tasks (see Figure 5) have a relationship with the execution time.

4 THREATS TO VALIDITY

Internal: In order to avoid additional overhead in our experimental platform, we used a remote host to collect our results. Therefore, the need of a wireless connection was necessary, which might result on additional energy requirements (by making use of the SSH to start and stop the WUP). Moreover, we cannot have full control of our OS workloads and background operations. Therefore, is possible that some daemons might start running while testing our experiment.
External: Our real-time power analyzer offers minimum sampling interval of a second. In Figures 1, 3, and 5 the energy dissipation results to zero can be interpreted as follows. When the tasks execution is less than a seconds, this makes it impossible for WUP to capture such measurements.

5 RELATED WORK

Most empirical studies evaluate software projects from particular programming language families. Here, we count the energy consumption of programming tasks across 14 programming languages. To the best of our knowledge, this is the first study that assesses the energy consumption in different programming languages using the Rosetta Code Repository. In the following, we present related work to our topic and compare our results with the results from previous studies.

5.1 Programming Languages

Studies regarding the strengths and weaknesses of different programming languages can help developers to decide *which* programming language they will use to perform specific programming tasks. For instance, if programmers aim at the scalability and performance of their systems, they use functional programming most of the times. On the other hand, when they want to develop programs with high modularity, they use object-oriented programming languages.

Closest to our paper is the empirical study that Nanz and Furia conducted on the Rosetta Code Repository to compare the efficiency of eight popular programming languages, including C, Go, C#, Java, F#, Haskell, Python, and Ruby [11]. Contrary to this work, we used a power analyzer to run programming tasks on 14 different programming languages in order to compare the energy consumption at runtime.

In addition, Meyerovich and Rabkin conducted an empirical study by analyzing 200,000 SourceForge projects and asking almost 13,000 programmers to identify characteristics that lead the latter to select appropriate programming languages in business level [10]. However, this study is a survey on the adoption of programming languages in the industry. Our goal here is different. We compare the energy consumption of programming tasks performed in several programming languages.

5.2 Energy Consumption and Performance

Several researchers have investigated the energy efficiency and runtime performance impact over different programming languages. Also, a significant amount of works have taken into account the execution environment where the programs can run efficiently.

In particular, Abdulsalam et al. conducted experiments on workstations [1], whereas Rashid et al. on an embedded system [12] and Chen and Zong on smart-phones [4]. Abdulsalam et al. evaluated the energy effect of four memory allocation choices (`malloc`, `new`, `array`, and `vector`) and they showed that `malloc` is the most efficient in terms of energy and performance [1]. Chen and Zong 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++ [4]. Finally, Rashid et al. compared the energy and performance impact of four sorting algorithms written in three different programming languages (ARM assembly, C/C++, and Java). They found that Java consumes the most energy [12]. From all these studies it seems that Java and Python consume a lot of energy and perform slowly in comparison with C/C++ and Assembly.

Additionally, 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 [13]. Furthermore, in their study, Linares-Vsquez et al. performed analysis over 55 Android applications from various domains and they reported the most energy consuming API methods [9]. For instance, they found that the 60% of the energy-greedy APIs, 37% were related to the graphical user interface and image manipulation, while the remaining 23% were associated with the database.

Contrary to previous works, here we compare energy consuming programming tasks in more than 14 programming languages. Our results show that significant diverge with respect to energy consumption exist for interpreted programming languages. Moreover, we provide comparison in programming languages such as Go, Rust, VB.NET, and C# which is not available in prior works.

6 CONCLUSIONS AND FUTURE WORK

Nowadays, software development has been shifted from monolithic to more agile architectures, such as micro-services. This imposes the energy efficient development of independent and reusable small services in a variety of programming languages. Goal of this paper is to compare the energy consumption of specific programming tasks in different programming languages and identify which languages are appropriate to be used in modern services. In brief, we conducted an empirical study using a power analyzer to measure the energy consumption of several programming tasks found in the Rosetta Code Repository, for 14 popular programming languages.

According to our findings, the average energy consumption—for the tested tasks—of compiled programming languages seems to be much lower compared to this of the interpreted ones. Overall, our experiments revealed that VB.NET and Swift are the most inefficient programming languages among the compiled and interpreted, respectively. In particular, compiled programming languages, such as C and C++ ,that have the optimization option enabled, record low average energy usage, whereas Go presents the lowest total energy consumption for sorting algorithms. Regarding interpreted programming languages, JavaScript indicates the highest level of energy–efficiency. On the other hand, Swift seems to consume the most energy in total.

As far as future work is concerned, we would like to test all the 29 collected tasks and, furthermore, to implement and evaluate additional ones, including exception handling, and particular tasks for functional programming. Moreover, we will test the collected tasks in different CPU architectures, such as AMD and ARM. We, also, plan to collect resource usages to identify possible relationships between programming languages and resources. To this end, we expect that the obtained results can shed light on how to efficiently develop larger and more complex applications.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Vasiliki Efstathiou for her valuable help to complete this paper.

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. https://doi.org/10.1109/IGCC.2014.7039169

[2] 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.

[3] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1855840.1855861

[4] 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

[5] K. Eder. 2013. Energy transparency from hardware to software. In *2013 Third Berkeley Symposium on Energy Efficient Electronic Systems (E3S)*. 1–2. https://doi.org/10.1109/E3S.2013.6705855

[6] 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.

[7] Erol Gelenbe and Yves Caseau. 2015. The Impact of Information Technology on Energy Consumption and Carbon Emissions. *Ubiquity* 2015 (June 2015), 1:1–1:15. https://doi.org/10.1145/2755977

[8] 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. https://doi.org/10.1145/2597073.2597097

[9] 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. https://doi.org/10.1145/2597073.2597085

[10] 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 &#38; Applications (OOPSLA '13)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/2509136.2509515

[11] 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. https://doi.org/10.1109/ICSE.2015.90

[12] 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. https://doi.org/doi.ieeecomputersociety.org/10.1109/ESEM.2015.7321198

[13] 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. https://doi.org/10.1109/SBESC.2013.39

[14] Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. 2014. Trends in worldwide ICT electricity consumption from 2007 to 2012. *Computer Communications* 50 (Sept. 2014), 64–76. https://doi.org/10.1016/j.comcom.2014.02.008