Analyzing Programming Languages' Energy Consumption: An Empirical Study*

Extended Abstract†

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: Swifting from a traditional monolithic approach towards a more cutting-edge such as micro-services implies new challenges in terms of energy usage.

Goal: In the preliminary study, we aim is 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 available tasks from Rosetta Code, a publicly open repository for programming chrestomathy.

Results: Our analysis shows a the majority of compiled and semi-compiled programming languages achieve higher energy efficiency after applying the optimization options for *sorting, object generation, url-encoding* and *decoding* tasks. Moreover, we show which of the interpreted programming languages offers more energy gains than others.

CCS CONCEPTS

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

KEYWORDS

GreenIT, 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.

*Produces the permission block, and copyright information
†The full version of the author's guide is available as `acmart.pdf` document
‡Dr. Trovato insisted his name be first.
§The secretary disavows any knowledge of this author's actions.
¶The secretary disavows any knowledge of this author's actions.

PCI '17, September 2017, Larrisa, Greece

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

https://doi.org/10.475/123_4

1 INTRODUCTION

The increase demands on services and computational applications from ICT-related products are the major facts for contributing to increase energy consumption.[1] Recent research by Gelenbe and Caseau [5] and Van Heddeghem et al. [7] indicates a raising trend of the IT's sector energy requirements, which are expected to reach 15% of the world's total energy consumption by 2020. Moreover, obtained outline, also, raise of green house gas emissions due to the IT-sector which are growing much faster than initially predicted and are estimated around 2.3% globally, as claimed in SMARTer2030[2] report. Therefore, providing energy efficiency at different fields of IT is essential and of paramount importance, since the fact is not only economic but also environmental concern.

Traditionally, most of the studies, regarding energy efficiency, considered energy consumption at hardware level. However, there is much of evidence that software can also alter energy dissipation significantly [1, 3, 4]. Therefore, many conferences had identified the energy-efficiency at the software level as an emerging research challenge in order to reduce energy consumption of a software without comprising its run-time performance.

Nowadays, the development of a software has swifted from traditional monolithic architectures and follows a more agile, cutting-edge approach such as 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.

In order to identify trends and possible gains for reduce energy consumption in software development, we conducted an empirical study aiming on eliciting energy usage results starting from small tasks implemented in a variety of well-known and highly used programming languages. To this end, our aim in this research is to identify which programming languages offer more energy efficient implementations for different tasks.

The remainder of this paper is organized as follows. In Section 2, we discuss prior work done in the field and compare it with ours.

[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://smarter2030.gesi.org/downloads.php

Table 1: Programming Languages, Compilers and Interpreters

	Programming Languages	Compilers and Interpreters version
Compiled	C, C++	gcc version 6.3.1 20161221-(Red Hat 6.3.1-1) (GCC)
	Go	go version go1.7.5
	Rust	rustc version 1.18.0
Semi-Compiled	VB.NET	mono version 4.4.2.0 (vbnc)[a]
	C#	mono version 4.4.2.0 (mics)[b]
	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[c]

[a] http://www.mono-project.com/docs/about-mono/languages/visualbasic/
[b] https://www.codetuts.tech/compile-c-sharp-command-line/
[c] https://github.com/FedoraSwift/fedora-swift2/releases/tag/v0.0.2

Section 3 describes in details our experimental platform, the software and the hardware tools we used, how we refined our dataset, and our methodology for retrieve our results. In Section 4, we provide a discussion based on our preliminary results and Section 5 details threats to validity. Finally, we conclude in Section 6 and we discuss future work and possible directions for this research.

2 RELATED WORK

3 EXPERIMENT SETUP

In this Section, we describe the experimental approach to conduct our research and retrieve measurements. Initially, we provide information about the obtained dataset and the way we decided to select our tasks and refine it. Moreover, we argue on the selected tasks and programming languages we used and the way to refined our dataset. Furthermore, we explain the setup up of our experimental platform, the additional hardware tools, and the software tools used to conduct this research.

3.1 Dataset

In the context of this study, we used Rosetta Code,[3] 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 of (and cannot) tasks are implemented in all languages. We found and downloaded a Github repository[4] which contains all the currently implemented tasks introduced in Rosetta Code website.

[3] http://rosettacode.org/wiki/Rosetta_Code
[4] https://github.com/acmeism/RosettaCodeData

Table 2: Optimization Options for Compilers

Programming Languages	Optimization Option	Explanation
C	gcc -O3	
C++	g++ -O3	
Go	go run -gcflags -N	Active by default, otherwise use the -gcflags -N option to disable it.
Rust	rustc -O	
VB.NET	vbnc -optimize+,-	+ and - are used to add or remove optimization respectively.
C#	mcs -optimize+,-	
Java	java Xint	Used to execute byte-code in interpreted mode

For selecting the highly used programming languages, we made use of tiobe,[5] a software quality company. By making use of a formula,[6] 25 of the highest ranked search engines (according to Alexa),[7] and a number of requirements enlisted for programming languages, tiobe provides a search query for index rating of the most popular programming languages around the web for each month. Initially, we decided of choosing the top 15 programming languages as enlisted for June 2017. From the current list, we excluded programming languages such as Delphi and Assembly. In contrast, we included Rust in our dataset which is a memory safe programming language and is gaining vast popularity in the web. Therefore, we ended up with 14 programming languages as illustrated in Table 1.

In terms of selecting tasks, we developed a shell script (more details in Subsection 3.2.2) 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 choose only nine tasks implemented in the most of the programming languages of our selection. The selected dataset 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 refine further our dataset we used the following steps:

- Some of the tasks offered more that one implementation for the same programming languages. Thus, we had to drive manually through each directory and remove them until we have only one that is consistent with the other implementation. For example, when most of the implemented tasks used iterative implementation we removed the ones using recursion.
- The Java file's names were different from the public class names which results to compilation error if not changed accordingly.
- Some of the implementations didn't have main classes, nor the same data with other tasks. Therefore, we change the source code to offer consistency.

[5] https://www.tiobe.com/tiobe-index/
[6] https://www.tiobe.com/tiobe-index/programming-languages-definition/
[7] http://www.alexa.com/

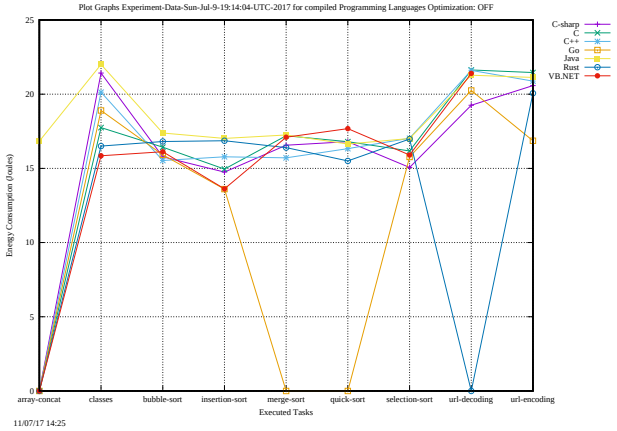

Figure 1: Average Results for Compiled Programming Languages Optimization: OFF

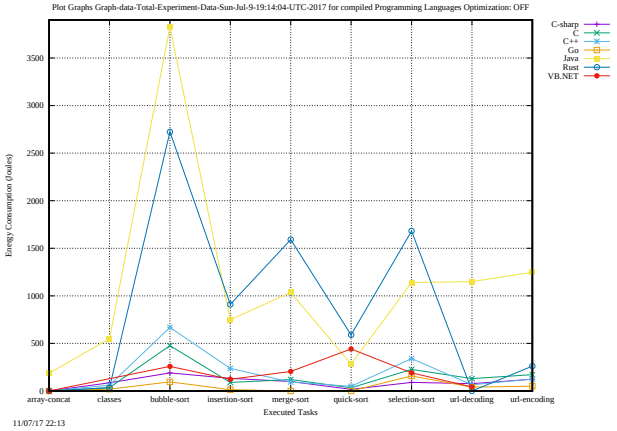

Figure 2: Total Results for Compiled Programming Languages Optimization: OFF

Table 3: System hardware and software specifications

	Description
Hardware	**HP EliteBook 840 G3**, Intel Core i7-6500U (2 physical cores of 2.5 GHz), 8 GB DDR4 memory, 256 GB SSD hard disk, **Raspberry Pi Model 3b**, 4x ARM A53 1.2GHz, 2 GB LPDDR2 memory, 64 GB SD
Operating System	Fedora 25 kernel version 4.11.5-200, Raspbian
Software	WUP software (retrieving measurements from the device), bash script, Gnuplot 5.0

- For some programming languages which do not offer the class option such as C and Go, we used structs.
- Some of the tasks are relatively small and may finish 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 modification on our dataset, we categorized our programming languages in three main categories as illustrated in Table 1. Moreover, for the programming languages which offer a compiled approach such as Java, VB.NET, and C#, we added them under the category of compiled languages. In addition, we compared the compiled and semi-compiled implementations while using the available compiler optimization and without them, as showed in Table 2.

3.2 Hardware and Software components

3.2.1 Hardware Components. The physical tools composed mainly from a portable personal computer, a real-time electricity usage monitoring tool, and an embedded device. Our systems' specifications are depicted in Table 3. The real-time power usage tools we used is the Watts Up Pro (WUP).[8]

In general, there are two venues for retrieving energy consumption from a computer-based system: on one hand, by indirect energy measurements through estimation models or performance counters, core component of software monitoring tools, and on the other hand, via direct measurement, hardware power analyzers and sensors. However, each approach has it own pitfalls such as: coarse-grained measurements for the whole systems' energy consumption and low sampling rate for direct measurements case and inaccuracy, lack of interoperability, and additional system overhead while using indirect measurements. Therefore, in our research we decided to retrieve our energy consumption measurements using direct approach such as WUP since our tasks are relatively short in terms of source-code lines.

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.[9] This software helped us retrieve measurements such as timestamps, watts, volts, amps, etc. 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 used a Raspberry Pi[10] to retrieve power consumption from our test-bed.

3.2.2 Software Components. To extract data, manage, and use our Rosetta Code Repository, we developed a number of shell scripts as enlisted below and are publicly available on our Git repository.[11]

[8] https://www.wattsupmeters.com/secure/products.php?pn=0
[9] https://github.com/pyrovski/watts-up
[10] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[11] https://github.com/stefanos1316/Rosetta-Code-Research

- **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 amount of programming language implementations.
- **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 directory tree's letters from upper to lower case to offer consistency for our scripts.
- **script.compileTasks**, compiles all tasks found under the Tasks' directory and produces error reports in a task fails to compile.
- **script.executeTasksRemotely**, executes all the tasks' implementations found in under Tasks directory. Moreover, it sends command to WUP, retrieves measurements and stores then 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[12] to synchronize both systems' clock 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,[13] 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. We tried to automate the execution procedure as much as possible in order to remove the burden from users who would like to use our scripts. Moreover, we suggest for the users not to change the directory's names or locations since it will alter the correct sequence of the execution.

3.3 Retrieving Energy Measurements

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

We estimated after an OS is launched, it's necessary to wait for a short period to reach a *stable condition* which is close to five minutes [2]. After reaching *stable condition*, we launched our main script *i.e.,* **script.executeTasksRemotely**, that executes all the tasks implemented in different programming languages. Before executing a tasks, the execution script sends a command to the remote host

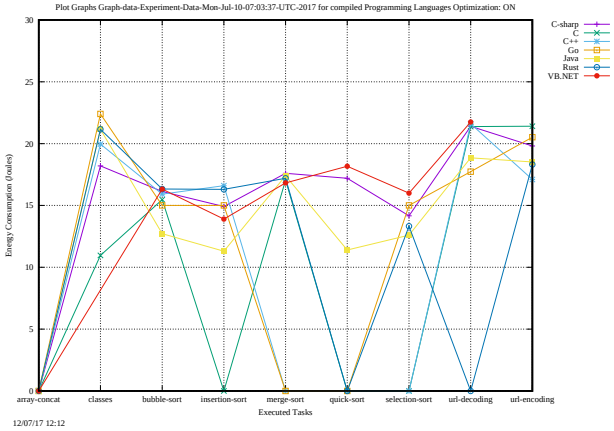

Figure 3: Average Results for Compiled Programming Languages Optimization: On

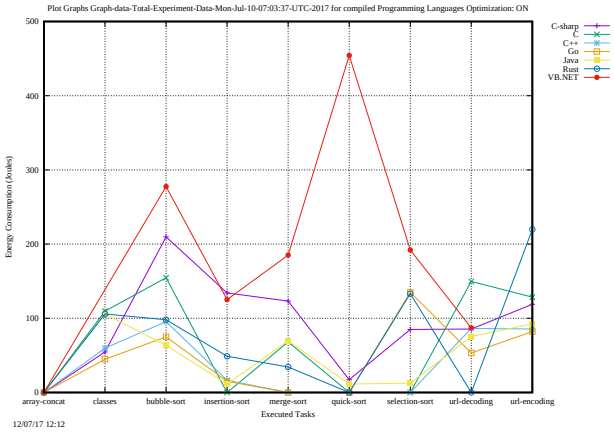

Figure 4: Total Results for Compiled Programming Languages Optimization: On

,*i.e.,* Raspberry Pi, through a password-less SSH connection to starting collecting power consumption measurements from WUP for the currently executing task. In addition, the local host retrieves run-time performance measurements through command time[14] and stores them in timestamped directories which we analyze later. Between each execution of a task, we added a sleep[15] period of three minutes. The time gap exist to ensure that our experimental platform reached a *stable condition*, 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.

[12] http://www.ntp.org/
[13] http://www.gnuplot.info/
[14] https://linux.die.net/man/1/time
[15] http://man7.org/linux/man-pages/man3/sleep.3.html

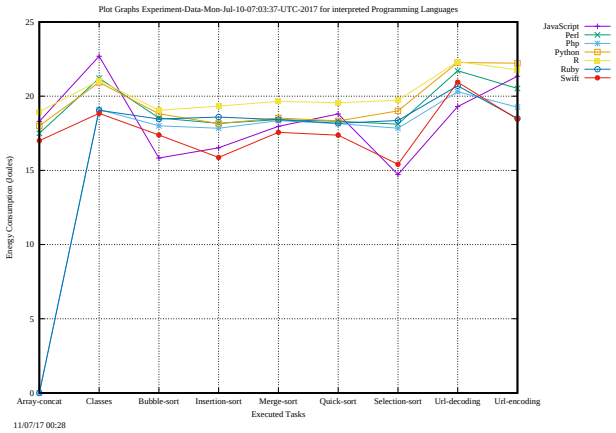

Figure 5: Average Results for Interpreted Programming Languages

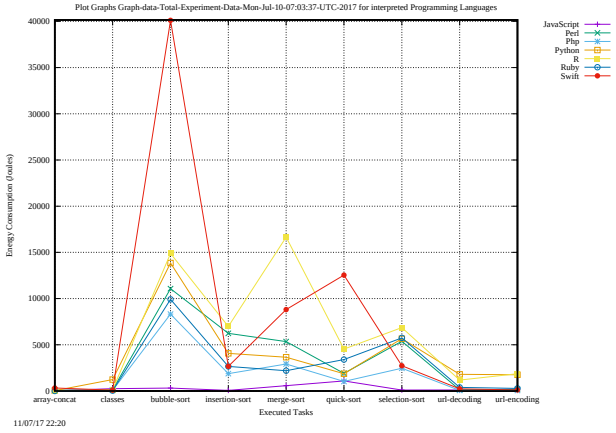

Figure 6: Total Results for Interpreted Programming Languages

4 RESULTS AND DISCUSSION

The collected results in Figure 1 and 2 illustrate the average energy consumption per second and the total, respectively. Both Figures depict that Java and Rust contributes to the highest energy consumption among the compiled programming languages without the optimization option. In the case of Java, we used the *-Xint* flag that runs the byte-code as interpreted and avoids the use of Just-in-Time[16] compiler.

Figure 3 and 4 show the average energy dissipation per second and the total energy consumed till the task is done, respectively, while making use of the compilers optimization. The results illustrate energy reduction, apart from Java, after applying the optimization option it exhibits energy savings. Moreover, in the case of Rust, the energy reduction after applying the compiler optimization option is significant In many cases, the results in Figure 3 tend to zero energy consumption. A fact that stems from the short tasks' execution time, *i.e.,* less than a second, which makes in impossible for the WUP analyzer to retrieve energy measurements at this time interval. Also, C and C++ are the programming languages that offer the most of energy gains in average after the *-O3* optimization flag is applied. Moreover, Java, VB.NET, and C# offers the most inefficient energy measurements, in average, for almost all tasks with the optimization option. A possible reason of the outcome is the use of virtual machines to run the byte-code, memory management, and the garbage collector.

For the interpreted programming languages, we can see the energy consumption scales similarly for all the tasks. On one hand, we see that Swift and Python results to the highest energy dissipation for all tasks except from *Classes*. On the other hand, JavaScript offers the lowest energy usage in total.

5 THREATS TO VALIDITY

Internal: In order to avoid additional overhead on our experimental platform, we used a remote host to collect our results. Therefore, the need of wireless connection was necessary, which might incur in 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 test our experiment.
External: Our real-time power analyzer offers minimum sampling interval of a second. Therefore, in Figures 1, 3, and 5 the energy dissipation resulting to zero are interpreted as the tasks execution to be less than a seconds, which makes in impossible for WUP to capture such measurements.

6 CONCLUSION AND FUTURE WORK

To this end we conclude as follows. The average energy consumption of compiled programming languages is much lower compared to the interpreted ones for the tested tasks. In addition, optimization option shows significant results are available for almost all the test cases. Our current results shows Java and Swift are the most inefficient programming languages among the compiled and interpreted, respectively.

In respect to compiled programming languages with the optimization option C and C++ contributes the lowest average energy usage while Go has the lowest total energy consumption for tasks such as sorting algorithms. For interpreted programming languages, JavaScript offers the highest level on energy-efficiency and Swift consumed the most energy in total.

As for future work, we would like to test all the 29 collected tasks and, furthermore, to developed more such as exception and task for functional programming. Moreover, we will test the collected tasks in different CPU architectures such as AMD and ARM. In addition, we plan to collect resource usage to identify possible relationship between programming languages and resources. To this end, we expected the obtained results to shed light and provide further

[16] https://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm

understanding on developing in an energy efficient manner for larger and more complex applications.

REFERENCES

[1] 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.

[2] 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

[3] 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

[4] 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.

[5] 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

[6] 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

[7] 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