ENERGY AND RUN-TIME PERFORMANCE PRACTICES IN SOFTWARE ENGINEERING

DISSERTATION FOR THE AWARD OF THE DOCTORAL DIPLOMA
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

2020

Stefanos Georgiou
Department of Management Science and Technology
Athens University of Economics and Business

Department of Management Science and Technology
Athens University of Economics and Business
Email: sgeorgiou@aueb.gr

Supervised by Professor **Diomidis Spinellis**

In loving memory of my grandma

Contents

List of Figures

List of Tables

Acknowledgments

A PhD is not as an easy task as I initially thought. A dear friend of mine used to say that it is also a lonely way. However, I was blessed to be surrounded by excellent people, both in terms of knowledge and character, who guided me through this road.

First of all, I would like to thank a really special and smart person from whom I have learned a lot, Professor Diomidis Spinellis. His devotion and principles of doing the right thing are fascinating, and these are qualifications that I admire the most. I never felt that there is a single question that he could not answer or guide me towards the answer. From every meeting and discussion with him, I felt that I learned something new or even adopted a different way of thinking how to solve problems. We have spent a lot of time talking about new, cutting-edge technologies and their applications. Moreover, we discussed so many new research ideas that I cannot wait to deliver this PhD thesis and start working on them. Also, I will never forget his happy expression every time I had a research paper accepted. For that, I can only say: Professor, it was an honour working with you, thank you for everything that you did for me, I will never forget that!

Although he is not a statistician, or a beer brewer, or whiskey distiller, it is a strange fact that he knows so many about these things. I guess that Panos Louridas is a person who holds great knowledge about many things. Sometimes, I feel he is a kind of superman because he can balance both academia and industry and be super productive in both of them. Apart from this, he is a person that makes you feel comfortable in conversations and fun to talk with. I am pretty sure that everything I have learned from him, regarding statistics, will be used in my future academic career. Moreover, information regarding beers and whiskeys will not be discarded for any reason!

I thank Professor Rizos Sakellariou of his guidance and comments regarding my research work and for promoting them in his lab. Every time he visited Athens, he always stopped by our lab to talk to me and see my current PhD progress. It is always a pleasure to talk to him about new research ideas with respect to the development of energy-efficient software. However, apart from research he was always interested to talk about politics and simple daily things.

SENSE was the initial name of the lab when I started my PhD. A lab with excellent people who helped me do my first baby steps in terms of research and academic English writing (a point that I still work on very hard). People like Antonis, Tushar, Maria, and Marios were always there for me, physically or virtually (through Skype) and they did their best to support me. Apart from working, we had Star Wars nights. These were geeky nights that took place on Fridays each time at a different person's home, accompanied by junk food or Indian super spicy food. Also, we had an internal agreement to write in our daily PhD progress report, that we had a Star Wars night. After the SENSE lab, our team was renamed to BALab. Although the name was changed, the way we did things such as drinking and eating nights did not change much. Admittedly, we started working more on our research because it was about time to finish and move on!

Among the lab's personnel, I would especially like to thank my partners in crime and dear friends Antonis and Tushar with whom I started my PhD journey. The team of the "Indian" (Tushar), the "Cypriot" (Me), and the "Other" (Antonis) as the students usually used to refer to us for fun. Antonis, my koumparos, is a real mentor to me in terms of life skills and correctness. He is the guy who I will ask for political correctness advice. Moreover, from him, I adopted many habits such as beer and rum drinking, growing beard, listening to rock music, enjoying the sunset with music and drinks, and mountain hiking. He had such a strong impact on me, but sadly I could not affect him with any of my strong habits such as going to the gym or running. Regarding Tushar, he is like my oldest brother with whom I spent long time discussing research and good places for food around Athens. Tushar is a man with great sarcastic sense of humour that I like much, the team's photographer, the wise-man, and the guy who eats his watermelon with salt and pepper (quite shocking, I know). Also, he is the one who introduced me to real whiskeys, the single malts. In conclusion, I have gained so much from these two persons and I am more than thankful to have them close to me.

Maria and Marios, were the first to show us around the University and help us with the PhD enrollment process. Maria was my Programming II course mentor and also the person who helped me submit my first prestigious conference paper. Although she is not in the lab anymore, we discussed many research ideas over Skype and keep meeting every time she is visiting the city. She is a reliable and gentle person that never misunderstands or likes to have a conflict with others. Whenever I like to have a discussion on tools, utilities, or BASH scripting I turn to Marios. Marios is a very kind, easy-going, trustworthy, and hardworking person. It is always a pleasure to meet and talk to him about different subjects.

Vasiliki, Konstandinos, and Thodoris were among the latest members of our lab. Vasiliki, our eldest sister, as Antonis usually calls her, is the labs world-traveler and the most outgoing person. I guess that is the reason she knows so many places to crash for a beer anytime we go out. Also, she is a funny person and has great organizing skills. I would especially like to thank her for proof-reading all my research studies. Her comments and help were valuable and the reason that I am able to present this thesis sooner. Thank you koumpara for everything! Konstandinos, on the other hand, is that person who knows great places for food. Although I had many arguments with him on some restaurants, I can say his choices are pretty good. I was really glad that I had the opportunity to give tutorial lectures with him for the course of Programming II. Although he is the guy who likes to tease others a lot, he is very kind and always with good intentions and suggestions. Kostnadine, thank you for storing all my belongings in your apartment when I moved out! Thodoris is among the smartest persons that I know. He has strong principles and likes to do the right things both in work and life. Although he looks like a quiet person, he will strongly speak up against unjust behaviors. Moreover, I am amazed by his hard-working motivation in research and gym exercising. I have the feeling that I am going to meet him regularly in conferences.

My dear Daniela, thank you for being there and making my stay here in Greece enjoyable. With her, I have learned so much about analysing people and situations. She is a fast-thinking person and I fully trust her opinions and judgment. Never forget to be this joyful, silly, and happy girl. Vangeli, you are a crazy, honest, and reliable guy with good intentions. Thank you for taking good care of me during my stay in Athens. My friend Ilias, the captain who is always well dressed, even on mountain hiking or rafting. He is a funny guy with good intentions, advice, and judgment.

Other Lab members that I hold dear and I appreciate their support and collaboration are Dimitris, Constantina, Zoi, Stefanos, Vitalis, and Vangelis. The team also know as "the first shift" since we used to go together for lunch at 13:00. Thank you for feeling this crucial need of mine and supporting the early lunch time group idea. Special thanks to Zoi for giving me great research paper reviews.

Matina is a gentle, kind, inspiring person, and the superwoman in writing research proposals. She has always supported me and Antonis when we were at Singular Logic. Moreover, she guided me and Antonis through our first steps in research and taught us how to think in an industrial way of researching. Mr. Stelios, is a person that despite the heavy workload he is always cheerful, fun to talk with, kind, and cool. A really good example of a manager. Moreover, I would like to thank Marios, Nikos, Kostas, Panagiotis, Gianna, Evelina, and many others from Singular Logic's personnel for their support and advises. It was a pleasure to work with you all! I also thank the Singular Logic S.A. for hosting me during my PhD studies and for buying all the necessary equipment to perform my experiments.

I would like to thank AllCanCode start-up from where I have obtained Backend development and DevOps skills. By working there, I adopted so many new research and blog ideas that I would like to implement soon. I thank Kostas who taught me with patience, smile, and accepted me as a part time developer while doing my PhD. Thank you Niki and Konstantina for all the good times, discussions, suggestions, moments you have offered me, and thanks again for being my friends. Your company made my day, while your absence was just another day at the office. I also thank Maria, Spyros, and Spiros for your time, the joyful moments at work, and all the fun discussions.

My second academic host, Delft University of Technology, hosted me for my secondment and people there gave me multiple opportunities for collaboration. It was a really pleasant working environment and the country's mentality favors many people to chill out and focus on their work. I would like to say thank you, to all the personnel from TU Delft for offering me this opportunity to stay and learn from you.

I would like also to thank all the people from the Athens University of Economics and Business for helping me out whenever it was needed. Moreover, thank you for giving me the chance to work with you, learn from you, and become better.

I am grateful of the European Union and the Marie-Curie association supporting researchers to pursue doctoral studies. You are giving many people incomparable opportunities to fulfill their dreams. Please, keep up the good work.

Special thanks to my father and mother who always supported my decision on doing a PhD and spent time to hear my endless stories and complains. I feel also very lucky for having my sisters' support during the years of my studies.

Last, I would like to thank my dear and sweet girlfriend Anett who supported me through this journey and helped me in any way she could to make my studies easier. Your help and support were invaluable during the Covid-19 crisis, where you took a one-way flight to Athens and stayed with me until I wrote up this thesis. Thank you for standing next to me on every decision that I made, tolerating me when I was not in a good mood, listening with excitement every crazy research idea that I had, and for making so many nice meals while I was studying. You are a very caring person and you motivate me to become better. I do not believe that I could have a better life partner.

– Stefanos Georgiou

Summary

Energy efficiency for computer systems is an ever-growing matter that has caught the attention of the software engineering community. Although hardware design and utilisation are undoubtedly key factors for affecting energy consumption, there is substantial evidence that software can also significantly influence the energy usage of computer platforms. To this end, researchers in software engineering have carried out numerous studies at different granularities of the software stack.

By investigating literature related to energy efficiency in software development, we performed a survey study and found several research challenges. First, the phase of software requirements lacks the knowledge for building energy-efficient applications. Second, the appropriate selection of configurations and parameters such as the number of threads, data size, and data locality to achieve the best energy and run-time performance for a program is done manually, hence proves to be a slow and cumbersome procedure. Third, appropriate data-structure selection can significantly alter the energy consumption of applications. However, the lack of refactoring tools to guide developers in data-structure selection to reduce an application's energy demands makes the selection process challenging. Forth, there is a need for tooling support to allow developers to easily fetch their applications' energy consumption, in a fine-grain approach, in order to understand and optimise them. Current tools require a number of configurations which most of the time prove to be cumbersome to use. To this end, accurate and versatile software-based energy monitoring tools are of paramount importance. This will promote a wider take-up by software researchers and developers. Lastly, some of the new research directions that we have identified in our survey study and we investigate in this thesis are the energy and run-time performance implications of: (1) tasks implemented in different programming languages and run on distinct computer platforms, (2) various remote Inter-Process Communication technologies that are implemented in different programming languages, and (3) different security mechanisms that protect modern operating systems from malicious users.

Software practitioners can select from a large pool of programming languages to develop software applications and systems. Each of these programming languages comes with several features and characteristics that can affect the energy consumption and run-time performance of programming tasks implemented in them. To this end, we perform a large scale empirical study to measure the energy consumption and run-time performance of commonly used programming tasks implemented in different programming languages and computer systems. We 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 them on three different computer platforms, a server, a laptop, and an embedded system.

Our results show that the compiled programming languages outperform 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, respectively, for the Energy Delay Product, Rust appears to be well-placed for I/O-intensive operations, such as file handling. We also find that a good behavior, energy-wise, can be the result of clever optimisations and design choices in seemingly unexpected programming languages.

We further extend the above study by analysing another important aspect of modern IT services, the remote Inter-Process Communication technologies that enable computer systems to use the Internet for creating, reading, updating, and deleting shared data. Compared to the above study, in this work, we use a client/server approach to study the energy and run-time performance. To evaluate the energy consumption of Inter-Process Communications technologies and the corresponding run-time performance implications, we examine technologies such as RPC, Rest, and gRPC implemented in Go, Java, JavaScript, Python, PHP, Ruby, and C#. We perform our experiments on computer platforms equipped with Intel and ARM processors. The results suggest that JavaScript and Go implementations of gRPC yield the lowest energy consumption and execution time. Furthermore, by analysing their system call traces, we find that inefficient use of system calls can contribute to increased energy consumption and poor execution time.

An operating system (OS) typically includes numerous security mechanisms that protect the confidentiality, integrity, and availability of its data and services. However, such safeguards may impact energy consumption and encumber the run-time performance of applications running on a system. We present a large scale study, where we investigate how the various security mechanisms affect energy and run-time performance cost at the OS-level. We focus on well-known mechanisms including encrypted communication protocols, memory zeroing, GCC safeguards, and CPU vulnerability patches against critical vulnerabilities, such as Meltdown. To do so, we utilise 128 benchmarks of different application types found under the well-established Phoronix test suite. Our findings suggest that security mechanisms lead to an increased energy consumption and significantly degrade the run-time performance of various applications including web servers, databases systems, kernel operations, and disk usage. Notably, when we disabled the various security mechanisms, real-world applications such as Apache and Redis, indicated important energy (from 18% to 41%) and run-time performance (23% to 45%) gains. Additionally, we examined the correlation between energy consumption and performance. Our findings showed that the two are not always related. Overall, our results suggest that administrators should consider disabling such security mechanisms when a computer system runs inside a secure environment to benefit from energy and run-time performance gains.

In conclusion, this thesis presents research studies in three different directions First, it offers a large-scale study and points out which languages are more energy-friendly than others for particular programming tasks and computer systems. Second, it presents the energy and run-time performance impact of various Inter-Process Communication technologies and how they can affect simple daily tasks. Finally, this thesis, goes beyond the selection of various languages and frameworks and depicts the daily energy and run-time sacrificed to keep the integrity of our data against malicious users. Along with the thesis, we offer two data-sets with tools to install system and library dependencies, to manage and execute them, and to visualise the results. Software practitioners can utilise these data-sets to perform related studies of interest.

Chapter 1

Introduction

In this chapter, we initially provide the context of this work, the problem statement, the proposed solutions, the contributions, and the thesis outline. Regarding the section on how to read this thesis, we inform a reader on how our research studies are divided across the sections of this thesis and in which tense each section is written in. In the context section, we highlight the importance of energy efficiency for computer systems, while in the problem statement we define the limitations of previous studies. We propose on how to fill the research gaps defined in the *Problem Statement* section, and we discuss the contributions of this thesis. After, we present the outline of this work. Finally, we present a guideline on how to read this thesis.

1.1 Context

The energy consumption, for the IT-related products, is an ever-growing matter that has caught the attention of academic researchers and industry. This is primarily due to the increasing costs, as IT-related energy consumption is estimated to reach 15% of the world's total by 2020 [170], while it is expected to reach 50% by 2030, in the worst-case scenario [11]. Energy consumption of IT systems is particularly important in two areas. First, the data centres, one of the vital contributors to the IT sector's global energy consumption, are housing a large number of server nodes that are running programs and communicating with clients through energy-intensive remote IPC technologies. Second, the blossoming field of IoT, where low energy performance is critical, has multiple embedded devices connected with cyber-physical systems to manage, analyse, exchange, share, and transmit data. To this end, providing sustainable solutions, by reducing energy consumption, to ensure data centres' and IoT infrastructures environmental sustainability and business growth is of paramount importance.

Energy efficiency for computer systems can be examined in terms of software and hardware. In this thesis, we investigate the energy implications of the software-related category. Developing software to be energy-efficient is a demanding and challenging task because of the lack of tooling support, practices, and guidelines [136, 117, 94]. Researchers have carried out studies on different aspects and granularity of software artifacts to determine the energy consumption of data structures [138, 64, 126, 129, 174], different programming languages [5, 145, 88], multi-threaded applications [137, 134, 133], coding practices [168, 90, 155, 154, 140], and so on. Although there is a large and consistent body of knowledge in various fields of software engineering, there is still a lack of guidelines to identify energy-wise decision making on programming language and framework selection.

Apart from the applications, that are running in a computer system, there is also the Operating System (OS) that is responsible for managing these applications and their resource usage. Modern OSs are equipped with various services, background processes, daemons, and so on, each of which may increase the energy consumption and execution time of computer platforms. Among such services are the security mechanisms that are responsible for guarding a computer system against malicious users. Even though many studies examine the run-time performance of various security measures, their energy consumption has not been investigated thoroughly.

1.2 Problem Statement

This thesis aims to further enrich the existing body of knowledge regarding the development of energy-efficient software by (1) identifying which programming languages are more energy-efficient for particular tasks and computer systems, (2) presenting which remote IPC technology implementations can reduce web applications energy demands, and (3) pointing out the energy performance penalty of applications caused by security measures.

Programming languages are interfaces for communicating with computer systems in order to manage their resources and solve algorithmic problems. Although a large pool of programming languages exists, there is limited knowledge or understanding of their energy consumption. In many cases, researchers compared only a handful of programming languages on a limited amount of tasks or computer platforms [5, 34, 144]. In order to draw precise and reliable conclusions, it is important to conduct large scale studies on various programming languages, tasks, and computer systems to point out the most energy-efficient programming languages for developing specific tasks on selected computer platform types (embedded, laptop, or server systems).

IPC technologies are the hubs between client and server systems to manage remote shared data. To this end, many IPC technologies exist that are developed in different programming languages in order to create, read, update, or delete remote data on the Internet. Prior work [67, 33, 24, 100, 111] focused on investigating the energy consumption and run-time performance of smart phones and embedded systems on Java implementations for remote IPC such as RPC, REST, SOAP, and WebSockets. Considering that IPC technologies are the major components of web services and used for every interaction with the users' data, it is essential to identify their energy and run-time performance while being developed in different programming languages or executed in different computer platforms.

Unix-like OS's are equipped with various services, background processes, and daemons [165]. Among such services are the security mechanisms that protect computer systems from attackers trying to exploit potential vulnerabilities. Such security mechanisms can affect the energy consumption and the run-time performance of a computer system. Prior studies have highlighted that CPU vulnerability patches can add a significant computational overhead to an application (from 2% to 21%) [160, 142]. Likewise, researchers have found that GCC safeguards such as `stack canary` [41] and Position Independent Executable (PIE) [89] can tax an application's run-time performance from 6.5% to 18% [42, 179, 164, 125]. In terms of energy consumption, a number of studies have assessed the energy implications of encryption algorithms [139, 131]. The authors have shown that cipher algorithms can impact energy consumption from 60% to 150%. However, none of the above studies have investigated the energy implications of CPU vulnerability patches, encrypted network communications, the kernel's memory zeroing, and GCC safeguards. Moreover, the aforementioned studies utilised a small number of test cases to examine the run-time performance implications of security mechanisms. Finally, no one has examined how energy consumption is correlated with the run-time performance of different tasks and security mechanisms.

To this end, we define the research goals of this thesis.

- To understand the energy-delay implications that various programming languages have on different programming tasks and computer platforms.
- To explore the energy-delay implications of various IPC technologies that are implemented in different programming languages and running on distinct computer platforms.
- To identify the energy and run-time performance taxing that security measures (that shield modern OS) have on different types of computer programs.

1.3 Proposed Solutions and Contributions

This dissertation aims to cover the challenges defined in the above section. To answer the first challenge, we perform a large-scale empirical study where we examine the energy-delay product (EDP), a weighted function that takes into account a task's energy consumption and run-time performance. We conduct the study on 25 programming tasks implemented in 14 different programming languages. Moreover, we evaluate the performance of the 14 programming languages on three computer platforms (*i.e.*, embedded system, laptop, and server). Also, we analyse our results and, further on, we try to reason on the outcome regarding the most energy-efficient and inefficient implementations.

To fulfill the second goal, we examine the energy and run-time performance of the RPC, gRPC, and REST IPC technologies implemented in Java, JavaScript, C#, Go, Ruby, PHP, and Python. Moreover, we perform our experiments on computer systems equipped with Intel and ARM processors in order to point out which of the IPC implementations are the most energy and run-time performance-efficient. We also investigate what are the reasons behind the energy-efficient and inefficient implementations of IPC technologies by examining their system calls and resource usage.

Last, we perform an empirical study to examine the energy and run-time performance implications that security measures have on OS. Specifically, we investigate security mechanisms related to (1) CPU vulnerability patches (*i.e.*, Meltdown, Spectre, and MDS), (2) memory (*i.e.*, memory zeroing), (3) communication (*i.e.*, HTTP and HTTPS), and (4) compiler (*i.e.*, GCC safeguards). Although many of the above have been investigated in terms of run-time performance, there is a limited amount of work that appraises their energy consumption [160, 142, 42, 179, 164, 125, 139]. We perform our experiments on a rich data-set of 128 multi-purpose benchmarks running more than 300 tasks. Nevertheless, we try to figure out the reasons that make certain vulnerabilities more energy-demanding, while we also trying to identify which application types are more performance-sensitive to the associated security mechanisms.

In summary, this thesis provides the following contributions:

- A customised and extended data-set [1] that can be used as a benchmark to investigate the performance of various programming languages.
- A data-set [2] of 128 multi-purpose benchmarks that can be used to evaluate computer systems performance.
- A set of publicly available tools [3] for measuring the EDP of various programming tasks implemented in different programming languages.
- An empirical study [4] on programming language EDP implications, by using different types of programming tasks and software platforms (see Section 4.1).
- A programming language-based ranking catalogue,[5] in the form of heatmaps, where developers can find which programming language to pick for particular tasks and platforms depending on whether energy or run-time performance are important.
- An investigation on whether energy consumption and run-time performance of computer systems are proportional (see Sections 4.2.3 and 4.3.3).
- An empirical study [6] identifying the energy-delay implications of various IPC technology implementations.
- A study exhibiting the energy-delay impact that various security measures have on different applications (see Section 4.3).

1.4 Thesis Outline

This thesis is separated in six chapters, outlined below.

Chapter 2 provides a taxonomy of studies associated with software engineering for energy efficiency. Specifically, it presents tools and techniques that enable software practitioners to save energy through software engineering practices. Additionally, it illustrates the state-of-the-art and presents the features and limitations of the existing body of knowledge.

[1] https://github.com/stefanos1316/Rosetta_Code_Data_Set
[2] https://github.com/stefanos1316/phoronixDataSet.git
[3] https://github.com/stefanos1316/Rosetta_Code_Research_MSR
[4] https://stefanos1316.github.io/my_curriculum_vitae/GKLS18.pdf
[5] https://github.com/stefanos1316/Rosetta_Code_Research_MSR/tree/master/heatmaps
[6] https://stefanos1316.github.io/my_curriculum_vitae/GS19.pdf

Chpater 3 describes the methods used in each study of this thesis such as (1) research questions, (2) subject systems, (3) research approach, and (4) threats to validity.

Chapter 4 depicts the collated results and analysis of our experiments in the form of graphs and tables where we also present our reasoning on the outcomes.

Chapter 5 presents the thesis conclusions, contributions, and discusses potential future work and research directions.

1.5 How to Read this Thesis

This thesis comprises four research studies on the development of energy-efficient software. We advise to read the following instructions.

- The first chapter, Introduction, is written in present tense and presents an overall view of the research challenges set for this thesis and how we address them. Therefore, we advise on reading this chapter first because it shows the aims of this thesis.

- The Related Work chapter is also written in present tense and embodies studies from various granularities of software engineering. Moreover, it includes related work from the research studies that we performed and present in this thesis. Note that this is our first research study and it can be read as a stand-alone work.

- The remaining chapters present three more research studies. We advise to read first the sections associated with the programming languages energy and delay implications from each chapter. Specifically, readers may start by reading the Methods (written in past tense), then the Results and Analysis (written in present tense), and, finally, the Conclusions and Future Work (written in a mix of past and present tense). Afterwards, the readers can repeat the same for the sections corresponding to the remote Inter-Process Communication (IPC) technologies and, finally, the security mechanisms study.

Meanings of the highlighted boxes
In this thesis, the boxes below denote:

Goals, where we elaborate on the aims of the specific research study.

Contributions, where we discuss the significance and usefulness of our findings.

Key findings, where we present important results that emerged from our studies.

Opportunities, where we illustrate opportunities for further research.

Chapter 2

Related Work

In this chapter, we present a literature review that covers various phases of the Software Development Life Cycle (SDLC) with respect to energy efficiency. Section 2.1 introduces the importance of this field and discusses the methods used in order to obtain the data-set of the examined studies. Sections 2.2, 2.3, 2.4, 2.5, and 2.6 outline the different categories that taxonomised our work. Finally, we discuss research opportunities in Section 2.7.

2.1 Background

A proposed method to counter IT's growing energy hunger is through *GreenIT*: the practice of designing, implementing, using, and disposing of IT-related products in an eco-friendly way [105]. The concept of GreenIT has seen widespread adoption and acceptance from research communities and organisations [66].

Initially, GreenIT was mostly adopted and considered at the hardware level. For example, Bacha and Teodorescu [12, 13], Papadimitriou et al. [121] proposed firmware to reduce the voltage margin and supply voltage without degrading the operating frequency of the CPU to save energy. Leng et al. [82] showed the energy benefits obtained by reducing the GPU's voltage margin. Through a survey study, Mittal and Vetter [99] illustrated the energy optimisation opportunities that non-volatile memories can offer.

Although hardware design and utilisation are undoubtedly key factors affecting energy consumption, there is solid evidence that software design can also significantly alter the energy consumption of IT products [30, 53, 50]. To this end, dedicated conference tracks (*e.g.*, GREENS,[1] eEnergy[2]) have identified energy efficiency as an emerging research area for reducing software energy consumption through software development practices. Existing research studies in the area have tried to address some of the challenges for reducing energy consumption in software development by defining appropriate metrics, utilising energy measuring tools, and proposing best practices. For example, Lago et al. [81] presented several energy consumption metrics and classified them under various environments and purposes. In the context of energy monitoring tools, Noureddine et al. [110] performed a study to point out the current state-of-the-art by contextualising existing approaches regarding energy measuring tools for servers, personal computers, and smart-phones. An initial study by

[1] `http://greens.cs.vu.nl/`

[2] `http://conferences.sigcomm.org/eenergy/2017/cfp.php`

Procaccianti et al. [140] shows 34 best practices[3] that can improve the energy efficiency of computer programs.

Overall, current research provides a fragmented view of the energy-efficient techniques associated with the SDLC and examines only particular phases of it. This chapter intends to fill this gap by presenting works in the development of energy-efficient software under the holistic scheme of SDLC. For the presented studies, we focus on eliciting implications at each phase of the SDLC, not only concerning energy consumption but also run-time performance, where it is relevant. The goal is to guide researchers and software practitioners on existing methods that can be beneficial and practical at each phase of software development. Additionally, we aim to raise awareness regarding current difficulties and limitations.

This chapter contributes to the field of energy efficiency for software development as follows.

- It provides an overall view, analysis, and taxonomy of existing technologies, tools, and techniques for each phase of the SDLC, *i.e.*, *Requirements* (section 2.2), *Design* (section 2.3), *Implementation* (section 2.4), *Verification* (section 2.5), and *Maintenance* (section 2.6), for energy efficiency.
- It identifies the state-of-the-art on energy-efficient design and development, presents a critical review on different parameters which may affect energy efficiency at each phase of the SDLC, and discusses limitations and future challenges (section **??**).

In the rest of this section, we explain our method for compiling related studies (subsection 2.1.1) and describe our approach for classifying them (subsection 2.1.2).

2.1.1 Methodology

This study aims to investigate dimensions that affect energy consumption at different phases of the SDLC. In order to retrieve related studies, we composed search queries from the keywords “energy” and “power”, combined with relevant words, *i.e.*, “software development life cycle”, “software requirements”, “design patterns”, “parallel programming/computing”, “approximate programming/computing”, “coding practices”, “data structures”, “programming languages”, “code analysis”, “benchmarks”, “monitoring tools”, “evaluation tools”, “maintenance”, and “refactoring”.

We searched for relevant publications by querying the following digital libraries, journals, and magazines: IEEExplore, ACM, ACM Computing Surveys, Springer, ScienceDirect, IEEE Software Magazine. Initially, we had restricted our search spectrum only to the main tracks of the top software engineering conferences as proposed by Pinto et al. [132]. However, the field of energy efficiency in software development is relatively new, hence related studies published in these venues are still sparse. Therefore, we extended our search to other energy- and software-related conferences and workshops to enrich our data-set. Additionally, when a retrieved paper was on a topic close to our interests, we used the *backward*

[3] wiki.cs.vu.nl/green_software/

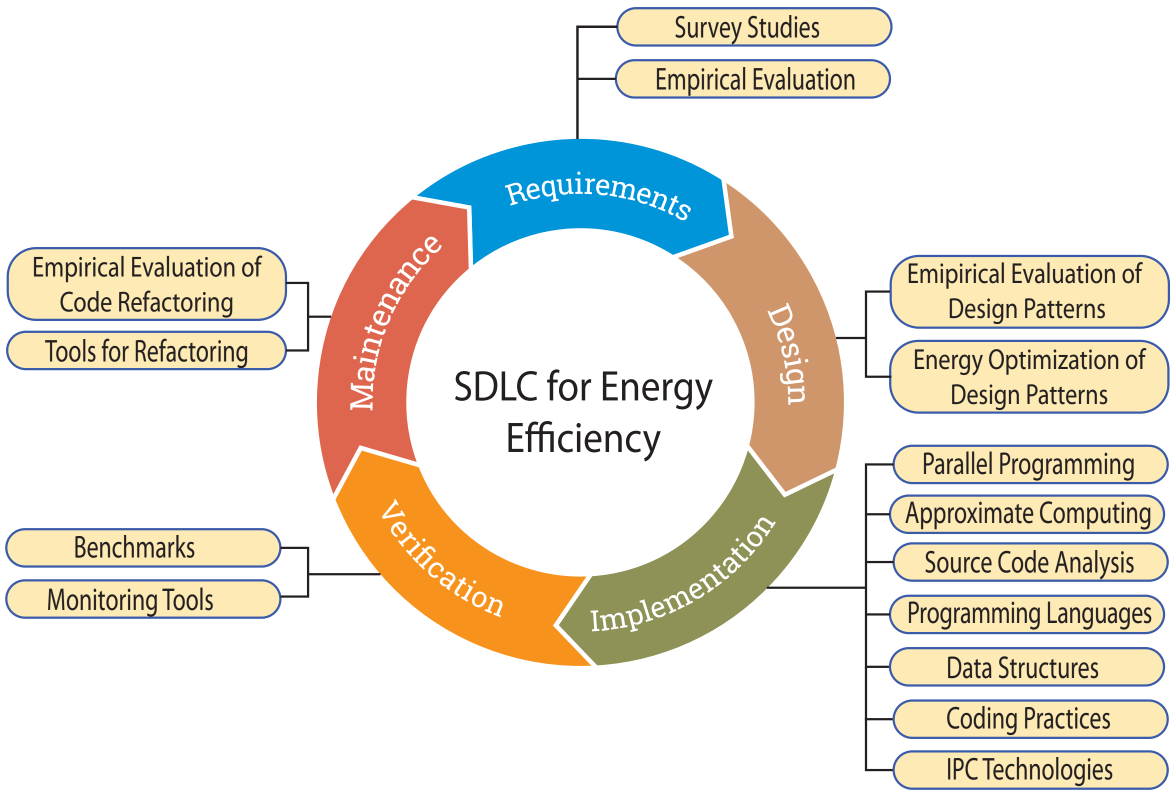


Figure 2.1: Software Development Life Cycle for Energy Efficiency Taxonomy

reference searching process (examining the references of the corresponding paper) to track down supplementary relevant publications .

2.1.2 Energy Efficiency in the Context of SDLC

In this chapter, we limit our scope of interest in existing work related to the energy efficiency focused at each phase of SDLC, *i.e.*, *Requirements*, *Design*, *Implementation*, *Verification*, and *Maintenance*, following the waterfall model [151]. Although the SDLC waterfall model is an outdated software development approach, it is still useful as a reference model for categorising the area's research. In Figure 1, we present the mapping of the energy efficiency techniques and tools under the SDLC process. Following this classification, we structure this chapter accordingly to provide a complete view of existing tools and techniques for energy-aware software development.

The *Requirements* phase (section 2.2) describes aspects and needs relevant to the development of software projects/systems when the energy efficiency is compulsory. During the *Design* phase (section 2.3), software design patterns document best practices used for solving common ground problems or poor design decisions taken during the development of an application that can impact the energy consumption. *Implementation* phase (section 2.4) clusters methods/practices, namely *Parallel Programming*, *Approximate Computing*, *Source Code Analysis*, *Programming Languages*, *Data Structures*, *remote IPC technologies*, and *Coding Practices*, which practitioners can adopt to reduce the software's energy consumption when developing. The *Verification* phase (section 2.5) considers metrics and tooling support aiming to evaluate the software's energy consumption before its deployment. *Maintenance* (section 2.6) is the phase that aims to apply refactoring patterns/techniques on a deployed application to reduce its energy consumption.

2.2 Requirements

The classification of energy efficiency for the SDLC falls under the non-functional requirements such as run-time performance and security. We present relevant work on requirements, concentrating on suggestions gathered by surveying practitioners and on results based on empirical evaluation. Accordingly, we divide the *Requirements* section into two subsections: survey studies and empirical evaluation requirements.

Table 2.1: Collected Requirements and Identified Limitations

Platform	Study Type	Identified Limitations	Source
Smart-phones	Survey	In programmers education	Pang et al. [117]
Workstations and Smart-phones	Empirical Evaluation	On sustainability model for software development	Beghoura et al. [18]
Smart-phones	Survey	On guidelines, support infrastructure, and reasonable cost of a given task	Manotas et al. [94]

2.2.1 Survey Studies

Mobile devices are more energy-dependent than servers or workstations; therefore, it is crucial to adopt an energy-conscious approach and consider the battery life, as a limitation, before developing a mobile application. A qualitative study conducted by Manotas et al. [94] examined green, energy-efficient software engineering perspectives of 464 practitioners from ABB, Google, IBM, and Microsoft software development departments. Although the study concerned different computing systems, specifically, i) mobile, ii) embedded systems, and iii) data centers, the practitioners expressed ideas, mostly, on how to reduce energy consumption on mobile applications. Some of the practitioners also provided examples regarding the sustainability of smart-phones energy consumption, such as: *"turn-by-turn guided navigation should not drain more battery than the car can charge"*, *"under normal usage, a device with an X W h battery should last for Y hours"* [94]. Additionally, practitioners suggest that particular tasks be executed without disturbing the users about battery drain.

Likewise, Pang et al. [117] performed a study surveying on-line 122 programmers and concluded in line with Manotas et al. that software practitioners mostly consider energy consumption as a requirement for mobile application development. To this end, the authors argued that developers could extract energy-efficiency requirements by correlating application functional requirements with the corresponding software components' energy consumption. To accomplish this, a consistent body of knowledge and understanding of software and hardware interaction is necessary. The authors listed the following relevant instructions that developers can take into account:

- Bulk Operations, in order to keep I/O calls minimum.
- Hardware Coordination, such as minimising memory access.
- Concurrent Programming, such as selecting appropriate thread construct.
- Efficient Data Structures, selecting less energy-greedy data structures.
- Loop Transformation, such as loop fusion to reduce control operations.
- Data Compression, to reduce file sizes before transmitting them.
- Offloading Methods, by calculating heavy operations remotely, *i.e.*, cloud.
- Approximate Programming, to reduce unnecessary precision of computations.

2.2.2 Empirical Evaluation Studies

An approach to identifying non-functional requirements in order to reduce energy consumption in software development has been proposed by Beghoura et al. [18]. The authors focused their study on desktops and smart-phones, where they argued that, to meet energy-efficient software development, it is necessary to identify the characteristics and requirements of the software. Along these lines, a practitioner may consider four characteristics for providing energy-efficient requirements for different types of systems, namely:

- Computations, to optimise energy consumption through less expensive computations (CPU-bound).
- Data Management, to reduce the amount of I/O operations, because they are slow and expensive for a computer system (storage-bound).
- Data Communication, to mind the amount of data sent or received through a network channel (network-bound operations).
- Energy Consumption Awareness, to provide energy-related information for individual software layers (*e.g.*, through energy profiling tools) and for the software as a whole.

To this end, Beghoura et al. developed a tool for testing application requirements, by estimating the energy consumption of CPU, RAM, and NIC. In contrast to Pang et al. and Manotas et al., Beghoura et al. empirically evaluate their proposed requirements. The work of Pang et al. and Manotas et al. are survey-oriented and do not provide an assessment of the proposed suggestions. However, Manotas et al. obtained their requirements from experienced and well-known software development companies. Overall, there is a lack of research work concerning practical guidelines for the *Requirements* phase regarding the energy efficiency of software development. Most of the works are limited to developers' experience. As illustrated in Table 2.1, all the researchers have identified some limitations and challenges in the phase of *Requirements* which can serve as future research guidelines.

2.3 Design

Proper design decisions are crucial when it comes to energy-efficient software development, as the software components and their interactions can alter significantly energy consumption. In the subsequent subsections, we present studies that evaluate the energy implications of design patterns [56] or recommend adjustments for optimising them in terms of energy consumption. We also indicate cases where inappropriate design decisions lead to increased energy consumption.

Table 2.2: Design Patterns Empirical Evaluation on Energy and Run-Time Impact

Study Type	Implications (avg. in %)			Platforms	Source
	Pattern	Energy	Run-Time		
Empirical Evaluation	Flyweight	58.00	–	Embedded System	Sahin et al. [153]
	Proxy	36.00	–		
	Mediator	9.56	–		
	Composite	-5.14	–		
	Abstract Factory	-21.55	–		
	Observer	-62.20	–		
	Decorator	-712.89	–		
Empirical Evaluation	Decorator	-133.60	-132.40	Smart-phone	Bunse et al. [26]
	Prototype	-33.20	-33.00		
	Abstract Factory	-14.20	-14.20		
Patterns Optimisation	Observer	4.32	–	Workstation	Noureddine and Rajan [107]
	Decorator	25.47	–		

2.3.1 Empirical Evaluation of Design Patterns

Design patterns are general and reusable solutions to commonly appearing software design problems. In the context of design patterns, Sahin et al. [153] and Bunse et al. [26] performed empirical studies where they compared the energy consumption of selected patterns. Both studies evaluated design patterns from the *creational*, *structural*, and *behavioral* categories introduced by Gamma et al. [56].

Specifically, Sahin et al. examined the energy consumption of different applications[4] running on an embedded system, with and without the application of design patterns. The authors illustrated a method of correlating software design and energy consumption that helps software developers understand the trade-offs of their design decisions with energy consumption. For their experiment, they used 15 out of the existing 23 design patterns. As an outcome, three out of the 15 used patterns resulted in substantial energy savings, while the remaining resulted in mini-scale changes or negatively affected energy consumption.

[4] https://sourcemaking.com/

Particularly the *Flyweight*, *Mediator*, and *Proxy* patterns resulted in energy savings when applied on selected applications, while the *Decorator* pattern tremendously increased energy consumption.

Bunse et al. focused on evaluating the energy consumption and run-time performance impact of design patterns on Android applications. The authors observed an increase in both energy consumption and execution time after applying six out of the 23 design patterns (*Facade*, *Abstract Factory*, *Observer*, *Decorator*, *Prototype*, and *Template Method*) on selected applications.

The results summarised in Table 2.2 show that even in between instances of applying the same design patterns, large variations in energy consumption exist. For instance, according to the results of Bunse et al. the *Decorator* pattern increased energy consumption by 133%, while Sahin et al. found a significantly higher energy consumption, *i.e.*, 712%; this might occur because Bunse et al. employed a smart-phone and Java-based code snippets to experiment with, while Sahin et al. used an embedded system and code snippets written in C++.

In conclusion, both Sahin et al. and Bunse et al. observed the negative impact of some design patterns in embedded and smart-phone devices regarding energy consumption. Moreover, Bunse et al. showed that particular design patterns causing high energy consumption were also contributing to lower run-time performance. As stated in both studies, the number of objects and communications among the software components comprise primary factors for increasing energy consumption. Both studies identified the *Decorator* and *Abstract Factory* as the most energy-inefficient design patterns. Additional work using specific benchmarks may help understand in depth the effects of design patterns in diverse applications and platforms.

2.3.2 Energy Optimisation of Design Patterns

Structural changes in the design patterns can lead to significant energy optimisations as presented by Noureddine and Rajan [107]. Initially, the authors performed an experimental study where they manually applied 21 different design patterns on eleven applications written in both C++ and Java. Their experiment revealed the *Observer* and *Decorator* as the most energy-greedy design patterns. The authors transformed the existing source code, by reducing the number of created objects and function calls, and accomplished important energy savings. Specifically, after optimising the *Observer* and *Decorator* design patterns, the authors reduced the applications' energy consumption by 10%, on average.

The work of Noureddine and Rajan (like Sahin et al.) lacks run-time performance measurements. Consequently, it is uncertain how the design pattern optimisations affected an application's execution time. However, such design pattern optimisations seem to be a promising area for further investigation. To this end, a tool that indicates structural changes for design patterns which lead to optimised application energy consumption could benefit software practitioners.

2.4 Implementation

In the *Implementation* phase, there are several tools, techniques, and strategies that software practitioners can exploit to improve the energy consumption and run-time performance of their applications. To this end, we present research results associated with programming techniques, source code analysers, programming languages, and so on.

2.4.1 Parallel Programming

Parallel programming is the process of breaking a large problem into smaller ones to solve them simultaneously. In this section, we discuss works that empirically evaluate applications and algorithms that utilise parallel computing. Table 2.3 summarises implications of related thread management strategies and applied adjustments on energy consumption and run-time performance for different application types. Fields with negative values indicate an increase in energy consumption or execution time for the corresponding study.

2.4.1.1 Experimental Studies

By performing an empirical study, Kambadur and Kim [74] identified configurations and parameters that can reduce the energy consumption of parallel applications. The authors compared nine existing energy management strategies by using a standardised system architecture, OS, measuring tool, and five benchmark suites. The strategies such as `Processor Frequency Tuning, Overclocking, Parallelism, and Compiler Optimisation` were run on 220 experimental configurations and tested on 41 applications totaling in more than 200,000 executions. The obtained results highlight the importance of effective source code parallelisation by employing the appropriate number of threads (see Table 2.3).

To evaluate the energy efficiency of different thread management constructs, Pinto et al. [137] performed an empirical study comparing three constructs (*i.e.*, *explicit thread creation*, *fixed-size thread pooling*, and *work stealing*) on nine benchmarks by adjusting the number of threads, the task's granularity level, the data size, and the nature of the data access. The authors observed that the constructs' energy consumption vary in different situations. For example, *explicit thread creation* exhibits the lowest energy consumption when it comes to I/O-bound applications. For highly parallelised benchmarks, *work stealing* outperforms *explicit thread creation* and *fixed-size thread pooling* by being 30% more energy-efficient; however, for more serialised benchmarks, *work stealing* is under-performing. Also, the authors noticed that energy consumption increases as the number of threads increases. This occurs until the number of threads reaches the number of CPU cores. Afterwards, the authors detected a reduction in energy consumption for most of the tested applications.

As we can see in Table 2.3, the gains and losses concerning energy consumption and run-time performance are feasible by selecting the appropriate number of threads. For instance, both Pinto et al. and Kambadur and Kim tuned the numbers of threads to evaluate their applications. By using different thread management constructs and a various number of threads, Pinto et al. altered energy consumption from -50% to 30%, while Kambadur and Kim decreased it by 55%, on average. A major distinction between the two works, that can explain the discrepancies in results, is that Pinto et al. used Java applications, while Kambadur and Kim utilised Java and, also, C programs with compiler run-time optimisation flags.

Table 2.3: Parallel Programming Energy and Run-Time Impact

Thread Management	Adjustments	Implications (in %) Energy	Run-Time	Application Type	Source
Effective Parallelization[c]	From 1 to 16 Threads	55 avg.	69 avg.	CPU & GPU bound[d]	Kambadur and Kim [74]
Parallelization & Thread Management Constructs	Explicit Threading	–	–	I/O-bound[e]	Pinto et al. [137]
	Work Stealing	(-50) – 30	(-23) – 10	Embarrassingly Parallel[f]	
Pack & Cap[a]	Criticality level DVFS	56 avg.	Unaffected	CPU-bound[b]	Cai et al. [27]
Work Stealing[g]	Load-base DVFS	11–12	(-3)–(-4)	CPU & GPU bound[h]	Ribic and Liu [149]

[a] Collecting threads under the same core and reducing its frequency.
[b] Recognition-Mining-Synthesis (RMC), an Intel's application [35].
[c] Statically selecting the effective number of threads.
[d] Parsec 3.0, SPLASH-2X, SPEC CPU 2006, DaCapo 9.12, SPEC JBB 2013.
[e] Application such as *Largestimage*.
[f] Applications such as *Sunflow*, *Spectralnorm*, *N-Queens*, *Tomcat*.
[g] *"Underutilised processors take the initiative: they attempt to steal threads from other processors"* [20].
[h] Applications from Problem-Based benchmark suite [158].

As shown above, performing experiments with a varying number of threads or thread management constructs can help practitioners identify the configurations most likely to reduce the energy consumption and execution time of their applications. However, user interaction is mandatory to point out the best configurations and parameters through experimentation—a fact that makes the selection process time-consuming and cumbersome.

2.4.1.2 Algorithms

To take advantage of parallel processing, Cai et al. [27] and Ribic and Liu [148] developed algorithms to minimise the energy consumption by efficiently managing thread workloads. According to Cai et al., most of the Dynamic Voltage Frequency Scaling (DVFS) techniques—a mechanism that tunes CPU voltage to adjust its frequency based on the current workload—are not built to run on multi-threaded processors; therefore, they are unable to save a considerably large amount of energy. To this end, the authors suggested *thread shuffling*, a way of combining techniques such as thread migration and DVFS to reduce the energy consumption of an application without compromising its run-time performance. The basic idea behind *thread shuffling* is to identify threads with the same *thread critical degree* (slow threads execution) by using a prediction algorithm and map them under the same core via thread migration. Afterwards, the algorithm applies voltage dynamically to scale the cores' frequency for the cores housing non-critical threads (fast threads execution).

Likewise, Ribic and Liu introduced HERMES, a strategy for work-stealing that employs a *thief-victim* approach. The authors refer to *thief* as the thread which finishes its tasks and *steals* work from other threads, the so-called *victims*. HERMES is composed of two main algorithms, the *workpath-sensitive* and *workload-sensitive*. The *workpath-sensitive* algorithm defines a thread's tempo (execution speed) based on the *thief-victim*'s relationship; that means, when a thief steals from a victim worker, its tempo is set lower (because it always steals insignificant tasks) and it is raised once the victim runs out of work. The *workload-sensitive* algorithm is a work-flow based tuning mechanism for the workers execution tempo that, if necessary, adjusts the core's frequency via DVFS to reduce energy consumption. For instance, if a worker's queue is empty, it tries to steal work from other workers; afterwards, it adjusts its core's frequency according to its workload.

In the above works, both *thread shuffling* and HERMES utilise DVFS techniques with workload migration to accomplish energy savings. *Thread shuffling* packs all the threads with a similar criticality level under specific cores and then adjusts the cores' clock frequency accordingly, while HERMES alters the clock frequency of each core based on the threads' current workload. Particularly, the *work-stealing* approach of HERMES resulted in minor energy savings viz-a-viz the *thread shuffling* technique which resulted in 56% of energy savings, on average, as illustrated in Table 2.3. In conclusion, HERMES provided energy savings with a minor run-time performance penalty (*i.e.*, 3–4%), while *thread shuffling* achieved compelling energy savings without compromising run-time performance at all (see Table 2.3).

2.4.2 Approximate Computing

Approximate computing is an approach for sacrificing computation accuracy—when an application tolerates it—to increase run-time performance or energy savings [98]. For instance, techniques such as `loop perforation` are employed, that allow users to manage run-time performance and accuracy trade-offs based on the desired output quality [159]. In this section, we discuss studies divided into programming frameworks, memoisation, annotation-based, and directive-based extensions. In Table 2.4, we summarise the corresponding works.

2.4.2.1 Programming Frameworks

For their research, both Misailovic et al. [97] and Baek and Chilimbi [14], introduced energy-conscious programming frameworks to achieve energy savings through approximate computations. Specifically, Misailovic et al. proposed *Chisel*, an optimisation framework that acts in an automated manner by selecting approximate kernel operations that result in energy, reliability, and accuracy optimisations. Baek and Chilimbi suggested GREEN, a framework aiming to optimise expensive loops and functions by considering user-defined Quality of Service (QoS) requirements. GREEN achieves energy savings through approximate computations for functions and early loop termination. Moreover, it addresses the QoS and energy consumption trade-offs by applying approximate programming techniques only when the QoS requirements are fulfilled. A common element of both GREEN and *Chisel* is the comparison between precise and approximate instances for calculating the reliability of the results. However, what differentiates GREEN from *Chisel* is the periodical run-time QoS sampling to adjust its approximation techniques and QoS model to meet the target requirements. In contrast to GREEN, *Chisel* is utilising the approximate memory of its running platform to increase energy savings by sacrificing some of its quality output (see Table 2.4).

Table 2.4: Approximate Computing Energy and Run-Time Impact

Name	Optimisation Focus	Implications (in %)		Precision Loss	Source
		Energy	Runtime		
GREEN	Computations and loop termination	14 avg.	21 avg.	0.27	Baek and Chilimbi [14]
–	Function memoisation	74 avg.	79 avg.	< 3	Agosta et al. [10]
Parrot	Optimises regions of imperative code	66 avg.	56 avg.	10 avg.	Esmaeilzadeh et al. [52]
Chisel	Computational kernel operations	9–20	–	< 3	Misailovic et al. [97]
EnerJ	Computations, Data Storage, and Algorithmic[a]	10–50	–	–	Sampson et al. [157]
Axilog	Source code parts	54 avg.	–	10 avg.	Yazdanbakhsh et al. [175]
–	Selected group of tasks[b]	–	–	–	Vassiliadis et al. [171]
DCO Scorpio	Selected group of tasks[b]	56 avg.	–	–	Vassiliadis et al. [172]

[a] Programmer can write two different implementations, one is invoked when the data are precise and the other when they are approximate

[b] Allows the developer to set which computation tasks, from a group, are going to be executed approximately/precise

2.4.2.2 Annotation-Based Extensions

Sampson et al. [157], Esmaeilzadeh et al. [52], and Yazdanbakhsh et al. [175] proposed *EnerJ*, *Parrot* transformation, and *Axilog*, respectively; all are extensions that achieve energy savings by executing approximately annotated source code portions. Specifically, *EnerJ* is a Java-based extension, furnished with a manual annotation functionality for defining approximate or precise data selection for an application. By declaring variables and objects as approximate, *EnerJ* maps them to approximate memory[5] and generates low-cost energy code by using approximate operations and algorithms. *Parrot* transformation is a neural network model that identifies imperative code regions and offloads them to neural processing unit, instead of CPU, to increase energy and run-time performance. *Axilog* is a Verilog extension composing brief and high-level annotations for full control and governance of approximate hardware. In contrast to the manual annotating approach of *EnerJ* and *Parrot*, *Axilog* employs a Relaxability Interface Analysis algorithm to automate the approximation processes based on the designer's choices. All three annotation systems offer a safety mechanism for isolating approximate from precise portions of code, thus guaranteeing the main functionality of an application.

In contrast to GREEN, *Chisel*, and *Axilog*, with *EnerJ* and *Parrot* the developers are the wheel-holders of the applied approximation techniques, by selecting which code portions to

[5] memory parts with reduced voltage or refresh rate such as cache, registers, functional units, and main memory.

be executed as precise. Therefore, *EnerJ* and *Parrot* can help practitioners better understand the energy-approximation trade-offs of their design choices.

2.4.2.3 Directive-Based Extensions

Another approach for applying approximation techniques on computational tasks is to rely on their significance level (level of importance or criticality). To this line, Vassiliadis et al. [171, 172] proposed directive-based approaches by extending the OpenMP that use approximation techniques to reduce application energy consumption. In the first work [171], the authors proposed a programming model aiming to elicit the highest level of accuracy for an application according to a user-defined energy budget. Therefore, the authors introduced a run-time system that is responsible for choosing the appropriate configurations, (*i.e.*, number of cores, clock frequency, and accuracy ratio) for a specific input size. The appropriate configurations are inferred by a model which is trained to identify the configurations that provide the highest possible output accuracy for a given energy budget. Another approach is DCO/Scorpio, a framework suggested by Vassiliadis et al. [172] that supports automated analysis to identify the code's significance level. DCO/Scorpio accomplishes that by employing *interval arithmetic* [143] and *algorithmic differentiation* [106] to quantify the significance of particular computations for a specific input. This output is in turn used by an OpenMP-like model to classify the computations in task groups according to their significance level. Thus, it provides approximate methods based on each group's significance level.

DCO/Scorpio and the work presented in reference [171] differ in that the former reduces the energy consumption of an application, while the latter achieves the highest possible output quality within a specific energy consumption threshold. Also, DCO/Scorpio is releasing the hands of a user by selecting the computation tasks significance, while Vassiliadis et al. [171] require the programmer's involvement to input directives for critically important parts of the code that do not tolerate imprecision.

2.4.2.4 Memoisation

Another approach for saving energy through approximate computing is by using memoisation to store expensive function call results. Agosta et al. [10] developed a performance model to select and memoise computationally intensive function from financial applications and JavaGrande benchmark. Compared to the above approaches, memoisation seems to have the highest energy savings and run-time performance (see Table 2.4). However, the authors did not apply their method to real-world applications.

2.4.3 Source Code Analysis

Source code analysis is a testing process that focuses on revealing defects and vulnerabilities in a computer program before its deployment phase. In this section, we discuss works on dynamic source code analysis that aim to identify energy-related bugs and hot-spots by testing a computer program at real-time. In the context of source code analysis, we did not find available tools for static code analysis that provide rules for analysing source code before execution. Table 2.5 shows works on source code analysis and provides information on the target platform, energy measurements correlation at various software granularities, and the error margin rate.

Table 2.5: Source Code Analysis Related-Work Information

Tool Name	Platform	Energy Measurement Correlation	Error Margin (in %)	Source
Eprof	Android & Windows OS	Process, Threads, System calls, Routines	6<	Pathak et al. [123]
eLens	Android	Full Source Code Granularity	10	Hao et al. [62]
GreenAdvisor	Android	Routines, System-calls	–	Aggarwal et al. [8]
PEEK	Embedded	Function Level Systems	–	Honig et al. [70]
SEEDS	Java Platforms	User can set which portion to analyse	–	Manotas et al. [93]
–	Smart-phones	Function Level	–	Banerjee et al. [17]

2.4.3.1 System Calls

GreenAdvisor is a system calls profiler that predicts behavior, run-time performance, and energy-related modifications of an application [8]. To predict energy-related changes, *GreenAdvisor* compares the number of system calls on a current version of an application and its previous one. If energy consumption increases, it pinpoints the energy hot-spots that caused the changes, thus helping developers in analysing and understanding the implications of their decisions.

To diagnose energy bottlenecks at the source code level, Pathak et al. [123] presented *Eprof*, a system-call based power modeling tool for smart-phone applications. To achieve high accuracy in energy consumption measurements, *Eprof* incorporates two subsequent components. First, it uses finite state machines to model different power states and transitions for individual hardware components, and the smart-phone as a whole. Then, for each hardware component, it runs a benchmark suite that consists of applications for collecting the different system calls and their power transitions. Afterwards, it generates rules for the finite state machines by integrating the collected knowledge from the executed applications. To estimate energy consumption at the source code level, *Eprof* cross-references routines (blocks of code) with system call traces. By using Eprof, the authors found that applications using third-party processes tend to have a 65–75% increase in energy consumption.

Overall, the above-discussed tools are utilising distinct energy estimation models to present energy measurements. For example, *Eprof* employs a model that calculates the energy consumption of an application at the routine level and various hardware components, while *GreenAdvisor* compares the energy consumption of different versions of a product and points out the system calls that caused the change.

2.4.3.2 Optimisation Tools

Honig et al. [70] and Manotas et al. [93] recommended tools for dynamic source code analysis. Both Honig et al. and Manotas et al. suggested energy-aware programming approaches to guide developers during the implementation phase by providing energy-related hints. Honig et al. presented the Proactive Energy-awarE development Kit (PEEK), while Manotas et al. promoted the Software Engineer's Energy-optimisation Decision Support (SEEDS) framework. Periodically, both approaches analyse the source code under development and seamlessly create many different instances from the current source code to identify optimal code modifications. However, a significant difference between the two approaches is that PEEK's energy-associated hints are related to power management mechanisms (*e.g.*, sleep state, DVFS, idle state, program-code logic, libraries, and compiler run-time optimisation flags) while SEEDS' suggestions are limited to Java Collection Libraries (JCL), algorithms, or refactoring parts of the source code. Besides, SEEDS offers to developers the possibility to set a code block range for analysis while PEEK analyses source code at function granularity.

2.4.3.3 Tests Generation Framework

Another way for identifying energy-related bugs is through an automated test generation framework, introduced by Banerjee et al. [17]. The aforementioned research points out energy-related hot-spots in four categories of smart-phone applications which are 1) *hardware resources*, 2) *sleep-state transitions*, 3) *background services*, and 4) *defective functionality*. First, a detection process is invoked to search for possible user interactions through event flow graphs. Then, the advocated framework generates test cases to capture interaction scenarios and, subsequently, to identify energy hot-spots. Alongside the energy hot-spots identification, the tool issues test reports for the developers. However, compared to the work of Honig et al. and Manotas et al., the work of Banerjee et al. does not provide hints for energy optimisation, but only finds the energy-wasteful parts of an application.

2.4.3.4 Line-by-Line

To measure application energy consumption at different levels of the software granularity and raise energy-awareness during the development phase, Hao et al. suggested *eLens*. *eLens* estimates energy consumption via program analysis and per-instruction power modeling. The authors use program analysis to obtain execution-related information such as bytecode or API calls from various smart-phone components (*e.g.*, CPU, RAM, GPS, 3G). Then, the collected information is passed to the per-instruction power modeling component to estimate an application's energy consumption. Thus, *eLens* provides energy measurements at various levels of granularity: application, method, class, path, and source code lines. Compared to all presented tools, *eLens* is the only one that provides energy measurements at all levels of source code granularity, thus raising energy-awareness by providing fine-grained information on application consumption.

2.4.4 Programming Languages

Programming languages offer a set of instructions that allow users to utilise system resources to solve a problem. Languages differ in features and the way they allocate and use computing resources. In this section, we examine empirical studies investigating the energy consumption and run-time performance implications of programming languages. We also discuss studies related to the energy and run-time performance of compiler optimization flags. Table 2.6 lists related works in terms of energy consumption, run-time performance, employed optimisation flag, and test cases. We summarise related results in a consolidated list, found

Table 2.6: Programming Languages Energy and Run-Time Impact

Comparing X to Y	Implications (avg. in %) Energy	Performance	Flags	Env.	Apps	Source
C++ to C	8.40	8.67	–O3	–	All Apps[a]	Abdulsalam et al. [5]
C++ to Java	47.40	38.12				
C++ to Python	166.28	195.17				
C to Java	38.39	29.08				
C to Python	106.40	195.50				
Java to Python	195.87	194.77				
C++ to Java	166.14	159.13	–O3	Dalvik[b]	Quick Sort	Chen and Zong [34]
C++ to Java	Identical	Identical		ART[c]		
C to C++	Identical	3.38	–	–	Fibonacci	
ARM-assembly to C	15.38	10.52	–	–	Counting Sort	Rashid et al. [145]
ARM-assembly to Java	58.84	90.90			Counting Sort	

[a] Fast Fourier, Quick Sort, Linked List
[b] https://source.android.com/devices/tech/dalvik/
[c] Android Run-Time (ART) https://source.android.com/devices/tech/dalvik/

in Table 2.6, of average values calculated by us.[6] Table 2.8 illustrates works related to compiler types, used optimization flags, target platforms, and test cases.

Table 2.7: Programming Languages Configurations

Programming Languages	Optimisation Flags	Target Platforms	Test Cases	Source
C, C++, Java, and Python	–O{1,2,3}	Server system	Fast Fourier, Linked List, Quick Sort	Abdulsalam et al. [5]
ARM-assembly, C, and Java	–	Embedded system	Bubble, Counting, Merge Quick	Rashid et al. [145]
C, C++, and Java	–O{1,2,3}	Android devices	Fibonacci, Tower of Hanoi, Pi calculation	Chen and Zong [34]

2.4.4.1 Different Programming Languages

The studies presented in this section are based on different experimental platforms. Particularly, Abdulsalam et al. [5] performed their tests on a workstation, Rashid et al. [145] used an embedded system, and Chen and Zong [34] conducted their experiments on a smart-phone. However, Abdulsalam et al. and Chen and Zong used similar testing parameters, as depicted in Table 2.7. Furthermore, Abdulsalam et al. also compared the energy implications of four

[6] https://github.com/stefanos1316/Proof_for_Survey/blob/master/Programming_Languages_Average_Values.txt

memory allocation methods (*i.e.*, `malloc`, `new`, `array`, and `vector`) where they presented `malloc` as the most energy- and run-time-performance-efficient.

In their experiment, Chen and Zong utilised the Native Development Kit[7] tool-set for executing native code such as C and C++ inside Android applications. The derived output, by both Abdulsalam et al. and Chen and Zong, is that C and C++ achieved significant energy savings and, also, reduced execution time against the other programming languages. Also, both works showed that the run-time compiler optimisation flag `-O3`, had the most significant energy savings and increased run-time performance for workstation and smart-phone applications. Moreover, for Java applications, Chen and Zong showed that the use of Android Run-Time instead of Dalvik run-time environment, contributed to energy and run-time performance results similar to the C and C++ implementations.

In the context of embedded systems, Rashid et al. performed an experiment to compare the energy and run-time performance implications of four sorting algorithms written in ARM-assembly, C, and Java. By performing their experiments on a Raspberry Pi,[8] the authors showed that the implementations of ARM-assembly achieved the most energy-efficient results viz-a-viz the C and Java implementations. Likewise, Rashid et al. presented Java as the most energy-hungry among the selected programming languages.

Table 2.6 illustrates the superiority of compiled programming languages against the interpreted and semi-compiled,[9] in terms of energy consumption and run-time performance. Java and Python suffer the most from high energy consumption and low run-time performance. The use of an interpreter makes Python slower and less energy-efficient, because it has to interpret source code for each execution. Moreover, the dynamic compiling, library linking, and interpretation of byte-code in JVM are additional burdens on the execution of Java programs. However, Chen and Zong showed that the use of the Android Run Time environment could reduce the energy consumption and increase run-time performance of Android applications.

A limitation that we observed, for all the discussed works, is the lack of information regarding the versions of the employed compilers, interpreters, modules, and libraries used in the experiments. Additionally, we did not find any research study that compares the energy consumption and run-time performance implications of the same application across different compiler versions, run-time engine, or interpreter.

2.4.4.2 Compiler Optimizations

Compilers are computer programs that translate high-level language code to binary code. Also, they offer a large number of options to optimize run-time performance, memory usage, or even protect the source code against malicious uses. Several studies have investigated the run-time and energy performance of the GCC compiler. Specifically, Pallister et al. [115] examined the energy consumption of several GCC's performance optimization flags by using fractional factorial design [60] to account for interactions among the optimization on various multi-purpose benchmarks and different embedded systems. Their results suggest that most of the optimization flags affected energy consumption and run-time performance in the same way. Also, the authors have shown that it is possible to achieve energy savings up to 4%

[7] `https://developer.android.com/ndk/index.html`

[8] `https://www.raspberrypi.org/`

[9] Semi-compiled languages compile source code into intermediate code and execute it on a VM.

Table 2.8: Compiler Optimizations Energy and Run-Time Impact

Compilers	Flags	Platform	Test Cases	Source
GNU GCC	82 options	Embedded systems[a]	MiBench[b] and WCE[c]	Pallister et al. [115]
GNU GCC	Run-Time Performance	Laptop	6 tasks[d]	Branco and Henriques [25]
GNU GCC	Run-Time Performance	Embedded system	MMC suite	Patyk et al. [124]
MigGW, Borland C++, Visual C++, Gygwin	Run-Time Performance	Workstation	Sample Code	Hassan et al. [65]

[a] cortex-m0, cortex-m3, cortex-a8, xmos, epiphany
[b] A free, commercially representative embedded benchmark suite
[c] Worst Case Execution Time
[d] MMC, Grades, Bzip, Bzip2, Oggenc, and Pbrt

after compiling applications with a certain set of compiler flags. Likewise, Patyk et al. [124] suggested a statistical approach to automatically identify the optimal compiler options to reduce energy consumption. The authors achieved energy savings of 15% on average.

Some authors showed that compiler run-time optimization flags (-O1 to -O4) contribute to reduced energy consumption. In particular, Branco and Henriques [25] investigated the energy implications of C, C++, Objective-C, and Go programs. By experimenting on 12 benchmarks, the authors found that optimised code equals to greater energy savings. Similarly, Hassan et al. [65] have investigated the energy implications of run-time optimization flags but on different compiler *i.e.*, MinGW GCC, Cygwin GCC, Borland C++, and Visual C++. Their results suggest that major energy and run-time performance gains exist among the different compilers.

In conclusion, the above works present that major energy savings are available through compiler optimizations. Nevertheless, none of the above studies have investigated the energy consumption of GCC's safeguards against buffer overflow attacks (stack canary), read-only relocation (RERLO), address space layout randomisation protections, and so on.

2.4.5 Data Structures

A data structure is a way to organise, manage, and store data for further process or analysis. This section consists of *Empirical Studies* and *Tooling Support* for data structures. In the *Empirical Studies* part, we discuss works trying to identify which data structures are the most energy-efficient for particular cases. For the *Tooling Support*, we show tools that inform a practitioner which data structure to select to reduce energy consumption. Table 2.9 summarises works on the data structures collection interface and library, energy consumption, and tested applications for the relevant source.

Table 2.9: Data Structures Energy Impact

Interface	Library	Data Structure	Energy (in %)	Apps	Source
	C5	HashedLinkedList	23.27	Apps[a]	Michanan et al. [96]
	JCF	AttributeList	24.88	CEB[b]	Pereira et al. [126]
List	JCF	ArrayList	38.00	Gson	Hasan et al. [64]
		LinkedList	-309.00	SETS[c]	
	JCF	LinkedHashMap	50.16	CEB[b]	Pereira et al. [126]
Map	JCF	ConcurrentHashMapV8	17.80	XALAN	Pinto et al. [138]
	JCF	ConcurrentHashMapV8	9.32	TOMCAT	
Set	C5	HashSet	31.44	Apps[a]	Michanan et al. [96]
	JCF	LinkedHashSet	12.50	CEB[b]	Pereira et al. [126]
Queue	JCF	LinkedTransferQueue, LinkedBlockingDeque, ConcurrentLinked-Deque, PriorityQueue, ConcurrentLinkedQueue	7.50	Apps[d]	Manotas et al. [93]
Bag	C5	HashBag	16.93	Apps[a]	Michanan et al. [96]

[a] A* Path Finder, Huffman Encoder, Genetic Algorithm
[b] https://github.com/greensoftwarelab/Collections-Energy-Benchmark
[c] Stock Exchange Trading Simulator
[d] Barbecue, Jdepend, Apache-xml-security, Joda-Time, Commons Lang, Commons CLI

2.4.5.1 Experimental Studies

By conducting experimental studies, Pereira et al. [126], Hasan et al. [64], and Pinto et al. [138] identified some positive and negative cases concerning the energy consumption of various data structures. The authors evaluated the energy consumption of data structures from different interfaces, but mostly from the Java Collection Framework (JCF).

Pereira et al. performed an experimental study to evaluate the energy efficiency of different JCF interfaces methods such as search, iteration, removal, and insertion. By manually replacing data structures in applications, the authors obtained significant energy savings as illustrated in Table 2.9. Because the authors of the above work report an extensive list of results, we compared the energy consumption of the most and least energy-efficient interface,[10] and we present in Table 2.9 the data structure with the most energy-efficient methods for each interface.

By examining the memory usage and analysing byte-code traces of Android applications, Hasan et al. evaluated the energy consumption of different collection types. Apart from JCF, the authors used data structures from Apache Commons Collections (ACC) and Trove.[11] As an outcome, the authors showed that energy consumption starts to diverge among the data structures only when the number of elements they contain is above 500. Additionally, the authors noted for the data structures with primitive data types (found in Trove collection) that, while consuming less memory than objects, they are more energy-inefficient in most of the cases.

[10] https://github.com/stefanos1316/Proof_for_Survey/blob/master/Pereira_Data_Structure.txt
[11] http://trove.starlight-systems.com/

Similarly, Pinto et al. used 13 thread-safe and three non-thread-safe implementations of JCF. The authors tested the above data structures by utilising distinct configurations such as the number of threads, initial capacity, and load factor. As a result, the authors observed that the proper data structure selection and the number of threads (for most of the thread- and non-thread-safe implementations) can decrease applications energy consumption. For example, when the authors replaced the `HashTable` instances with `ConcurrentHashTableV8`, in real-world benchmarks, they achieved significant energy savings.

Overall, Pereira et al. showed that the same kind of method implementations (*e.g.*, add, remove, search) affect the energy consumption of data structures differently. In Table 2.9, we can see that Pereira et al. achieved the highest energy savings in their experiments. However, compared to Hasan et al. and Pinto et al., Pereira et al. used micro-benchmarks and not real-world applications. Pinto et al. examined the energy impact of real-world applications by i) replacing non-thread-safe with thread-safe data structures and ii) changing the number of threads. This helped them to achieve substantial energy savings as illustrated in Table 2.9. Furthermore, the results by Hasan et al. reveal the negative impact of unwise data structure selection which can significantly affect energy consumption (see Table 2.9).

2.4.5.2 Tooling Support

Predicting the most energy-efficient data structures for a given problem can decrease energy consumption. To this end, Michanan et al. [96] introduced GreenC5, a tool that can predict which data structures among the Copenhagen Comprehensive Collection Classes for C# (C5) collection can reduce application energy consumption based on the system's workload. GreenC5 is composed of a predictive algorithm based on machine learning and neural network models. As shown in Table 2.9, Michanan et al. achieved energy savings for real-world applications.

Manotas et al. [93] introduced SEEDS (also discussed in subsection 2.4.3.2), a decision support framework that can dynamically evaluate Java collection types and modify them to reduce the energy consumption of an application. To do that, SEEDS creates instances of an application under development using different `queue` data structures, to find the most energy-efficient ones. This helped Manotas et al. gain energy savings, in real-world applications, while choosing the proper `queue` type data structures.

For the tooling support, we observe several shortcomings such as a limited number of collection types (SEEDS is available only for `queue`) or focus on specific collections (GreenC5 uses only the C5 collection). However, these tools can assist a developer in selecting and experimenting with various data structures to obtain sufficient energy savings. Moreover, further support on various data structure types and different collections can make such tools more beneficial and attractive to the developers.

2.4.6 Coding Practices

Best coding practices are sets of rules, formally or informally, established by various coding communities that help software practitioners to improve software quality. In this section, we discuss works on empirical evaluation that examine the energy consumption of coding practices. Table 2.10 summarises results identified from the related works as "Good" and "Bad" coding practices. By the terms "Good" and "Bad", we refer to coding practices that

Table 2.10: Code Practices Energy and Run-Time Impact

Coding Practice	Practice Implication (in %) Practices	Energy	Run-Time	Platform	Source
Good	*For* loop with length	36–52	33–38	Android	Tonini et al. [168]
	For loop with length	10 avg.	–	Android	Li and Halfond [85]
	Efficient query usage	25.1 avg.	24.9 avg.	Linux	Procaccianti et al. [140]
	Put application to sleep	8.48 avg.	6 avg.		
	Change macros to function calls, loop unrolling, reducing lookUp tables sizes[a]	179 avg.	–	–	?]
Bad	Avoid getters&setters	24–27	24–30	Android	Tonini et al. [168]
	Avoid getters&setters	30–35	–	Android	Li and Halfond [85]
	Invoke static methods	15 avg.	–	Android	
	Avoid using relational database	–	–	Android	Linares-Vásquez et al. [90]
	Avoid unnecessary views and widgets				

[a] Comparing RC6 against Twofish, also run-time performance was calculated in clock cycles instead of time, therefore, we did not adding it in our results

may impact a program's readability, maintainability, efficiency, and usability positively or negatively, respectively.

In terms of embedded systems, **?**] evaluated the run-time performance, energy consumption, memory usage, and code size of five block ciphers (*i.e.*, RC6, Rijndael, Serpent, Twofish, and XTEA) on a StrongARM SA-1100 processor. The authors modified the existing source code of the cipher algorithms (to reduce their lines of code) by:

- Replacing `macros` with `function calls`.
- Using `loop unrolling` in the encryption and decryption functions.
- Replacing `T-lookup` with forward and inverse `S-box` tables and reducing their sizes.

Their results show that the block ciphers XTEA and RC6 (which had the smallest code size) were the most energy-efficient, offered the best run-time performance, and utilised less main memory for the encryption and decryption tasks.

In their experiment, Tonini et al. investigated the energy efficiency of best practices for Android development. Their results indicate that the proper use of `for loop` and `getters/setters` can improve energy consumption. Initially, the authors performed experiments by using different variations of `for loop`, *i.e.*, `for-each`, when the loop's termination condition is (1) calculated at each iteration, and (2) when it is passed as a variable. In addition, the author evaluated scenarios with and without `getters/setters` to access the class fields. Their results show significant energy savings by using a variable as the loop termination condition and accessing class variables without using `getter/setter` functions (see Table 2.10).

Likewise, Li and Halfond checked practices such as HTTP request bundling with specific size and memory usage, and performance tips. Particularly, for the performance tips, the authors inspected coding practices, obtained from the Android developer forum.[12] The application of coding practices helped Li and Halfond to obtain notable energy savings as illustrated in Table 2.10. The practice of avoiding calculating a data structure's length in a loop proved beneficial since, having the loop's termination condition in a variable saves energy by bypassing the calculation of the length at each iteration. The practice of direct field access also proved beneficial because no additional function call is required from the system, unlike the case where a field value is retrieved through method invocation. Finally, the practice of static invocation proved energy-efficient because calling a method statically saves energy as it avoids the lookup overhead for calling methods through an existing object.

Android offers a variety of Application Programming Interface (API) calls and if not used efficiently these can contribute in increased energy consumption [90]. In their study, Linares-Vásquez et al. performed an analysis on 55 Android applications from various domains and they listed the most energy-inefficient API methods. Moreover, the authors suggested a list of practices that can yield energy savings by effectively using API calls. To obtain their results, the authors correlated `timestamps` of a method's execution traces with energy consumption measurements. After analysing their results, they identified 133 energy-greedy APIs out of the total of 807. From the energy-greedy APIs, 61% are related to graphical user interface and image manipulation, while the remaining 39% fall under the category of database. In conclusion, the authors highlighted that the unnecessary refreshing of `views` (*e.g.*, redrawing a `view` upon receiving new data) and widgets can consume a significant amount of energy. In addition, they highly recommend users to avoid relational databases, such as SQLite,[13] when it is not of paramount importance.

Two best practices were recommended and evaluated by Procaccianti et al. [140]. The selected practices were *put the application to sleep*, *i.e.*, put processes or threads on sleep state if they are waiting for I/O operations or they are no longer active, and *use efficient queries i.e.*, avoid the use of expensive energy operations such as ordering or indexing when not needed. As an outcome, the authors achieved energy efficiency by using both practices and, also, increased run-time performance (see Table 2.10).

The results in Table 2.10 show that the usage of `for` with a given length size contributes to energy savings from 36% up to 52% according to Tonini et al. and Li and Halfond, respectively. Furthermore, in both works the authors avoided the use of `getters/setters` and saved energy in the range of to 24–27% [168] and 30–35% [85]. Although both studies used micro-benchmarks, their results offer a different scale of energy savings. This may have occurred because they employed different hardware devices, distinct Android versions, and different tools to obtain their energy measurements. From Table 2.10, we can also observe that **?** achieved significant energy savings (*i.e.*, 179%) by using the corresponding coding practices. However, it should be noted that the authors compared the block ciphers with one another instead of the original and optimised versions.

To sum up, there are opportunities for energy consumption and run-time performance improvements even by applying minor code changes such as passing a variable in a loop's termination condition or by invoking a method statically [168, 85]. Moreover, the proper API selection that demands fewer system resources can also reduce energy consumption,

[12]https://developer.android.com/training/articles/perf-tips.html
[13]https://www.sqlite.org/about.html

according to Linares-Vásquez et al. Additionally, Procaccianti et al. improved energy consumption and run-time performance by reducing the use of expensive database operations (when these were not mandatory) and putting applications to sleep (when they were not performing any action).

Table 2.11: Remote IPC Energy and Run-Time Impact

Remote IPC	Applications	Platforms	Results	Source
SOAP and REST	Various message sizes	Android	REST more energy-efficient	Mizouni et al. 2011
REST and WebSockets	Various message sizes	Android	REST more energy-efficient	Boven and Hennebert 2012
REST	Axis2 and CXF	RPis	Axis more energy-efficient	Nunes et al. 2014
REST and WebSockets	Various message sizes	Android	WebSockets are more energy-efficient	Herwig et al. 2015
REST, SOAP, WebSockets, gRPC	Bubble, Insertion, and Heap sorting	Android	REST and SOAP are energy-efficient	Chamas et al. 2017

2.4.7 IPC Technologies

Most modern information technology devices use the Internet for creating, reading, updating, and deleting shared data through remote IPC. In this section, we discuss work on IPC technologies and we illustrate then the form of table (see Table 2.11).

Herwig et al. [67], Chamas et al. [33], Boven and Hennebert [24], and Mizouni et al. [100] performed experimental studies to identify the energy consumption of popular IPC technologies (see Table 2.11). More specifically, Chamas et al. performed an experiment where they investigated the energy consumption of three sorting algorithms (bubble, insertion, heap sort) by using three different input sizes (1.000, 10.000, and 100.000). The authors performed their experiments locally (on an Android phone) and remotely (server offloading) to examine REST, SOAP, WebSocket, and gRPC IPC technologies. Their results suggest that (1) the size of data indeed affects energy consumption, (2) the complexity of the sorting algorithms significantly affects energy consumption, (3) local execution can save more energy than remote for small input sizes, and (4) REST and SOAP are the most energy-efficient architecture styles.

Mizouni et al. investigated the energy consumption and run-time performance of SOAP and RESTful web services. They showed that a RESTful web service not only has 10% lower energy consumption against SOAP but has also 30% better run-time performance. Bovet and Hennebert compared the energy consumption of RESTful and WebSocket web service in the context of smart phones and also showed that RESTful web services are far more energy-efficient [24]. Similarly, Herwig et al. investigated the energy consumption of REST and WebSockets by sending and receiving data packets on three different network types *i.e.*, WLAN, 3G, and Edge. They showed that REST consumes more energy viz-a-viz the WebSockets; however, this contradicts Chamas et al., Mizouni et al., and Boven and Hennebert who proved

REST as the most energy-efficient IPC.

To evaluate the run-time performance and energy consumption of RESTful web services built on two well-known frameworks (*i.e.*, Axis2 and CXF), Nunes et al. performed an experimental study in the context of a Raspberry Pi platform [111]. In their experiment, they compared the marshaling and unmarshaling of different message sizes and also different CPU clock frequencies. Their results illustrate that the Axis2 framework can offer efficiency in energy consumption and better run-time performance. Also, they found that CPU overclocking contributes to reduced energy consumption and faster execution time.

According to the above literature, we point out that all of the studies related to the energy consumption of IPC technologies focus on embedded systems or mobiles devices, while workstations are out of the context. Likewise, the focus is mostly on Java frameworks that utilise IPC technologies.

2.5 Verification

In this section, we discuss a range of tools to measure and test the software's energy and power consumption after the development of an application. We divide the collected works into *Benchmarks* and *Monitoring Tools* in Sections 2.5.1 and 2.5.2, respectively.

Table 2.12: Benchmark-Related Work Information

Tool Name	Target Platform	Energy Correlation With	Optimisation Level	Source
GBench	Linux	Hardware Components	Data memory movement, block size, number of cores	Subramaniam and Feng [161]
ALEA	Linux	Code blocks	Compiler optimisation, power capping, DVFS, thread throttling	Mukhanov et al. [104]
AxBench	Linux	Hardware components	Source code via approximation	Yazdanbakhsh et al. [176]
Power Bench	TinyOS	Each node of the test-bed	–	Haratcherev et al. [63]

2.5.1 Benchmarks

Benchmarks are tools consisting of two main components: (1) a profiling tool, responsible for obtaining instructions used from a specific execution and (2) a performance benchmark, that generates workloads for a system. The above components combined, perform energy consumption measurements. Table 2.12 summarises collected information in terms of target platform, energy measurement correlations with hardware or software components, optimisations applied, and the related resources.

PowerBench is a scalable test-bed infrastructure that benchmarks and retrieves power consumption traces, in parallel, from various wireless sensor nodes of a cluster [63]. To do that, *PowerBench* utilises particular hardware and software components to offer an off-line processing, analysis, debugging, and visualisation of the elicited power measurements. The features of such an approach can aid developers to detect power consumption fluctuations and anomalies in clusters and complex IoT environments.

Subramaniam and Feng [161] proposed Green Benchmark (*GBench*), an approach that employs the Load Varying-LINPACK[14] that produces a variety of workloads to evaluate the energy consumption of a system. The authors tested *GBench* with different configurations such as block size, work-loads, number of cores, and memory access rate. As an outcome, they detected a correlation between the energy consumption and run-time performance for the second level (L2) of cache misses, on specific workloads.

Mukhanov et al. [104] suggested Abstract-Level Energy Accounting (ALEA), a highly accurate (mean error of 1.4%) portable tool that retrieves energy measurements from any micro-processor architecture. At its core, ALEA has a fine-grained energy profiling tool that retrieves measurements from basic code blocks. For evaluation, the authors employed well-known benchmark suites such as SPEC 2000,[15] SPEC,[16] OMP,[17] etc., and analysed the relation between basic code blocks' energy consumption and cache accesses to identify energy hot-spots. As a result, by using ALEA, the authors achieved 37% of energy savings, on some of the mentioned benchmarks, through different kinds of energy-efficient strategies and adjustments, *e.g.*, concurrency throttling, thread packing.[18] Moreover, they revealed a strong correlation between energy consumption and cache access rate.

AxBench, proposed by Yazdanbakhsh et al. [176], is a benchmark suite that supports tests in diverse domains such as finance, signal processing, image processing, machine learning, and so on, aiming to evaluate a system's run-time and energy performance by exploiting approximation techniques. Specifically, the approximation techniques supported by *AxBench* consist of `loop perforation` and `neural processing units`. *AxBench* obtains energy measurements of the CPU, GPU, and Axilog hardware [175]. *AxBench* offers (1) the feature to test various levels of the computing stack (software and hardware), (2) various test inputs, and (3) application-specific quality metrics, *i.e.*, average relative error for numeric output, miss rate for `boolean` result, and image difference. However, to perform benchmarking, a user has to identify and manually annotate regions of code that can tolerate imprecision.

The major difference between *GBench* and ALEA is that the latter correlates energy measurements with basic code blocks, while the former maps the energy consumption of the entire application to hardware components. *AxBench* is equipped with benchmarks for CPU and GPU aiming to offer a fine-grained understanding of their energy and run-time performance implications. In contrast to the above benchmarks, *PowerBench* is the only one to evaluate the energy consumption of an IoT-like infrastructure and cluster.

[14]`https://www.top500.org/project/linpack/`
[15]`http://www.spec2000.com/`
[16]`https://www.spec.org/`
[17]`https://www.spec.org/omp2012/`
[18]collecting threads under a specific number of cores

2.5.2 Monitoring Tools

To derive the energy consumption of a computer system, two approaches currently exist: (1) indirect energy measurements through estimation models or performance counters and (2) direct measurements, through hardware energy analysers and sensors. In this section, we discuss tools that are using indirect and direct approaches to perform measurements, some of the tools we present, analyse running applications or system calls to estimate energy consumption. However, compared to the source code analysis tools presented in Section 2.4.3, the energy monitoring tools only report the energy consumption of an application without pointing out energy hot-spots or providing hints for improving the spotted deficiencies. Tables 2.13 and 2.14 present the discussed software and hardware energy monitoring tools. The tables depict a variety of information such as tool names, target platform, measurement types, sampling intervals, and median error rates.

2.5.2.1 Software Energy Monitoring Tools

We use the term "software energy monitoring tools" for software-based analysers that utilise performance counters or estimation models to measure the energy consumption of running applications. We analyse the monitoring tools concerning their features, limitations, architecture (energy estimation model), and supported OS. Moreover, we further classify the software energy monitoring tools according to the platform they target into three categories: (1) workstations and servers, (2) VMs, and (3) smart-phones.

Workstations and Servers Running Average Power Limit (RAPL) monitors and controls energy consumption via performance counters [116]. By utilising the Linux kernel pseudo file system (`sysfs`), RAPL sysfs exposes kernel subsystems, hardware devices, and device driver information from the kernel to userspace allowing the estimation of the software's energy consumption. Liu et al. [92] introduced jRAPL,[19] a framework that combines RAPL and the Java Native Interface to measure the energy consumption of CPU, RAM, and Package[20] components for a Java application. Apart from jRAPL, Pantels et al. [120] and Pantels [119] introduced the tools PowerGadget and SoCWatch, respectively. Both tools are using RAPL to elicit power consumption from the CPU's performance counters. PowerGadget retrieves package power metrics exposed by the CPU and GPU, and can be integrated within a user's application through a C++ API. SoCWatch provides power-related information for the CPU's and GPU's `C-` and `P-state` residencies. All the discussed RAPL-based tools have a high sampling rate and can retrieve a substantial number of samples per millisecond as shown in Table 2.13. However, tools incorporating RAPL work under specific hardware limitations such as particular microprocessor architecture [116]. For instance, PowerGadget is compatible only with Intel's second generation CPUs, and it is not yet supported for architectures such as *Skylake*, *Broadwell*, and *Haswell*.

To estimate application energy consumption, Noureddine et al. [108], Bourdon et al. [23], and Noureddine and Rajan [107], proposed a number of tools. Specifically, Noureddine et al. proposed Jalen,[21] a Java agent, which, when attached to an application, ti gathers

[19]`https://github.com/kliu20/jRAPL`

[20]the core (all CPU cores) and the un-core (GPU, LLC, etc.)

[21]`https://github.com/adelnoureddine/jalen`

Table 2.13: Software-based Monitoring Tools Related-Work Information

Name	Target Platform	Measurement Type	Sampling Rate (msec)	Median Error	Source
Jalen	Linux	Energy	500	–	Noureddine et al. [108]
PowerAPI	Linux	Energy	500	0.5–3	Bourdon et al. [23]
jRAPL	Linux	Energy	1	1.13	Liu et al. [92]
Jolinar	Linux	Energy	500	3	Noureddine and Rajan [107]
RAPL	Linux	Energy	1	3	David et al. [44]
SoCWatch	Windows, Linux, & Android	Power	1–1000	–	Pantels [119]
Power Gadget	Windows & Linux	Power	1–1000	–	Pantels et al. [120]
JouleMeter	VMs & Windows	Power	1000	5	Liu et al. [91]
VMeter	VMs	Power	–	6	Bohra and Chaudhary [21]
BitWatts	VMs	Energy	500	2	Colmant et al. [38]
Power Booter	Android	Power	–	0.8	Zhang et al. [178]
Green Oracle	Android	Energy	–	10	Chowdhury and Hindle [36]
AEP	Android	Power	–	–	Chen and Zong [34]
PETrA	Android	Energy	1000	0.04	Di Nucci et al. [47]

energy measurements after the initialization of the JVM. Jalen measures selected methods and classes. Additionally, it provides measurements on explicit hardware components (*e.g.*, CPU and HDD) and estimates the energy consumption of software by analysing executed Java instructions. Jolinar[22] is a Java-based tool for monitoring energy consumption at the process level. According to its developers, Noureddine and Rajan, the tool measures the energy consumption of specific hardware components such as CPU, RAM, and HDD. However, both Jalen and Jolinar use an old Intel's energy module that is not supported by the Linux kernel versions 3.10 and above unless Intel `p_state` is disabled. [23]

A coarse-grained tool for monitoring process-level energy consumption is PowerAPI.[24] PowerAPI is a Scala-based middleware that implements an API for monitoring applications at real-time [23]. It estimates the energy consumption of various hardware components (CPU, RAM, HDD, etc.). Noureddine et al. [109] evaluated its accuracy against the powerspy2 power analyser and showed a low median error rate (*i.e.*, 0.5–3%). A weakness of PowerAPI is that it expects time duration from a user to collect energy measurements. Likely worst-case scenarios here are (1) over-collection of measurements (when an application elapses and the tool still measures the idle time) and (2) the incomplete collection of measurements

[22] https://github.com/adelnoureddine/jolinar

[23] https://github.com/stefanos1316/Proof_for_Survey/blob/master/Emails%20from%20Nourredine%20Adel.txt

[24] https://github.com/Spirals-Team/powerapi

(when an application is still running but the tool's given duration time is too short).

A noteworthy fact is that both Jalen and Jolinar have PowerAPI as their core component for extracting energy measurements. However, PowerAPI exposes energy measurements of an application at the system process-level, whereas Jalen and Jolinar correlate the collected energy-related information with Java applications. Compared to the tools proposed by Pandruvada, Liu et al., Pantels et al., and Pantels, the tools above are estimating energy and power consumption through instrumentation profiling, while RAPL utilises performance counters. Moreover, as shown in this section most of the tools lack interoperability which can be a hardware or software limitation.

Virtual Machines Joulemeter, a tool introduced by Liu et al. [91], fetches energy measurements from VMs, servers, desktops, laptops, and specific applications by tracking resource usage from various hardware components such as CPU, RAM, HDD, and screen. Its model estimates energy consumption by utilising the VM's resource tracing, which in turn obtains information through the performance counters. Joulemeter is not limited to energy consumption measurements; additionally, it offers the feature of per-VM power capping, sleep, and remote wake-up control management procedures that significantly lessen the energy consumption of a server. However, Joulemeter is not supported by an OS newer than Windows 7.[25]

Bohra and Chaudhary [21] presented VMeter, a VM power modeling method for estimating energy consumption of various components such as CPU, cache, DRAM, and HDD. The tool monitors regularly the system's resource usage and calculates energy consumption by employing a power model, which has minimal overhead on the system's total energy consumption (*i.e.*, 0.012%). To extract resource usage information from a VM, VMeter utilises the performance counters and a disk monitoring tool.

BitWatts[26] is a middleware solution that calculates the energy consumption of an application inside a VM via an energy estimation model [38]. It is an extension of the PowerAPI toolkit [23] and is designed to collect energy measurements from modern and complex microprocessors that support multi-cores, hyper-threading, DVFS, and dynamic overclocking. A strong feature of BitWatts is the collection and aggregation of energy measurements from multiple processes that are located in a distributed environment.

BitWatts energy model was compared viz-a-viz PowerSpy,[27] a Bluetooth energy meter and RAPL. As an outcome, BitWatts measurements were in the scale of 2% median error rate, which is the lowest against VMeter and Joulemeter. Also, compared to VMeter and Joulemeter, BitWatts can be used to analyse the energy consumption of more complex environments such as IoT infrastructure and data centers.

Smart-Phones Measuring the energy consumption of mobile devices is done in various ways. Zhang et al. [178] introduced PowerBooter, an on-line energy estimation model, which calculates energy consumption by combining measurements from battery's voltage sensors and discharge rate. Thus, PowerBooter estimates energy consumption without utilising external hardware power meter. By combining PowerBooter and PowerTutor,[28] the authors

[25] https://social.microsoft.com/Forums/en-US/home?forum=joulemeter
[26] https://github.com/mcolmant/powerapi/tree/BitWatts
[27] http://www.alciom.com/en/products/ powerspy2-en-gb-2.html
[28] https://github.com/msg555/PowerTutor

obtained energy measurements based on the activity of various hardware components such as CPU, LCD/OLED, GPS, Wi-Fi, and cellular network components.

Similarly, Chen and Zong [34] developed the Android Energy Profiler (AEP), a tool that correlates process resource usage activities with voltage and current information. The voltage and current information is generated through a smart-phone's built-in voltage sensor and is used by AEP to estimate energy consumption. AEP is not limited to energy consumption measurements; it can also provide run-time performance results. However, in contrast to PowerTutor, AEP provides energy measurements only for the CPU and main memory. A pitfall of AEP is a run-time performance degradation of around 25%, that incurs from the data collection process.

Another tool to obtain energy measurements from smart-phones is GreenOracle [36]. GreenOralce is an energy estimation model which, after being trained with a variety of Android applications, it can obtain energy measurements for any mobile application. The introduced model, once trained, is usable by application developers to retrieve energy measurements without the need of an external instrumentation tool. To estimate energy consumption, GreenOracle uses dynamic tracing on system calls (*i.e.* strace) and CPU utilisation.

Profiling Energy Tool for Android (*PETrA*), is a software-based energy profiling tool [47]. A feature of the tool is the fine-grained energy measurements that are obtained at the method level. Additionally, compared to prior work, *PETrA* does not require calibration. *PETrA*'s precision was validated on 54 Android applications and compared against a hardware-based energy consumption toolkit, the Monsoon. The outcome shows that *PETrA*'s energy measurements deviate little from these of Monsoon (see Table 2.13).

In contrast to all the preceding tools, PowerBooter offers energy measurements for a large number of smart-phone components. However, in contrast to PowerBooter and *PETrA*, GreenOralce suffers from high median error rate (see Table 2.13). Compared to the above, GreenOracle has a sophisticated energy estimation model that, once trained, offers energy measurements for various Android applications.

Table 2.14: Hardware-based Monitoring Tools Related-Work Information

Test-bed's Name	Target Platform	Measurement Type	Sampling Rate (in sec.)	Median Error Rate	Source
AtomLEAP	Linux	Energy	1	–	Peterson et al. [130]
SEFLab	Windows	Power, Energy	1	1%	Ferreira et al. [53]
GreenMine	Android	Power	1	insignificant	Hindle et al. [69]

2.5.2.2 Hardware Energy Monitoring Tools

Hardware energy analysers or sensors can also retrieve energy consumption measurements from a computer system. However, the disaggregation of the coarse-grained energy measurements into software or hardware components is a demanding task. The hardware energy monitoring tools are typically no-stand-alone tools that require additional hardware components such as an external device to obtain power or energy measurements. In this section, we describe some hardware energy monitoring tools for workstations, servers, and smart-phones.

Workstation and Server Monitoring Tools Software Energy Footprint Lab (SEFLab) aims to capture the energy consumption measurements of a computer system and map them to its hardware components (*i.e.*, CPU, RAM, HDD) [53]. Atom Low-Energy Aware Platform (LEAP) is a test-bed that is capable of measuring the energy consumption of small code segments in the kernel and userspace [130]. Both tools make use of the Data AcQuisition device (DAQ), an external energy profiling tool that obtains coarse-grained measurements from a computer system. After obtaining the energy consumption of an application, both tools are trying to correlate the retrieve measurements with time-stamps and system resource usage. Moreover, SEFLab uses Joulemeter [91] to compare its accuracy against the DAQ energy profiler. The major distinction between the two approaches is that Atom LEAP tries to map the collected energy consumption of an application with software components, while SEFLab maps them with various hardware components. However, to utilise one of the above approaches, a number of tools and set up is required.

Smart-phone Monitoring Tools *GreenMiner* is an experimental platform introduced by Hindle et al. [69], which retrieves the energy consumption of a smart-phone through scheduled tests. All the tests are scheduled by a web service, while a Raspberry Pi[29] is responsible for launching test scripts on a smart-phone by using the Android Debug Bridge interface. Once the tests are launched, an INA219[30] chip is used to record the smart-phone's current and voltage measurements. Afterwards, an Arduino[31] device is responsible for fetching, calculating, and storing all power measurements from the INA219 chip. Subsequently, the Raspberry Pi collects the energy consumption measurements and meta-data from the Arduino device and forwards them to a server where a web service aggregates, analyses, and stores the experiment's data. Additionally, *GreenMiner* acts as a continuous integration tool for running tests and comparing the energy consumption of different repositories. Through data aggregation and analysis, it provides insights that can guide developers in determining energy consumption change through the evolution of source code at particular product versions. Similarly to the workstation and server monitoring tools presented in the previous paragraph, *GreenMiner* also requires a number of tools and a set up to estimate an application's energy consumption.

2.6 Maintenance

Maintenance is the process of enhancing or fixing errors in software after its deployment. For the maintenance phase, we discuss techniques and tools for refactoring source code aiming to demonstrate energy savings. In the context of SDLC for energy efficiency, refactoring is the practice that aims to optimise the energy consumption of applications through code-level modifications without altering the underlying code structure. In this line, this is also used for source code analysis during software development to detect energy hot-spots or bugs (as discussed in Section 2.4.3). In this section, we present refactoring techniques applied after the deployment phase (*i.e.*, during the maintenance phase).

[29] `https://www.raspberrypi.org/`
[30] `http://www.ti.com/product/INA219`
[31] `https://www.arduino.cc/`

Table 2.15: Refactoring Energy and Run-Time Impact

Refactoring Techniques	Implications		Cases	Source
	Energy	Run-Time		
Dead Local Variable, Non Short Circuit,[a] Parameter By Value, Repeated Conditionals, Self Assignment Variable	$<$ 1 avg.	Energy Code Smells are not affecting execution time	Authors' Apps [e]	Morisio et al. [102]
Convert Local Variable to Field,[b] Extract Local Variable,[c] Extract Method, Introduce Indirection,[d] In line Method,[g] Introduce Parameter Object[h]	-7.5 to 4.54	There is no correlation between execution time and energy consumption for JVM 6 and 7	Selected Apps [f]	Sahin et al. [155]
Replace Method with Method Object and Encapsulate Collection	-7.91 to 6.99	-	M.Fowler's Code Samples [i]	Park et al. [122]
Loop Unrolling, Loop Unswitching, Method Inline	6.4 to 50.21	Observed runtime performance boost	Cocos2d game engine	Li and Gallagher [86]

[a] Using "&" and "||" cause both sides of the expression to be evaluated, in contrast to "&&" and "||||"
[b] Refactoring local variables to public class fields
[c] Occurrences of the same expression can be replaced from a variable
[d] Redirecting all the method invocations to a newly created static method
[e] Use of micro-benchmarks
[f] Commons-{Beanutils, CLI, Collections, IO, Lang, Math}, Joda-Convert, Joda-Time, Sudoku
[g] In place of the method's invocation its source code is added instead
[h] Created a new class at the top level (super class)
[i] Code Samples by Fowler et al. [54]

2.6.1 Empirical Evaluation of Code Refactoring

In this section, we consider empirical evaluations of refactoring techniques and patterns for energy-performance optimisation that software practitioners may employ to reduce application energy consumption. Table 2.15 depicts some of the authors' collected results[32] concerning the relevant refactoring method's energy and run-time performance implications, and their test cases.

Fowler et al. [54] introduced the concept of code refactoring to improve understandability, maintainability, and extensibility of existing source code. To this end, Sahin et al. [155] and Park et al. [122] used the described patterns to perform empirical studies that examine how certain code refactoring patterns affect an application's energy consumption. Both authors used distinct embedded systems, parameters, and a number of smells for their empirical studies. Specifically, Sahin et al. evaluated six widely known refactoring techniques

[32]concerning consistency for the research of Park et al. [122], we added only the results of the highest and lowest energy consumption since they investigated over 63 refactoring patterns.

on two Java Virtual Machine (JVM) versions (*i.e.*, 6 and 7) over nine applications. Park et al. evaluated 63 out of the 68 refactoring techniques over code samples proposed by Fowler et al. Sahin et al. concluded that every refactoring method may increase or decrease the application's energy usage; apart from *Extend Local Variables* which always reduces energy consumption. Similarly, Park et al. found that some of the refactoring code smells may alter positively or negatively the energy consumption of applications. Particularly, from the obtained results Park et al. illustrated that 33 of the refactoring techniques lead to energy savings while the remaining 30 do not. Besides, the authors shared that the energy consumption between Java versions, in the context of the refactoring techniques, is not consistent.

Energy Code Smells is a term stated by Morisio et al. [102] relating to inefficient implementation choices that increase energy consumption. Through their experiential study, Morisio et al. aimed to determine a number of *code smells* [54] that can alter the energy, run-time performance, or even both. Hence, they performed their experiment on existing *code smells* found by CppCheck[33] and FindBugs[34] tools. To minimise the software noise from threads which may run in parallel and subsequently affect energy measurements, the authors performed their experiment on an embedded system (Waspmote V1.1).[35] By testing nine different refactoring patterns, Morisio et al. inferred that only five of those reduce the energy consumption by less than 1%, on average, (see Table 2.15) while the remaining increase the energy consumption or leave it intact. Nevertheless, the authors claim that no correlation exists between *Energy Code Smells* and *Performance Smells* apart from a single case (*i.e.*, *Mutual Exclusion OR*).

The results depicted in Table 2.15 show that Morisio et al., Sahin et al., and Park et al. elicited both gains and losses in energy consumption such as 1%, -7% to 5%, and -8% to 7%, respectively. We can observe that the results of Sahin et al. and Park et al., used quite similar refactoring patterns, do not differ much even though they used different languages to produce their test cases (*i.e.*, Java and C++, respectively). Morisio et al. also employed the refactoring patterns introduced by Fowler et al. [54]; however, they demonstrated only minor energy savings (less than 1%). Moreover, Morisio et al. showed that the *Energy Code Smells* are not affecting *Performance Smells* or vice versa. The above results suggest that the energy savings, achieved by refactoring techniques, are very low. This fact highlights that the applied refactoring techniques are mainly focused on improving code maintainability, extensibility, and understandability. Thus, they cannot offer any substantial gains regarding energy efficiency.

2.6.2 Tools for Refactoring

In this section, we discuss work that offer tools aiming to refactor the source code of an application in order to reduce its energy consumption and run-time performance.

An energy optimisation framework for Android applications has been introduced by Li and Gallagher [86] which focuses on lending energy optimisations through a set of refactoring strategies on the deployed source code. The proposed framework takes source code as an input and analyses it to retrieve energy-related data to correlate them with basic code blocks. Afterwards, the framework spots energy hot-spots of the source code and applies,

[33]https://sourceforge.net/p/cppcheck/wiki/Home/
[34]http://findbugs.sourceforge.net/
[35]http://www.libelium.com/development-v11/

either autonomously or through manual involvement, refactoring strategies (*e.g.*, *Loop Unrolling*, *Loop Unswitching*, *Method In-line*). Employing such a refactoring tool can help a user to modify existing or legacy systems source code, so as to reduce their energy consumption and increase run-time performance. The authors evaluated their tool on real-world applications and obtained energy savings ranging from 6% to 50%.

2.7 Research Opportunities

Overall, our analysis reveals that techniques and tools for achieving energy efficiency exist for each phase of the SDLC. However, for software practitioners to adopt and use existing tools and techniques, interoperability, usability, and adequate support are crucial factors. To this end, we point out a number of possible research challenges that we identified from our study. In the following paragraphs, we analyse each of these research challenges and we provide future research directions.

RC1. *Selection of configurations and parameters.*

Parallel programming is proven beneficial (for workstations and servers) regarding energy consumption by applying the appropriate configurations and parameters such as the number of threads, data size, and data locality. As expressed by Kambadur and Kim, it is possible to gain energy savings of up to 55% and run-time performance of up to 69% if application configurations and parameters are tuned effectively. Likewise, as shown in Section 2.4, the utilisation of approximate techniques can bring an energy reduction of 10–50% for imprecision tolerant applications.

An open challenge for approximate computing is to locate portions of source code that can be optimised, while for parallel computing to choose suitable parameters and configurations to reduce energy consumption. A possible approach could be a source code analyser that can label possible energy hotspots and suggest approximation techniques for the former, while it can suggest the selection of an adequate number of threads for the latter.

RC2. *Limited investigation on diverse programming languages.*

The same application developed in distinct programming languages varies concerning energy usage and run-time performance (see section 2.4). Programming languages such as C and C++ might be challenging when it comes to memory management safety and reliability; nevertheless, they do pay back the developers with lower energy consumption and, at the same time, better run-time performance.

Researchers have investigated only a small portion of the available programming languages, while a large pool of them exists. With the advent of cloud computing, mobile applications, and hyper-physical systems, it is important to appraise the energy consumption and run-time performance of different programming languages for different programming tasks.

RC3. *Appropriate data structure selection.*

As presented in Section 2.4, selecting energy-efficient data structures is crucial, because it can significantly affect the energy consumption of an application. In some instances, selecting the most efficient data structure in real-world applications, such as Google Gson, Xalan, Tomcat, and Huffman Encoder resulted in energy savings of up to 38%. However, knowing when to select particular data structures without the need of experimenting is quite an ambitious and challenging task. Existing tools such as SEEDS [93] can suggest data structures for Java applications; but, SEEDS is limited to `queue` collection interface selection and it is not yet available for public use.

A possible research direction regarding data structure selection is to identify which are more energy-efficient for selected cases, and compose this knowledge in the form of a refactoring tool that can suggest the replacement of collection implementations.

RC4. *Interoperability, usability, and precision for tooling support.*

In section 2.5, we found that most of the energy monitoring tools are oriented towards specific CPU architectures or OSs making them difficult to use across diverse computing environments. Additionally, to configure these tools it often takes a great deal of time and effort. For instance, in order to accurately retrieve energy measurements the user has to be aware of configurations such as CPU voltage at a specific frequency. Another challenging task is software energy monitoring tool evaluation. To do so, many researchers compare energy measurements fetched from hardware power analysers with the results of their proposed tools. However, it is hard to compare or correlate these measurements because many hardware power analysers have low sampling frequency and collect energy measurements in a coarse-grained manner.

Accurate and versatile software-based energy monitoring tools are of paramount importance. This will promote a wider take-up from the software developers.

Future work should also consider the evolution of software development processes by designing tools that are appropriate for agile software development. Along this lines, energy-efficient software development approaches should be tightly integrated with energy monitoring tools that continuously assess real energy consumption at the hardware level and provide feedback to the development process which can be used to fine-tune the source code for energy savings. Such knowledge is crucial and mandatory to allow software practitioners to pinpoint energy gains in ever more complex computer systems and domains such as cloud environments, data centers, and large IoT infrastructures.

Chapter 3

Methodology

The development of energy-efficient software is a wide topic as shown in Chapter 2. Therefore, to offer a fine-grained investigation and analysis, we break down our work into three different, smaller studies. First, we discuss the various programming language implications in terms of energy and run-time performance while executing tasks on different computer platforms (server, laptop, and embedded system). Afterwards, we proceed with a discussion on various remote IPC technologies energy and run-time performance implications, implemented in different programming languages and executing tasks on a client/server architecture. Finally, we discuss our research on investigating possible performance savings while dispensing security mechanisms in modern OSs. In the following sections, we discuss in detail the (1) research objectives, (2) software systems and tools, (3) research methods, and (4) limitations of our studies. Specifically, our research consists of three main parts as discussed below.

3.1 Research Objectives

In this section, we set the research objectives that we aim to answer in this thesis. Moreover, we discuss why it is important to answer these research questions. We use several prefixes to discuss the research questions presented in this thesis. In particular, we use the **PL-RQ** to discuss the programming languages research questions, the **IPC-RQ** to refer to the remote IPC technologies study, and the **SG-RQ** to discuss on security mechanisms work.

3.1.1 Programming Languages Research Questions

For the first study, we concentrate on how the programming languages impact various computer tasks. Specifically, we measure the 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 belong to.

Previous studies claim 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 [5, 145, 34]. 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:

PL-RQ1. *Which programming languages are the most EDP-efficient and inefficient for particular tasks?*—Our objective 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.

PL-RQ2. *Which types of programming languages are, on average, more EDP-efficient and inefficient for each of the platforms?*—Our goal is to evaluate the efficiency of different programming language 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.

PL-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 distinct compute platforms affect the EDP of tasks, developed in programming languages, in a different way.

3.1.2 IPC Research Questions

Works carried out in the context of the energy consumption for IPC technologies are limited to Java implementations executed on smart phones [33, 67] or embedded systems [111]. These have shown that Rest implementations offer the best results for these particular devices. However, IPC technologies are often being used heavily in other IT-related contexts such as data-centers and IoT. To this end, we investigate whether the same pattern exists for computer systems equipped with Intel and ARM processors by taking into account seven different programming languages and the platforms' system calls and resource usage.

Hereby, we define our research questions as follows:

IPC-RQ1. *Which IPC technology implementation offers the most energy and run-time performance-efficient results?*—Our objective here is to identify the implications that each IPC technology has on the energy consumption and run-time performance for the selected programming languages. This can help practitioners select among the IPC implementations that offer the most energy- and run-time performance-efficient solutions.

IPC-RQ2. *What are the reasons that make certain IPC technologies more energy and run-time performance-efficient?*—Here, we investigate under the hood how each of the selected IPC technologies works, by examining the implementations' system calls. This can show us which are the system calls that are mostly used by certain applications and for most of their execution time, thus possibly contributing to increased energy consumption and lower run-time performance.

IPC-RQ3. *Is the energy consumption of the IPC technologies proportional to their run-time performance or resource usage?*—For this research question, we investigate a conflicting view among researchers: that energy consumption is proportional to run-time performance. We address this issue only in the context of IPC technologies. Moreover, we investigate if energy consumption is in proportion with resource usage such as maximum memory usage, number of page faults, and context switches. This can act as an indication to warn developers regarding the energy consumption of their applications and libraries.

We answer the above research questions by using a number of metrics. For **IPC-RQ1**, we use energy consumption, that is, the product of the total power consumed and time taken for an executed task. In addition, we collect run-time performance measurements, that is the execution time of a task. For **IPC-RQ2**, we collect system calls, which are API calls of an application that requests services from the user to kernel space. Finally, we use the measurements of **IPC-RQ1**, the maximum memory set size, context-switches, and page faults to answer **IPC-RQ3**. The maximum memory set size indicates the total memory reserved for a task, the number of context-switches shows the number of times a system requested services from the kernel and the associated overhead, while page faults indicate memory pressure during execution time.

3.1.3 Safeguards Research Questions

For our final study, we seek to examine the energy consumption and run-time performance of diverse security measures enabled on computer systems. We identify the following questions:

SG-RQ1. *What are the energy and run-time performance implications of the safeguards on a computer system?*—Corresponding answers may lead to a more energy-aware system usage for single tenants of server systems. That is, if administrators or users turn off highly taxing security mechanisms in a secure environment, then they could reduce energy consumption and increase run-time performance. In addition, we can identify which security mechanisms introduce noticeable performance overheads. Such findings may act as a guide for potential refactorings in the source code.

SG-RQ2. *How do security mechanisms affect the energy consumption and the run-time performance of diverse applications and utilities?*—In this context, we examine various applications such as database systems, servers, memory-intensive, compute-intensive, graphic-intensive applications, and so on. Consequently, we can inform developers and administrators to disable security mechanisms when they use relevant applications in an isolated environment.

SG-RQ3. *Is the energy consumption of the examined security mechanisms proportional to their run-time performance?*—Note that recent related studies provide conflicting results. We attempt to shed light on this matter and through our findings, help developers understand the energy requirements of their applications by observing their performance.

3.2 Software Systems and Tools

The aim of this section is to elaborate on the tools, computer systems, data-sets, programming languages, and utilities adopted to perform our experiments. We discuss the studies on (1) programming languages, (2) IPC technologies, and (3) security measures, respectively.

3.2.1 Programming Languages Subject Systems

Data Set. We made use of the Rosetta Code Repository, a publicly available repository for programming chrestomathy. [1] Rosetta Code offers 868 tasks, 204 draft tasks, and has implementations in 675 programming languages [101]. 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 [7] 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 website [166] 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

[1]word stemming from the Ancient Greek "χρηστομάθεια" that means useful for learning

Table 3.1: Programming Languages, Compiler and Interpreter Versions, and Run-Time Performance Optimisation Flags

Categories	Programming Languages	Compilers & Interpreters Embedded	Laptop	Server	Optimisation 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	-optimise+
	VB.NET	4.6.2	4.6.2	4.6.2	-optimise+
	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	–

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,[2] and (3) have their own Wikipedia page [167]. Taking Tiobe October's 2017 index rating, we 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 [166]. The selected programming language categories (compiled, interpreted, and semi-compiled), along with their names, compilers and interpreters' versions, and compile-time performance optimisation flags are shown in Table 3.1. We selected the highest possible performance optimisation flags, since some languages (*e.g.*, Go) by default apply the highest degree of optimisations. 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 3.2. Table's 3.2 first and second columns list the selected categories and the names of the tasks, as they were found on Rosetta Code's

[2]can be used to emulate a Turing machine

official website. 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 Figures 4.1 and 4.2.

Handling the Data Set. To use the above-mentioned data set properly, 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 analyser, *i.e.*, Watts Up Pro [173], has a sampling rate of a second (lowest rate). 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 optimisations. Then, once they find out that the same function is being repeated multiple of times, they optimise 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 to 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 [57]. 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 GiB RAM) server [80], an HP EliteBook 840 G3 Notebook laptop [72], and two Raspberry Pi 3b model [146], with an Intel i7-3770, an Intel i7-6500U, and a quad-core ARM Cortex-A53 micro-processors, respectively.

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` [49]. 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 system. Finally, to extract the collected measurements from WUP's internal memory, we used an open source Linux utility-based interface available on GitHub [15].

Table 3.2: Selected Categories, Tasks, Explanation, Input Test, and HeatMap Abbreviations

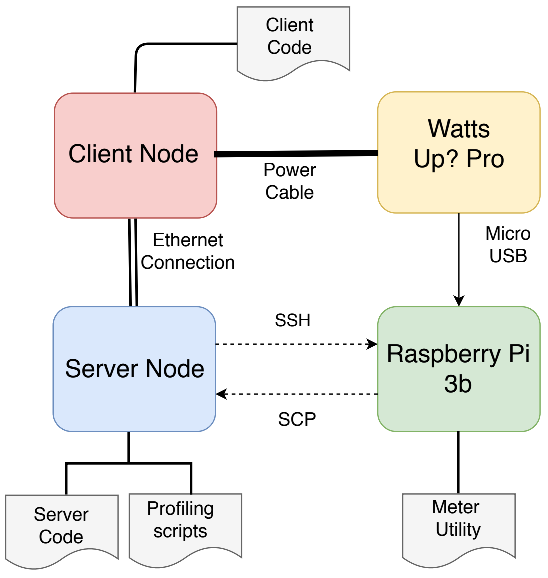
Categories	Names	Explanation	Input Test	Abbreviations
Arithmetic	*exponentiation-operator*	exponentiates integer and float	$2017^{12}, 19.88^{12}$	exp.-operat.
	numerical-integration	calculates the definite integral by using methods rectangular{left,right,midpoint}, trapezium, and Simpson's for 10^x 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.

3.2.2 IPC Subject Systems

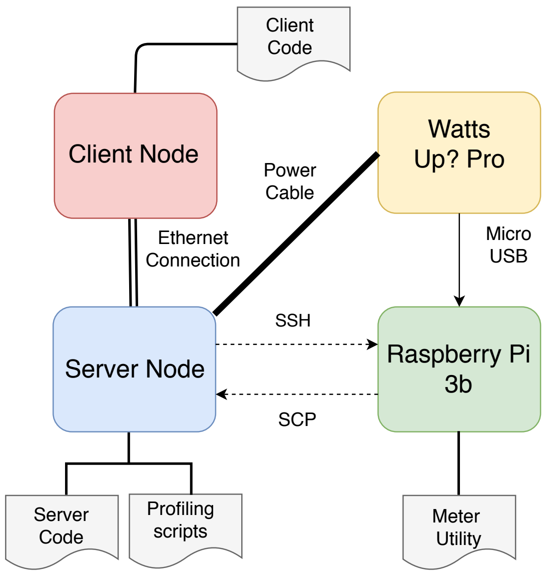
Experimental Platform: We performed two experiments to consider different contexts and environments. For the former, we used two Lenovo ThinkCentre M910t platforms [83], where one was acting as a server and the other as a client. In a similar way to the above, we utilised two Raspberry Pis 3B model (RPi), to simulate an IoT test case. In the context of this study, we will refer to the Lenovo and RPis platforms as Intel and ARM platforms, respectively. To retrieve energy consumption, we utilised an external device, WUP [173]. Also, we used an additional RPi to fetch the energy measurements, from the WUP's internal memory, in real-time with the help of a Linux-based open source utility interface [15]. We followed this approach to avoid further overhead on the server and client instances that could impact their energy consumption. To collect the run-time performance, we used the Linux `time` command to yield the *wall-time* of our implementations. We used the wall-time because WUP offers coarse-grained measurements for the whole computer platform and not only when a process is utilising the CPU. Figures 3.1a and 3.1b depict the platform connectivity between the subject systems.

Programming Languages: In order select our programming languages, we employed the GitHut.info [73] and PyPL [112] web-pages (December 2018). GitHut.info offers information concerning the popularity of various programming languages by taking into account the GitHub active repositories, total number of pushes, and so on. PyPL administers its ranking based on the frequency by which a programming language tutorial has been searched on Google every month. To this end, we picked the first six most popular and active programming languages according to the GitHut.info and PyPL statistics, which were JavaScript, Java, Python, PHP, Ruby, and C#. In addition, we covered in our selection Go, because it is one of the programming languages that earned the highest popularity the recent years, according to a Tiobe study [2].
IPC Technologies: The selected IPC technologies varied among REST, RPC and gRPC. REST is a stateless architecture style for distributed systems, widely adopted for offering world-accessible APIs. RPC is a publicly known way of causing a procedure to execute remotely and many organisations have developed APIs for it. Therefore, we selected and used the RPC and REST libraries as shown in the next paragraph. We also studied gRPC, a RPC technology developed by Google that uses Protocol buffers 3 and HTTP/2 to boost its speed and interoperability between services. We selected gRPC since many companies that are using micro-services (*e.g.*, Netflix, Cisco, CoreOS) are adopting it in their production. Also, gRPC offers library implementations in diverse programming languages making it a suitable candidate for our empirical study.

Web Frameworks: We used web frameworks to build our end-points for the server and client function; however, for this research, we do not examine their impact. To select them, we employed the HotFrameworks [55], that provides a monthly ranking on web frameworks' popularity based on the numbers of GitHub stars and StackOverFlow tagged questions. For JavaScript, we selected *Express* since the *AngularJS* started losing popularity after 2017 and *React* is mainly used to create user interfaces. Similarly, we excluded *Django* and kept *Flask* for Python. For C#, we used ASP.NET because it is the most influential web framework. Likewise, we selected Ruby on Rails and Laravel for Ruby and PHP to develop the RESTful tasks, respectively. For Ruby and PHP RPC tasks, we could not find any official implementation of



(a) Retrieving energy measurements from the client.



(b) Retrieving energy measurements from the server.

Figure 3.1: Experiment setup for Intel and ARM systems

Laravel and Ruby on Rails; therefore, we wrote the RPC tasks using the JSON-RPC 2 library. For Go, we selected its built-in packages since the HotFrameworks does not offer any in its ranking. We selected JAX-RS and JAX-WS for Java since the *Spring* is mostly used for RESTful applications. For gRPC, we utilised the latest available versions for each of the selected programming languages which are publicly available on GitHub.[3]

Table 3.3: Experimental Setup Parameters

Programming	Compilers & Interpreters		IPC Technology Packages and their Versions					
Languages	Servers	RPis	REST	Version	RPC	Version	gRPC	Version
Go	1.9.4	1.9.4	NET/HTTP	1.9.4	NET/RPC	1.9.4	gRPC-Go	1.17.0
Java	1.8.0	1.8.0	JAX-RS	2.1.0	JAX-WS	2.1.0	gRPC-Java	1.17.0
JavaScript	10.4.0	10.4.0	Express	4.16.3	Express–RPC	0.0.4	gRPC-Node	1.17.0
Python	2.7.14	2.7.14	Flask	1.0.2	Flask–RPC	1.0.2	gRPC-Python	1.17.0
PHP	7.2.12	7.2.12	Laravel	5.7.15	JSON-RPC	2.0	gRPC-PHP	1.17.0
Ruby	2.5.3p	2.5.3p	Rails	5.1.6	JSON-RPC	2.0	gRPC-Ruby	1.17.0
C#	4.8.0	4.8.0	ASP.NET	2.1.5	ASP.NET	2.1.5	gRPC-Csharp	1.17.0

Test Case: To perform our experiment, we either used existing or developed missing `HelloWorld` examples that are making use of the three IPC technologies discussed in this section. For gRPC, we could not test a PHP-written server program because currently there is none available.[4] Therefore, we performed the experiment using a JavaScript server code as suggested in the official documentation. Likewise, it was not possible to compile the C#'s gRPC source code for the ARM platform; therefore, we did not execute the C#'s gRPC for the corresponding platform. The reason we selected a straightforward scenario is that we were mainly concerned with examining only the cost of different remote IPC technologies when

[3] https://github.com/grpc
[4] https://grpc.io/docs/tutorials/basic/php.html

Table 3.4: The *Phoronix* [163] Benchmark Suite Used in Our Measurements

Category	Benchmark Suites
Audio Encoding	encode-mp3, encode-flac
Video Encode/Decode	dav1d, svt-av1, svt-hevc, svt-vp9, vpxenc, x264, x265, ffmpeg
Code Compilation	build-php, build-linux-kernel, build-gcc, build-gdb, build-llvm, build2
File Compression	compress-p7zip, compress-bzip2, compress-zstd, compress-xz, lzbench
Database Suite	sqlite, redis, rocksdb, cassandra, mcperf, pymongo-insert
CPU Massive	aircrack-ng, apache, blogbench, brl-cad, byte, cloverleaf, cpp-perf-bench, crafty, dacapo-bench, ebizzy, embree, fhourstones, glibc-bench, gmpbench, himeno, hint, hmmer, hpcg, javascimark2, m-queens, minion, nero2d, nginx, node-express-loadtest, numenta-nab, phpbench, primesieve, pybench, pyperformance, rodinia, rust-prime, scimark2, stockfish, swet, sysbench, sudokut, tensorflow, xsbench, sunflow, bork, java-jmh, renaissance, tiobench, openssl, blake2s, john-the-ripper, botan, octave-bench, oidn
Disk Suite	fs-mark, iozone, dbench, postmark, aio-stress
Kernel	schbench, ctx-clock, stress-ng, osbench
Machine Learning	rbenchmark, numpy, scikit-learn, mkl-dnn
Memory Suite	ramspeed, stream, t-test1, cachebench, tinymembench, mbw
Networking Suite	iperf, network-loopback
Imaging	graphics-magick, inkscape, rawtherapee, tjbench, dcraw, darktable, rsvg, gegl
Renderers	tungsten, ospray, aobench, c-ray, povray, smallpt, ttsiod-renderer, indigobench, rays1bench, j2bench, qgears, jxrendermark
Desktop Graphics	xonotic, openarena, tesseract, paraview, unigine-valley, unigine-heaven, nexuiz, glmark2

they are invoking remote procedures. More specifically, the client makes some remote procedure invocations towards the server's `HelloWorld` function, and the server replies with a *"Hello World"* message. To this end, Table 3.3 illustrates the selected programming languages, their compiler and interpreter versions, and the used IPC technology packages and versions for each programming language implementation.

Execution Scripts: To control our experiment's workflow, we wrote around 1,800 lines of Unix shell scripts to automate the execution, data collection, and results plotting process. All scripts are publicly available on GitHub.[5] For executing the tasks, we included the `basic` function, that makes the remote procedure call, in a loop of 20,000 and 5,000 iterations for the Intel and ARM computer systems, respectively. We took such an action to force the execution time of a task to take over a second. We did this because the WUP performs `power sampling` and reports the collected energy measurements, on a per second basis.

3.2.3 Safeguards Subject Systems

Computer Platform: We used a *Lenovo ThinkCentre M910t* equipped with an *Intel i7700 Kaby Lake* processor, running Ubuntu 18.04.1 OS (kernel version: 5.3.0-40-generic). We have selected Ubuntu because currently, it is the most popular OS used by servers worldwide [4].

[5] `https://github.com/stefanos1316/Rest_and_RPC_research/scripts`

Table 3.5: Hardware Platform

System	Kaby Lake (x86)
Microarchitecture	Kaby Lake
Processor/Soc	Core i7-7700
Cores $\times$ threads	4 $\times$ 2
Base Frequency	3.6 GHz
Max Frequency	4.2 GHz
Cache line size	64 B
L1-D/L1-I Cache	4 x 32 KiB 8-way
L2 cache	4 x 256 KiB 4-way
L3 cache	8 MB 16-way
I-TBL	4-KByte pages, 8-way
D-TBL	1-GB pages, 4-way
L2-TBl	1-MB, 4-way
RAM	16 GB UDIMM, DDR4 2400

Table 3.5, details the characteristics of our target platform.

To collect our energy measurements, we used the `perf`[6] profile monitoring tool that utilises the Running Average Power Limit (RAPL)[7] framework. RAPL uses hardware performance counters to estimate the energy consumption of the CPU cores, package,[8] and main memory. In our experiments, we used the main memory and package readings, because the package offers measurements for the cores, cache, and embedded GPU. Note that RAPL is a well-established utility that has already been used in related work [43, 126, 103] and its accuracy was validated by various studies [77, 45, 118, 76] and offers high sample interval (a reading per 1 msec).

Benchmarks: To observe how security mechanisms affect diverse settings and computer components, we used Phoronix [163], a well-established benchmark suite that is composed of various open-source benchmarks. We used various benchmark suites such as audio and video encoding, compilation, compression, computational, cryptography, databases, disk, memory, games and more. Some benchmarks required the creation of account in order to download specific packages or libraries. Therefore, we excluded them from our data-set, because they do not allow us to automate the installation of our data-set. Table 3.4 shows the different categories as defined in the official site of Phoronix with the corresponding benchmark suites. Note that some benchmarks can be found in more than one category. We have selected Phoronix because it covers a large set of tasks and can illustrate the whole picture regarding the impact of security mechanisms.

Phoronix executes a large number of benchmark suites to verify a computer system's performance and throughput. Therefore, we developed a number of scripts to (1) download and install all 128 benchmark suites, (2) install all system dependencies and packages to execute the benchmarks, (3) generate input data for the benchmarks, (4) execute benchmark suites

[6]https://perf.wiki.kernel.org/index.php/Main_Page
[7]https://01.org/rapl-power-meter
[8]cores and uncore components

tasks, and (5) collect energy, run-time performance, and resource usage measurements.

Some of the benchmarks are measuring throughput and elapsing after a specific period of time. Such a fact could not allow us to compare the energy consumption needed for a particular number of operations since a benchmark could consume more energy, after disabling a security mechanism, but have a high throughput. To this end, we modified the source code of those benchmarks to perform a specific number of operations and then terminate. We took such an action to compare the energy needed from a task to perform a specific number of operations with the security mechanism enabled to the energy consumption of the same tasks with the security measures disabled. To examine memory zeroing, we used the `redis-benchmark` [3] utility because of its memory allocation flexibility. Focusing on HTTPS we used the client / server built-in libraries of Node.js without changing any of the default cipher suites. Note that we did not focus on the cipher suites because Potlapally et al. [139] have already studied their energy and performance implications.

Scenarios: To produce our results, we ran the benchmarks in different setups. First, we executed all the benchmarks included in Table 3.4 with all CPU vulnerability patches enabled. Then, we executed the benchmarks by only disabling either (1) Meltdown, (2) Spectre, (3) and MDS together with TAA. Note that we disabled both MDS and TAA because the latest kernel updates do not allow users to disable MDS or TAA separately. Finally, we disabled all CPU vulnerability patches and ran the benchmarks again. We used the system's `Grub` bootloader to configure all the setups for the corresponding CPU vulnerability patches.

The GCC toolkit offers a vast set of security flags. However, we narrowed our scope and examined the ones recommended in the RetHat's Developers site. [9] Also, we choose GCC safeguards that are (1) compatible to our processor, (2) supported by the corresponding GCC version (7.5–latest for the Ubuntu OS at the time of the experiment), and (3) available for both C and C++ code. Specifically, we investigate the following safeguards (1) `-fno-stack-protector` that stops GCC from emitting extra code to check for buffer overflows, such as stack smashing attacks, (2) `-z execstack` that allows instructions execution of any memory page in the stack, (3) `-no-pie` and `-fno-pic` that prevent the loading of binaries, dependencies, and code into arbitrary memory locations, (4) `-Wl,-z,norelro` that disable protection against Global Offset Table overwrite attacks, and (5) `-U_FORTIFY_SOURCE` to avoid buffer overflow detection on run-time. To this end, we run the benchmarks by enabling and disabling all the above safeguards and then enable them one at a time.

In a similar manner, we performed requests between the NodeJS client and server over HTTPS and then over HTTP. To work with HTTPS, we generated self-signed certificates.

To perform calculations related to memory zeroing, we first measured `memset` when zeroing large blocks of memory. Then, we ran four different *Redis* functions (i.e., `set`, `lpush`, `hset`, and `lrange`) and traced the system memory allocations.

When we performed our experiments we set the CPU frequency governor to the "performance" mode. We did this to avoid the dynamic voltage and frequency scaling (DVFS) power management policies. Such policies may reduce the power of embedded GPUs with reasonable performance degradation. We also avoided using governors such as "ondemand" and "powersaving" because they are more suitable for experimenting with large clusters and systems and they can introduce power saving mechanisms that can degrade the run-time performance of a program [40, 162]. Contrariwise, our experiments involved micro-

[9] https://developers.redhat.com/blog/2018/03/21/compiler-and-linker-flags-gcc/

benchmarks.

3.3 Research Method

In this section, we discuss precaution and necessary measures that we took to ensure the correctness of our results. Moreover, we describe an experiment that we performed to ensure the accuracy of our measurements obtained from WUP against an oscilloscope.

3.3.1 Programming Languages Measurements Collection

Our calculations are based on Energy-Delay Product (EDP), an equation introduced by Horowitz et. al [71] and applied as a weighted function by Cameron et. al [28]. The EDP is defined as follows:

$$EDP = E \times T^{w} \tag{3.1}$$

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 address **PL-RQ1** and **PL-RQ3**, and we used all three weights (to see how performance affects the average EDP) to answer **PL-RQ2** in Section 4.1.

We chose EDP among other energy metrics (*e.g.*, Greenup, Speedup, and Powerup [6]) because normalised EDP can offer fair comparisons among programming implementations, which run on different execution platforms. Contrary to Abdulsalam's et al. study [6], we do not evaluate the efficiency of optimisations, where performance and energy is measured separately.

Before collecting our measurements, at the computer nodes' 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 a minute to avoid adding overhead to our results. As it is suggested by Hindle [69], 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` [150] for the server and laptop platforms and `vcgencmd` [51] 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 the system reached a stable condition.

Upon reaching a stable condition, our script initiated the execution of the tasks. Before the execution of a task the computer node sent an SSH command to the energy monitoring system device to start retrieving energy measurements from the WUP's internal memory.

By the end of the whole use case, the energy monitoring system had collected and sent the energy measurements to the computer node through the SCP utility. When the results were received by the computer node, a plotting script depicted them in the form of heatmaps.

3.3.2 IPC Measurements Collection

To perform our experiment, we followed the experimental approach described below.

- We started our computer systems and stopped unnecessary background processes according to suggestions by Hindle [69] and waited for our system to reach a *stable condition i.e.*, where the energy consumption was idle (23 and 1.5 Joules for Intel and ARM processor, respectively).
- Then, we started our execution script that initiated, through SSH, the (1) server instance to receive requests, (2) the RPi to retrieve energy consumption measurements from WUP's internal memory, and (3) the client to perform the tests and collect execution time.
- When any IPC implementation finished with its execution, we left a small window of a minute, using the Linux `sleep` command, to avoid *tail power states* [22] and to allow our device to reach the stable condition before executing the next implementation.
- Once the whole experiment was done, all the data from the nodes (*i.e.*, the client and RPi) were transferred to the server, using the SCP utility, to sanitise the energy measurements from the idle time and plot graphs.

Because we had only a single WUP at our disposal, we executed the above experiment twice, once for measuring the Intel's server instance energy consumption and once for measuring the client's consumption. Afterwards, we performed the same experiment for the ARM platforms. To minimise measurement noise, we performed each experiment 50 times and computed descriptive statistics such as the standard deviation, mean, and median values of energy consumption and run-time performance. By plotting histograms[10] for each of the programming language IPC implementations, we observed that our measurements exhibit a lot of variance. To this end, we decided to retrieve and depict as results the median values (shown in Chapter 4).

3.3.3 Accuracy of obtained Results

According to Saborido et al. [152], a low sampling rate might miss energy consumption measurements appearing for a short period (energy spikes). Therefore, using a device such as the WUP could output imprecise measurements.

To evaluate if the above statement is true and obtain more samples, we performed experiments on the RPis with some JavaScript and Go tasks, that exhibited low energy and run-time performance. To perform these experiments, we had at our disposal: (1) an oscilloscope[11] connected with a current probe[12] and (2) a multimeter.[13] We first connected

[10]https://github.com/stefanos1316/Rest_and_RPC_research/tree/master/arm/statistics_client
[11]`https://www.picotech.com/oscilloscope/3000`
[12]`http://www.all-sun.com/EN/d.aspx?pht=1066`
[13]`https://www.uni-t.cz/en/p/multimeter-uni-t-ut139c`

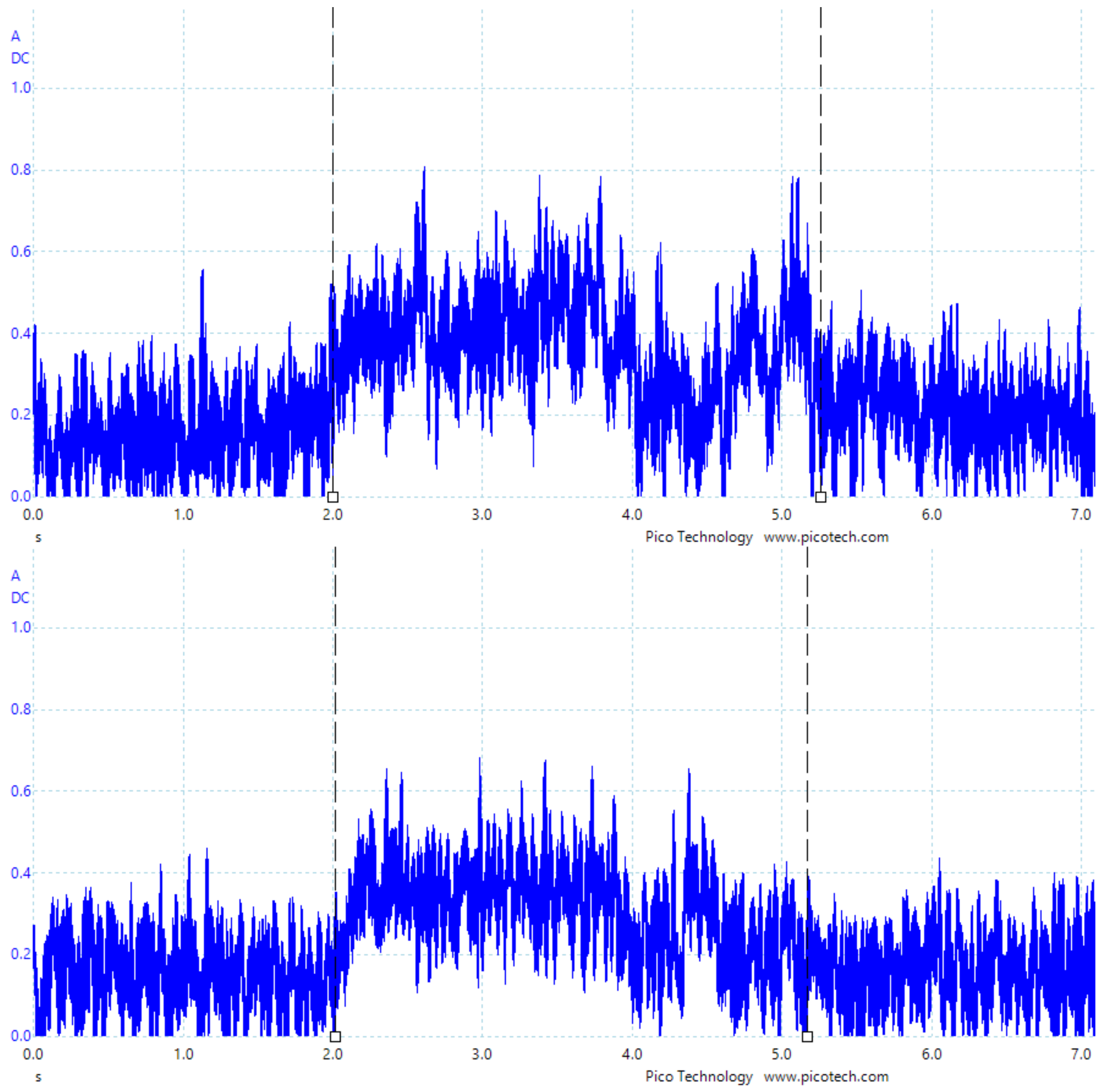


Figure 3.2: Client (top) and server (bottom) waveforms of gRPC task implemented in JavaScript running on ARM platforms

Table 3.6: Comparison between Watts Up? Pro and Oscilloscopes Measurements

IPC	Language	Device	Accuracy	Average Power (in Watts)	
				Server	Client
gRPC	JavaScript	WUP	± 5%	1.94	2.04
		Oscilloscope	± 6%	1.90	2.09
RPC	Go	WUP	± 5%	1.77	2.00
		Oscilloscope	± 6%	1.81	2.13
REST	Go	WUP	± 5%	1.68	1.88
		Oscilloscope	± 6%	1.71	2.02

the multimeter and the current probe on an extension cable live, where a computer system was plugged in. Next, we compared the oscilloscope's current measurements (obtained as input from the current probe) against the multimeter's. We utilised a multimeter (as ground truth for current measurements) to ensure that its measurements aligned with the current probe's. We then made the appropriate settings to the oscilloscope (input voltage, scaling, and true root mean square measurements) according to the current probe's specifications. Also, we measured the average Direct Current (DC) for the tasks execution time by using the oscilloscope's `between rulers` option (see the dashed lines of Figure 3.2), which allows to obtain measurements for a specified period of time.

Figure 3.2 depicts the server and client waveforms while executing the gRPC task with a sampling rate of 200,000 units per second. The X-axis illustrates the sampling duration in seconds, while the Y-axis shows the current measurements in Amperes. We present the results obtained from the waveforms in Table 3.6. Specifically, we show the related task, the language it was implemented with, the devices used to measure its energy consumption and their accuracy, and the average power consumption for the client and server implementations. For the oscilloscope, we obtained our measurements in current and multiplied them with 5V (RPi's power supply) to obtain power consumption. The results indicate that the WUP's measurements fall slightly shorter than the oscilloscope's (*i.e.*, 2–7.1%). This happens because the oscilloscope exhibits measurements with the precision of millisecond for the tasks execution (*e.g.*, 4.432 seconds), while the WUP captures per second measurements (*e.g*, 4 seconds). Therefore, if a task elapses some milliseconds after the WUP reports energy consumption then the energy consumption until the next second will also be added in the average DC, that is partially idle energy consumption of the RPi.

Algorithm 1: Data Collection

```
Input   : $taskList$ a list of all Phoronix tasks
Output: $results$ including energy consumption and run-time performance
1  $stopBackgroundProcesses()$
2  foreach security mechanism do
3      Function executeTasks (taskList)
4          foreach $T_i$ in $taskList$ do
5              repeat 5 times
6                  $results[\,] = executeAndMeasure(T_i)$
7              end
8              $sleep(60)$
9          end
10     end
11     $allResults[\,] = results[\,]$
12 end
13 return $allResults[\,]$
```

3.3.4 Safeguards Measurements Collection

Algorithm 1 presents the steps we performed to collect our results. First, we stopped all unnecessary background processes (as suggested in other similar studies [69]) to let our system reach a *stable condition* (*i.e.*, where the energy consumption is idle). Then, we started executing tasks to obtain their energy and run-time performance. After the end of each task execution, we let the computer platform idle for one minute (using the `sleep` command). In this manner, we were able to avoid *tail power states* [22] and allow the system reach a stable condition again (idle energy consumption) before executing the next task.

To reduce noise in our measurements, we executed the above steps five times for each scenario by using the `-r 5` command-line argument that outputs the average value of the five executions and the percentage of difference among the five mean values. To this end, we used the mean values to depict our results, because the difference among the five executions for each task ranged between 0% to 6%, with the majority of them being below 1%. The above fact indicates that a task's measurements do not diverge much among different executions. Collecting performance measurements for a single CPU vulnerability patch in the aforementioned manner, took more than three days. Likewise, for each of the GCC safeguards, the experiment took more than two days. Finally, the measurement collection for memory zeroing and HTTPS communications, took less than an hour.

3.4 Threats to Validity

Our studies have internal, external, and reliability 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 for each one of our studies. **External validity** defines to the extent to which the results of our study can be generalised to other programming implementations, applications, or computer platforms. **Reliability validity** offers links to our scripts and data in order to replicate our results.

3.4.1 Programming Languages Limitations

Internal validity. First, we used cabled instead of wireless connection, because the former is more energy-efficient than the latter [16]. 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 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. 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) [156]. 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 generalise our arguments for other programming languages and computer platforms.

Reliability validity. We host all the used scripts and the obtained data in our GitHub repository [14] to allow the replications of our study.

3.4.2 IPC Limitations

Internal validity. Having full control over our operating systems' workload and background operations is hard, because, at any time, different daemons may operate. Also, when a task is executing and enters in a waiting state (*e.g.*, due to an I/O operation) the WUP will still record the energy consumption of our computer platform. This could affect our calculations, too.

For the Java's gRPC tasks, we were not able to compile its native extensions on the embedded systems. However, because of the JVM, we were able to execute the task of the Intel platform on the embedded systems without the need to compile it. Therefore, we are not aware to which extent this fact can affect our results.

External validity. For our study, we used a simple micro-benchmark to evaluate the energy implications of IPC technologies. As discussed by Sahin et al., this may not always depict the whole image regarding the energy implications in a real-world application [156].

Finally, we evaluate the energy consumption and run-time performance of three IPC technology tasks written in seven programming languages, and running on server platforms and embedded systems. Hence, we are not sure to which extent we can generalise our arguments for other programming languages or platforms.

Reliability validity. The scripts/data are open source for the replication of the study on out GitHub repository.[15]

3.4.3 Safeguards Limitations

A threat to the **internal validity** of our experiment may involve running background processes and daemons while we collect data. Indeed, having total control over all background processes or daemons of a system is difficult. Therefore, this could have affected our measurements to some extent. Also, obtaining energy measurements via RAPL, outputs fine-grained metric results of specific computer components such as CPU, RAM, and package (cores and uncore components). Therefore, we were unable to measure the impact on components such as NIC or HDD. As a result, we do not have a holistic view of the energy consumption of the network or disk benchmark suites.

[14] `https://github.com/stefanos1316/Rosetta_Code_Research_MSR`
[15] `https://github.com/stefanos1316/Rest_and_RPC_research`

As we mentioned earlier, we experimented only on one platform running a specific operating system which is a threat to the **external validity** of our experiment. Hence, we cannot argue that the same energy or run-time performance implications will persist on other platforms or operating systems. In addition, we only focused on micro-benchmarks and not on large applications. Note that, Sahin et al. [156] have pointed out that energy consumption is different in the case of real-world applications than the one observed in micro-benchmarks.

For **reliability validity**, we publicly offer our scripts to install any requirement, library, or dependency needed by our benchmarks though an Ansible file.[16] Moreover, the data of our experiment can be found on the same link.

[16]https://github.com/stefanos1316/phoronixDataSet.git

Chapter 4

Results and Analysis

In this chapter, we show the collected results of our experiments and provide a discussion for each one of them. We discuss our results regarding the energy consumption and run-time performance of (1) programming languages, (2) remote IPC technologies, and (3) security mechanisms.

4.1 Programming Languages Results Analysis

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.

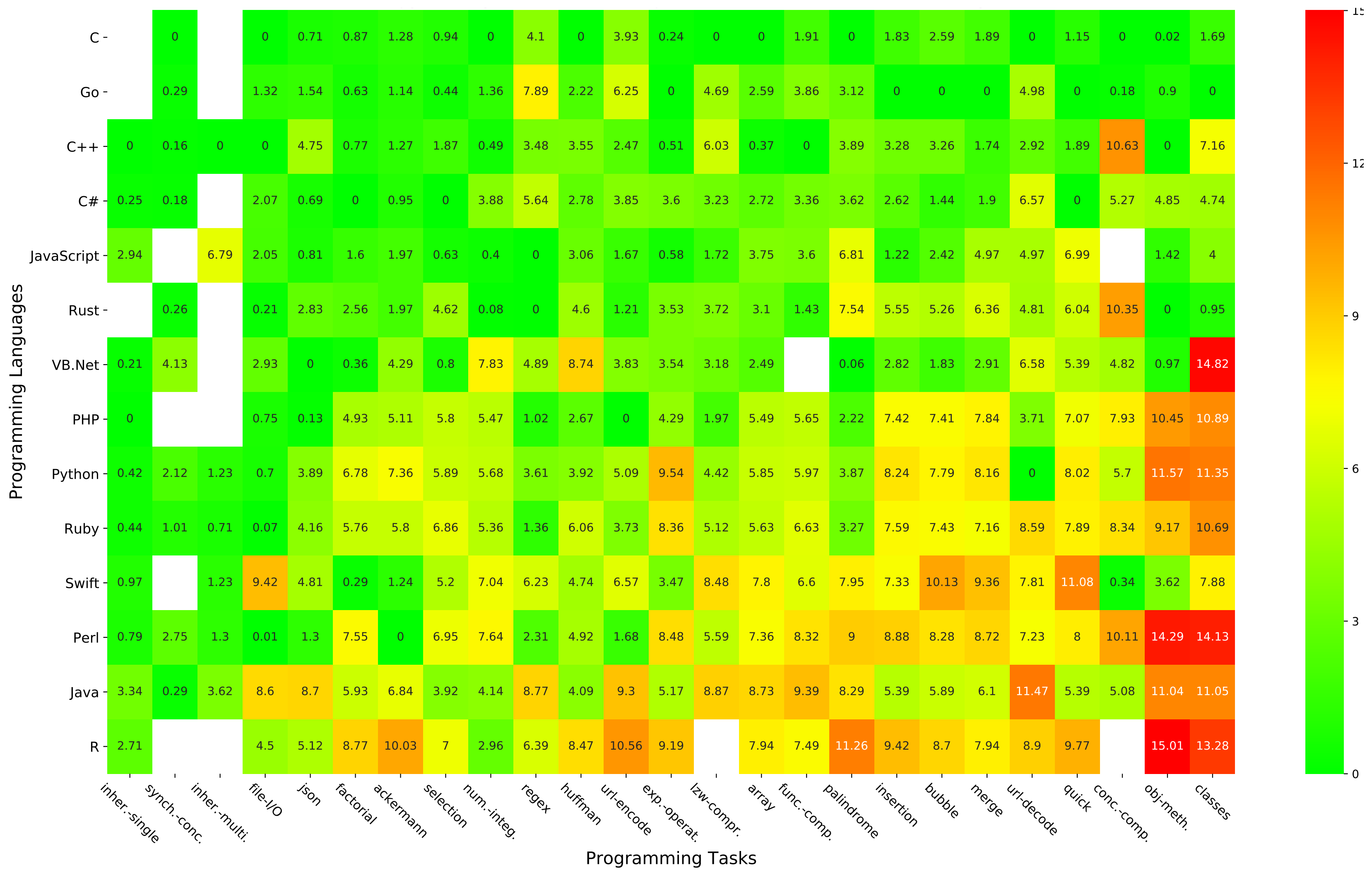
Before plotting the heatmaps, we ranked the tasks based on their EDP value. For each particular task, we took the lowest EDP value and we used it to normalise the measurements of similar tasks. Some of the resulting values varied substantially among the selected platforms. Therefore, we used the base ten logarithm to reduce the influence of extreme values or outliers in the data without removing them from the data set. To perform the above action, we developed a script that applied the base ten logarithm on our raw (initial) EDP values.

Figure 4.1 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 4.1, correspond to the programming languages that achieved the lowest score of EDP for particular tasks, as $0 = \log 1$ after normalising. The green color in the heatmap 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 heatmaps. The remaining heatmaps can be found in Appendix A.

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

To answer this research question, initially, we discuss the results of each platform by grouping them into categories. We identify in which categories, in Table 3.2, specific programming



	inher.-single	synch.-conc.	inher.-multi.	file-I/O	json	factorial	ackermann	selection	num.-integ.	regex	huffman	url-encode	exp.-operat.	lzw-compr.	array	func.-comp.	palindrome	insertion	bubble	merge	url-decode	quick	conc.-comp.	obj-meth.	classes
C		0		0	0.71	0.87	1.28	0.94	0	4.1	0	3.93	0.24	0	0	1.91	0	1.83	2.59	1.89	0	1.15	0	0.02	1.69
Go		0.29		1.32	1.54	0.63	1.14	0.44	1.36	7.89	2.22	6.25	0	4.69	2.59	3.86	3.12	0	0	0	4.98	0	0.18	0.9	0
C++	0	0.16	0	0	4.75	0.77	1.27	1.87	0.49	3.48	3.55	2.47	0.51	6.03	0.37	0	3.89	3.28	3.26	1.74	2.92	1.89	10.63	0	7.16
C#	0.25	0.18		2.07	0.69	0	0.95	0	3.88	5.64	2.78	3.85	3.6	3.23	2.72	3.36	3.62	2.62	1.44	1.9	6.57	0	5.27	4.85	4.74
JavaScript	2.94		6.79	2.05	0.81	1.6	1.97	0.63	0.4	0	3.06	1.67	0.58	1.72	3.75	3.6	6.81	1.22	2.42	4.97	4.97	6.99		1.42	4
Rust		0.26		0.21	2.83	2.56	1.97	4.62	0.08	0	4.6	1.21	3.53	3.72	3.1	1.43	7.54	5.55	5.26	6.36	4.81	6.04	10.35	0	0.95
VB.Net	0.21	4.13		2.93	0	0.36	4.29	0.8	7.83	4.89	8.74	3.83	3.54	3.18	2.49		0.06	2.82	1.83	2.91	6.58	5.39	4.82	0.97	14.82
PHP	0			0.75	0.13	4.93	5.11	5.8	5.47	1.02	2.67	0	4.29	1.97	5.49	5.65	2.22	7.42	7.41	7.84	3.71	7.07	7.93	10.45	10.89
Python	0.42	2.12	1.23	0.7	3.89	6.78	7.36	5.89	5.68	3.61	3.92	5.09	9.54	4.42	5.85	5.97	3.87	8.24	7.79	8.16	0	8.02	5.7	11.57	11.35
Ruby	0.44	1.01	0.71	0.07	4.16	5.76	5.8	6.86	5.36	1.36	6.06	3.73	8.36	5.12	5.63	6.63	3.27	7.59	7.43	7.16	8.59	7.89	8.34	9.17	10.69
Swift	0.97		1.23	9.42	4.81	0.29	1.24	5.2	7.04	6.23	4.74	6.57	3.47	8.48	7.8	6.6	7.95	7.33	10.13	9.36	7.81	11.08	0.34	3.62	7.88
Perl	0.79	2.75	1.3	0.01	1.3	7.55	0	6.95	7.64	2.31	4.92	1.68	8.48	5.59	7.36	8.32	9	8.88	8.28	8.72	7.23	8	10.11	14.29	14.13
Java	3.34	0.29	3.62	8.6	8.7	5.93	6.84	3.92	4.14	8.77	4.09	9.3	5.17	8.87	8.73	9.39	8.29	5.39	5.89	6.1	11.47	5.39	5.08	11.04	11.05
R	2.71			4.5	5.12	8.77	10.03	7	2.96	6.39	8.47	10.56	9.19		7.94	7.49	11.26	9.42	8.7	7.94	8.9	9.77		15.01	13.28

Figure 4.1: Server's EDP with Weighted Function 2

languages show efficient or inefficient results—we consider a programming language implementation X to be more efficient than Y if X's EDP value is lower than Y's. 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. We took such an action because some languages do not offer *classes* and *call-an-object-method*. For instance, C implements the above tasks using `struct`. Therefore, if C implementation for the task *classes* offered the lowest EPD and C++ the second lowest, then we picked the C++ implementation. Finally, we show the confidence interval, mean, and margin or error, for all the programming language implementations of a particular task.

Table 4.1 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 base ten logarithm of 7. We, therefore, used the base ten logarithm of the following ratio:

$$\log_{10}(p/p_{min}) \tag{4.1}$$

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 show in Figure 4.2 the score of EDP results for all programming languages, for each task, using box plots.

4.1.1.1 Embedded System Results

Grouped into Categories. The collected results, in Table 4.1 for the embedded system, show that C and Rust provide better EDP results than 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 4.7 times more efficient than Perl. In terms of file-handling, Rust offers the best efficiency against Swift by 4.2 times. JavaScript is the programming language that achieves 3.8 times better EDP results for the regular expression category, compared to Java. For almost all sorting tasks, Go implementations are 3.9 to 6.5 times more EDP-efficient compared to Swift, R, and Ruby. For performing functional tasks, C++ is proven to be 7.9 times more efficient than Swift.

Uncategorised 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 4.7 times more efficiently for *ackermann-function* than R, VB.NET outperforms R by 3.6 times for *palindrome-detection*, and Swift outruns PHP by 4.1 times for the *factorial* task. For the *url-encoding* and *decoding* tasks, C and PHP achieve EDP efficiency of 6.3 and 7.3 times more than Java and R, respectively. For the OO tasks, the most efficient implementations vary: (1) C++ outperforms Python 5.6 times for the *call-an-object-method* task; (2) JavaScript outperforms R 5.1 times for the *classes* task; (3) Ruby

Table 4.1: 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
huffman-coding	C	VB.NET	4.94	C	VB.NET	4.54	C	VB.NET	4.39
numerical-integration	Rust	Perl	5.60	C	VB.NET	4.20	C	VB.NET	3.35
lzw-compression	Rust	Swift	9.56	C	Java	4.10	C	Java	4.46
concurrent-computing	C	Perl	4.79	C	C++	6.00	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	4.90	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	4.40	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	5.00	C#	Swift	5.48	Go	Swift	5.55
selection-sort	Go	Ruby	3.98	JavaScript	R	3.60	C#	R	3.52
bubble-sort	Go	R	4.84	C#	Swift	4.60	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	5.10	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	3.60	C++	Java	1.79	C++	Java	1.68
function-composition	C++	Swift	7.96	C++	Perl	5.62	C++	Java	4.48

outperforms JavaScript 7.9 times for the *inheritance-multiple* task; and (4) C++ outperforms JavaScript 3.6 times for the *inheritance-single* task.

4.1.1.2 Laptop System Results

Grouped into Categories. Table 4.1 shows the results for the laptop platform where C performs 4.1 to 5.8 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 4.4 times more EDP-efficient than VB.NET's. Recursion category implementations (*ackermann-function*, *palindrome-detection*, and *factorial*) prove to be 4.4 to 5.1 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 6.8 to 7.1 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 5.6 times more EDP-efficient.

Uncategorised Implications. For the tasks *concurrent-computing* and *synchronous-concurrency*, C and C++ achieve the best efficiency against C++ and Java by being 6 and 1.8 times more efficient, respectively. C and PHP outperform Java by being 4.4 and 4.9 times more EDP-efficient for the *array-concatenation* and *json* tasks, correspondingly. For sorting tasks like *insertion* and *selection*, JavaScript outruns R by 4.2 and 3.6 times. In addition, C# shows 4.6 and 5.4 times more EDP-efficient results in contrast to R for *bubble* and quick sorting. Also, Go performs 4.7 times more efficiently for *merge* sorting compared to Swift. For the tasks of *url-decoding* and *encoding*, C++ and PHP perform 4.8 and 5.6 times better than R.

4.1.1.3 Server System Results

Grouping into Categories. Table 4.1 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 4.3 times more efficient than Java. For most of the tasks falling under the sorting category, Go performs 4.6 to 5.5 times more efficiently against R and Swift. C++ outperforms JavaScript and Java for OO tasks such as single and multiple inheritance by 1.6 and 3.3 times respectively. In addition, C++ performs 4.4 times better for the functional category compared to Java.

Uncategorised 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 4.3 and 4.4 times more efficient for the data structures category, correspondingly. C#, Perl, and C achieve 4.4, 5.1, and 5 times better results for the recursion category than R. Moreover, C performs more efficiently for *url-decoding* and PHP performs more efficiently for *url-encoding* against Java and R.

Table 4.2: 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.BET	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

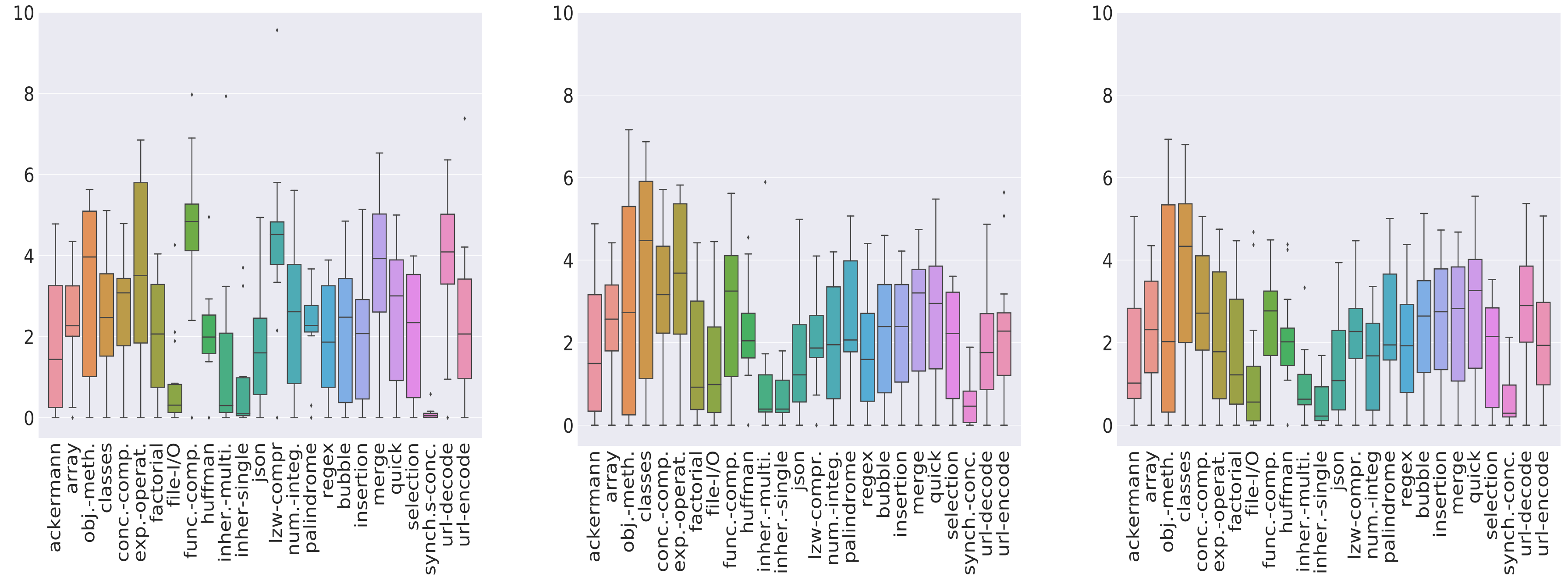
C is the best in arithmetic, compression, and concurrency, while C++, Go, and Rust are the runners-up. R, Perl, Swift, and Java show weak, overall, performance in terms of EDP. Go is exhibiting the best EDP results for sorting algorithms, Rust for file-I/O, JavaScript for pattern matching and replacing,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

4.1.1.4 Range of results

Figure 4.2 depicts the ranges of the EDP scores for each task that we measured 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.



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

Figure 4.2: 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.

4.1.2 PL-RQ2. Which types of programming languages are, on average, more EDP-efficient and inefficient for each of the 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 4.2 illustrates for each platform the ranking among the programming languages' average EDP and the influence of their weights; where this is applicable. The shaded areas in the corresponding table depict the languages that their average EDP were influenced by the weights. 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 embedded systems, 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 4.2. 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 3.1 for the *numerical-integration* case, while C# achieves 1.6, which makes it more efficient compared to JavaScript. However, when $w = 2$ or $w = 3$, JavaScript's EDP changes to 3.3 and 3.5 while for C# it changes to 3.1 and 4.3, respectively. This denotes that C# is consuming less energy but is slower than JavaScript.

By looking at the results presented in Table 4.2, 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.

4.1.3 PL-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 **PL-RQ1** and **PL-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.

Table 4.3: Wilcoxon's Pairwise Sum Ranking

Platforms	C#	C	C++	Go	JS	Java	Perl	PHP	Python	R	Ruby	Rust	Swift	VB.NET
E–L [a]	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
E–S [b]	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
L–S [c]	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

[a] Embedded system and Laptop platform
[b] Embedded system and Server platform
[c] Laptop and Server platforms

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

Let $\mathscr{L}$ be a programming language, and $\mathscr{P}_1, \mathscr{P}_2$, and $\mathscr{P}_3$ (where $\mathscr{P}_1 \neq \mathscr{P}_2 \neq \mathscr{P}_3$) the selected platforms of our experiment. Also, we pick the average EDP for all the tasks and we compare them in pairs: $(\mathscr{P}_1, \mathscr{P}_2)$; $(\mathscr{P}_1, \mathscr{P}_3)$; $(\mathscr{P}_2, \mathscr{P}_3)$. Since each platform's results were used twice in a comparison (once for each other platform), we used the Bonferroni correction [19] to counteract the multiple comparison problem. Therefore if the test's p-value for the pair $\mathscr{P}_x$ and $\mathscr{P}_y$ (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 $\mathscr{P}_x$ and $\mathscr{P}_y$ is statistically significant for the $\mathscr{L}$ 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 4.3 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 Table 4.3).

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.

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 cases, of the embedded and laptop platforms.

4.1.4 Discussion

We investigate the root causes of the results of this section 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 collective results from the heatmaps can be useful for practitioners in developing energy-aware applications.

4.1.4.1 Champions

Concurrency. Best efficiency in EDP is achieved in programming languages that rely on their library implementations for concurrency. For instance, C, that achieved far better EDP, uses OpenMP [84] and libco [31] 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 the `Node.js`, a run-time environment utilising the V8 JavaScript engine (built in C++) and `libuv` (built in C), that achieves a speed-up for regular expressions, then the RegEx library is built on top of Irregexp, in combination with CodeStubAssembler [59]. Therefore, we conducted an experiment where we used `nvm` [32], a `Node.js` version management utility, to install a `Node.js` 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 of the *regular-expression* tasks increased by 331% (for the particular Node.js version).

Object-Oriented Features. 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 are 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 another language such as 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 [61].

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` [48] Linux built-in command—and we found that VB.NET's intermediate code for the task of *file-I/O*, makes 14 times more system calls in total compared to Rust. According to Aggarwal et al. [9], when the system calls between two applications diverge significantly, the applications' power usage may 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 `munmap` (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 occur because VB.NET's I/O-buffers are smaller than Rust's and require 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 preprocessor. As a result, C++ has faster execution time and less energy consumption resulting in more efficient EDP compared to the other implementations.

4.1.4.2 Impact

We believe that a developer can consult our heatmaps and use them as a guideline to develop more EDP-efficient applications. For instance, a developer can create a micro-services application by developing each service using a particular programming language that is more EDP-efficient for a task. For the development of a more complex application—that may combine more than 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 4.1.2 that Java and Swift, the major programming languages for developing Android and iOS applications, they are on average EDP-inefficient. Therefore, developers can use alternative efficient practices. For instance, when developing Android applications, practitioners can use the Native Development Kit [46] 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 language 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.

4.2 IPC Results Analysis

In this section, we discuss the collected results of the energy consumption and run-time performance that the selected computer platforms achieved for each IPC technology implementation. We also answer our research questions, discuss the significance of our measurements, and offer reasons behind our findings.

4.2.1 IPC-RQ1. Which IPC technology implementation offers the most energy- and run-time performance-efficient results?

We first present the obtained results for the Intel and ARM computer platforms. Specifically, we show which type of IPC technologies and programming language implementations are the most energy- and run-time performance-efficient among our scenarios. Tables 4.4 and 4.5 illustrate the median values of the energy consumption (energy in Joules) and run-time performance (time in seconds) for a particular programming language. Moreover, we compare the corresponding implementations viz-a-viz the best implementation's results (most efficient case) in the form of the ratio. Also, Figures 4.3a and 4.3b present box plots regarding the programming languages results for each IPC technology.

Table 4.4: Energy and Run-time Performance Results for Intel Platforms

		Nodes' Collected Measurements				Ratio comparison against the most efficient case		
IPC Names	Programming Languages	Energy (in joules) Client	Server	Time (in seconds) Both Nodes	Total Energy Consumption	Energy (in joules) Client	Server	Time (in seconds) Both Nodes
gRPC	Go	99.0	105.8	28	204.8	3.5	3.9	9.3
	Java	451.5	364.0	35	815.5	16.0	13.4	11.6
	JavaScript	28.2	27.0	3	55.2	–	–	–
	Python	606.7	588.7	16	1195.4	21.5	21.8	5.3
	PHP	11543.9	1291.3	479	12835.1	409.3	47.8	159.6
	Ruby	139.4	61.6	39	201.0	4.9	2.2	13.0
	C#	65.6	105.2	55	170.8	2.3	3.8	18.3
RPC	Go	84.8	70.4	16	45.2	2.3	2.1	1.6
	Java	5224.5	2715.5	855	7940.0	144.3	84.0	85.5
	JavaScript	36.2	32.3	10	26.5	–	–	–
	Python	319.5	347.0	73	666.5	8.8	10.7	7.3
	PHP	930.5	1214.6	67	25.7	10.2	37.6	6.7
	Ruby	458.2	513.5	43	971.7	12.6	15.8	4.3
	C#	399.9	364.0	27	763.9	11.0	11.2	2.7
REST	Go	94.1	79.0	19	260.3	1.0	2.5	1.9
	Java	687.0	617.8	42	1304.8	7.7	19.6	4.2
	JavaScript	88.1	31.4	10	109.5	–	–	–
	Python	637.0	370.0	78	1007.0	7.2	11.7	7.8
	PHP	7003.0	20574.5	628	79.4	90.0	65.2	62.8
	Ruby	1988.2	6191.2	304	8179.4	22.5	197.1	30.4
	C#	1206.6	789.4	44	1996.0	13.6	25.1	4.4

4.2.1.1 Intel platforms

Comparison among IPC technologies. Table 4.4 illustrates the obtained results of the Intel platforms. We can see that, for both server and client, the gRPC offers the lowest energy consumption and execution time for all the IPC technology implementations apart from those of Go and PHP. For Go's implementation, gRPC has the highest energy consumption and lowest run-time performance compared to REST and RPC which are making use of the built-in `net RPC` and `HTTP` libraries. The results also present that RPC is the IPC technology that has the next best results, regarding energy consumption and run-time performance, for all the implementations except from Java, while it offers the best results for Go. Also, we can see that REST implementations contribute to the highest energy consumption and lowest run-time performance among the implementations.

Comparison among programming languages. The results of Table 4.4 show that JavaScript is the programming language that exhibits the best results in all cases. In addition, we can see that Go outputs the second most energy- and run-time performance-efficient results, while C# is following, and last we observe that Java, Python, Ruby, and PHP offer the lowest performance.

Table 4.5: Energy and Run-time Performance Results for ARM Platforms

		Nodes' Collected Measurements				Ratio comparison against the most efficient case		
IPC Names	Programming Languages	Energy (in joules) Client	Energy (in joules) Server	Time (in seconds) Both Nodes	Total Energy Consumption	Energy (in joules) Client	Energy (in joules) Server	Time (in seconds) Both Nodes
gRPC	Go	10.8	6.9	10	17.7	4.9	5.3	3.3
	Java	83.5	75.6	71	159.1	45.2	58.1	23.6
	JavaScript	2.2	1.3	3	3.5	–	–	–
	Python	11.7	16.5	18	28.2	5.3	12.6	6.0
	PHP	348.7	88.7	331	437.4	201.2	68.2	110.3
	Ruby	8.3	8.3	12	16.6	3.7	6.3	4.0
	C#	–	–	–	–	–	–	–
RPC	Go	5.2	6.6	7	11.8	–	–	–
	Java	73.3	40.6	258	113.9	14.0	6.1	36.8
	JavaScript	20.9	20.5	22	41.4	4.0	3.1	3.1
	Python	31.0	36.5	52	67.5	5.9	5.5	7.4
	PHP	7.7	8.8	28	16.9	1.4	1.3	4.0
	Ruby	33.0	35.0	42	68.0	6.1	5.3	6.0
	C#	219.3	290.7	53	510.0	42.1	44.0	7.5
REST	Go	4.1	3.2	8	7.3	–	–	–
	Java	33.9	43.5	22	77.4	8.2	13.5	2.7
	JavaScript	18.3	18.3	22	36.6	4.4	5.7	2.7
	Python	60.4	35.0	75	95.4	14.7	10.9	9.3
	PHP	175.8	360.1	153	535.9	42.8	112.5	19.1
	Ruby	196.0	1094.0	679	1290.0	47.8	341.8	84.8
	C#	45.3	96.5	68	108.5	11.0	30.1	8.5

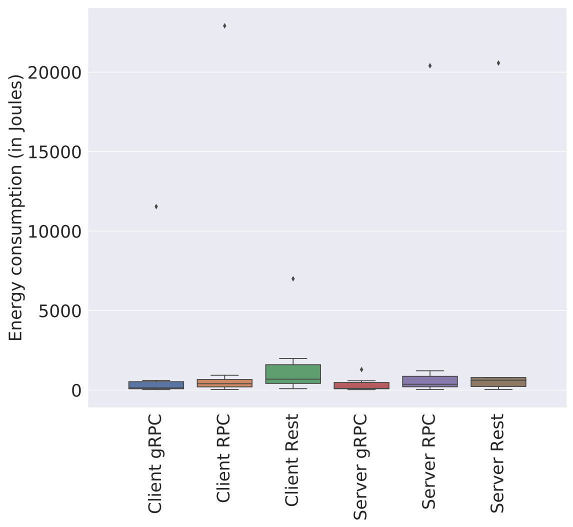
4.2.1.2 ARM platforms

Comparison among IPC technologies. Table 4.5 depicts the IPC results of the ARM platforms. Likewise to Intel platforms results, the gRPC again contributes to the lowest energy consumption and execution time, for most cases, among the selected IPC technologies in the context of ARM platforms for the server and client instances. In contrast, Go, Java, and PHP implementations are having the most inefficient results while executing the gRPC task. For RPC scenarios, we observe that our implementations have the next best results after those of Java and C#. Among the IPC technologies, REST resulted in the least energy- and run-time performance-efficient results for the ARM platforms.

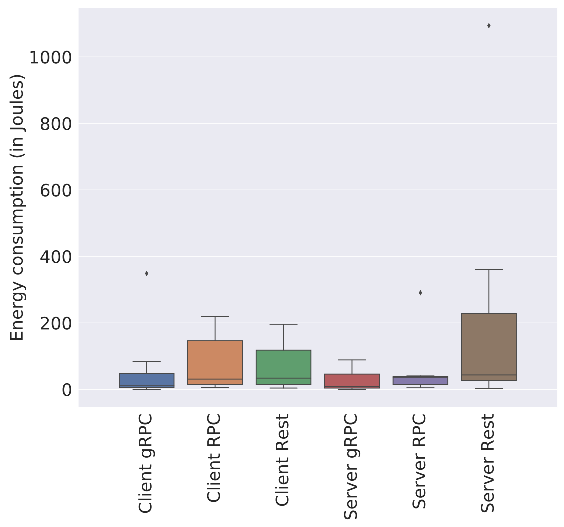
Comparison among programming languages. JavaScript also offers the best results for the ARM platforms when it comes to the gRPC task. However, in contrast to the Intel platforms, Go implementations resulted in the most efficient results in terms of energy consumption and run-time performance, while JavaScript is the runner-up for RPC and REST tasks. Compared to the Intel platforms, the presented results for the ARM systems do not depict a clear winner among the remaining language implementations.

4.2.1.3 Range of results

Figures 4.3a and 4.3b illustrate the box plots of the obtained median energy consumption of each implementation with a confidence interval of 95%. The points located in our box plots,



(a) Intel platforms' client and server results



(b) ARM platforms' client and server results

Figure 4.3: Intel and ARM computer platforms results

that are above the maximum values measurements are highly energy-inefficient implementations.

In the case of Intel platforms (see Figure 4.3a), we notice that only the REST implementations diverge significantly among the results, for both client and server instances. For the RPC's implementations, we can observe small differences in their energy consumption, which is even smaller for gRPC results. Also, we observe that some highly inefficient implementations exist for gRPC and REST which are those of PHP.

In contrast to Intel platforms, for the ARM systems, we see higher divergences in the implementations' energy consumption (see Figure 4.3b). Such a fact highlights the necessity of proper selection, of IPC protocol and language implementation in the context of embedded systems and battery-restricted devices. Likewise to Intel platforms, PHP implementation for gRPC has the lowest performance. However, for the REST and RPC scenarios, Ruby and C#, respectively, had the most inefficient implementations.

JavaScript and Go are the programming languages offering the most energy- and run-time performance-efficient library implementations for the Intel and ARM platforms. In addition, for almost all programming language implementations, we found that gRPC is the IPC technology having the most efficient results.

4.2.2 IPC-RQ2. What are the reasons that make certain IPC technologies more energy and run-time performance-efficient?

To answer **IPC-RQ2**, we execute the whole experiment one more time to retrieve and analyse system calls from the Intel and ARM platforms. Likewise, Aggarwal et al. [8, 75] investigated how the energy consumption of applications is changing according to the number of their system calls. They showed that when the number of system calls between two applications diverges significantly, it is more likely that the application's energy consumption will differ too. However, what we do here is that we examine the system call traces produced by our IPC technology implementations qualitatively and we try to delineate the reasons behind our results. Therefore, we first analyse the obtained system call traces (for the client and server) and then we try to interpret and discuss our findings.

To collect system call traces, we utilise the `strace` command-line tool and we collect data using the flags `-c` (provides a summary-like output) and `-f` (retrieves child process traces). Due to the large volume of results, we did not include the collected system call traces in this study; however, they are publicly available in our GitHub repository.[1]

4.2.2.1 Platforms' System Calls

On the ARM and Intel platforms' system calls, we identify a large number of wait-like system calls such as `futex`, `waitid`, and so on. By investigating the results, we observe that Go is using the `futex`, `waitid`, and `epoll_wait` system calls extensively; this is not happening for the JavaScript implementations. By reading the official documentation of the IPC technology implementations we found out that Go, for all the IPC technologies, is using *channels*;[2] a synchronous method to serialise main memory access and increase thread-safety [114]. Therefore, it forces the client to wait for an answer from the server before invoking the next remote procedure. This increases execution time and thereby adds to the energy footprint through the system's fixed energy consumption cost.

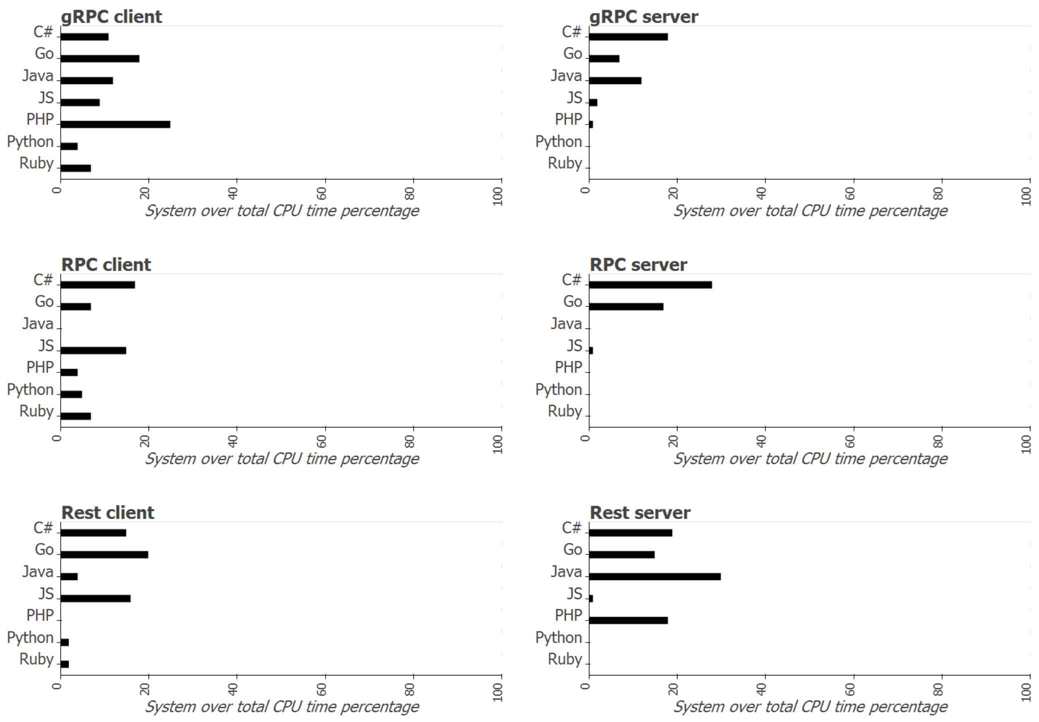
Likewise, Python implementation system calls are spread among `socket`, `connect`, `close`, `sendto`, `recvfrom`, `fcntl`, and `stat` for the REST and RPC technologies. Although Python is not using broadly wait-based system calls, it still has the implementations with the lowest performance for both energy usage and execution time. In the case of gRPC, Python supports both synchronous and asynchronous methods to interact with the client's and server's stubs. However, in the example, a synchronous method is used and, thus, the `futex` system call takes up most of the implementation's execution time.

We observe similar behavior, with the above, for C#, Java, Ruby, and PHP implementations. JavaScript, on the other hand, due to its asynchronous nature spends most of its execution time on system calls such as `writev, mmap, munmap, read, brk, socket, connect`, and less than 20% of its execution time on `epoll_ctl`.

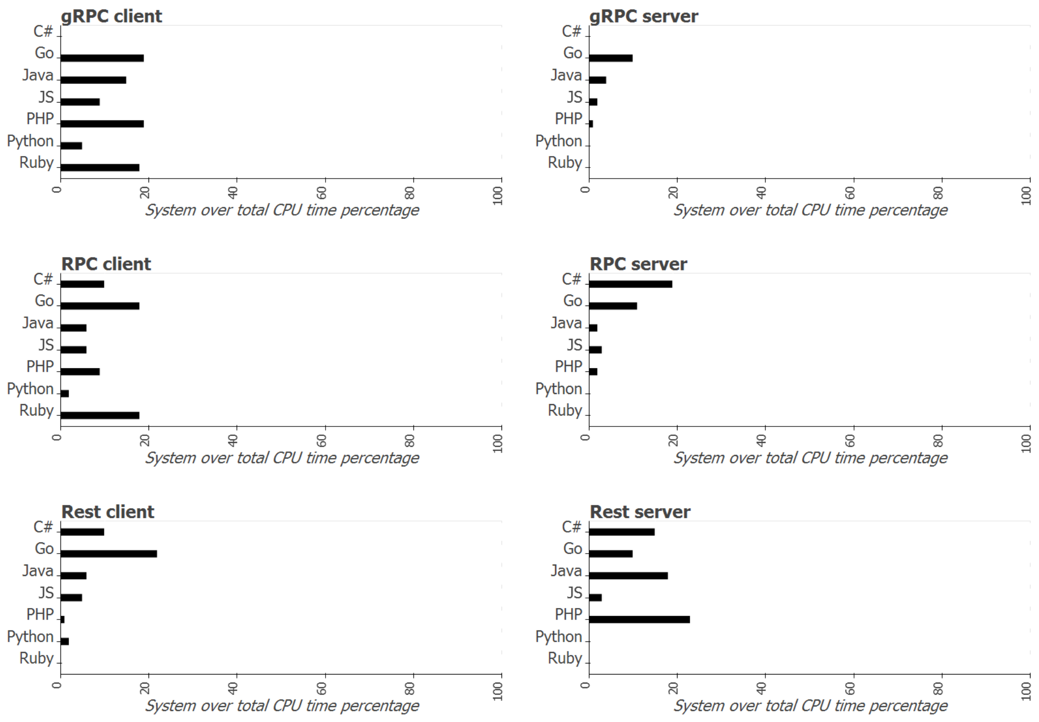
4.2.2.2 Identifying the Facts

We execute our experiment again by using the `-e` flag to sanitise our traces from the wait-like system calls (*e.g.*, `futex`, `wait4`). We do that since these system calls indicate that an implementation is not using any computing resource—since it is in a sleeping state—and to diagnose which system calls might impact its energy consumption and execution time. Additionally, we remove traces that are related to the compilation since they do not offer an actual execution of the tasks. Figures 4.4a and 4.4b, illustrate the time that each of the implementations spends in kernel space (`sys` time) against the `real` time for the associated computer platform and IPC protocol. The Y-axis supplies information regarding the total

[1]https://github.com/stefanos1316/Rest_and_RPC_research/arm/syscalls
[2]https://gobyexample.com/channels



(a) Intel platforms' system calls impact



(b) ARM platforms' system calls impact

Figure 4.4: Computer Platforms System Calls

median time (of 50 executions) that IPC implementations spent on system calls during their whole execution, while the x-axis shows the relevant programming language implementations.

After obtaining our traces, we compared the most and least efficient IPC implementation system calls across each of the programming languages according to the results of subsection 4.2.1. Also, we performed an intra-language (instead of inter-language) comparison of the IPC implementations to be just with languages supporting only asynchronous or synchronous function calls for the investigated tasks. We did that separately for the programming languages affected heavily from the system calls such as C#, Go, and PHP. Moreover, we analysed JavaScript's system calls because JavaScript offers the most energy- and run-time performance-efficient implementations.

For C#, the results for both platforms suggest that the server instances are way more affected by the system calls against the clients. For the RPC implementations, the `sched_yield` system calls occupy a major portion of time causing a large number of context switches which degrades the implementations' performance. This might also be the reason that places C#'s RPC among the implementations with the poorest energy and run-time performance, especially for the ARM platforms (see Table 4.5).

In contrast to C#, Go's client implementations are getting affected more from the system calls, close to 20% of their total execution time. Also, Go's RPC is the most energy- and run-time performance-efficient implementation, while gRPC gives the weakest results. To this line, the system call traces are showing that both of them are using mostly the same system calls *e.g.*, `write, read,` and `sched_yield`. However, for the Intel platforms, what makes them different is that RPC makes, in total, at least twice as many system calls against gRPC. Therefore, by taking into account the work of Aggarwal et al. [8] this can explain the reasons why RPC's implementation is more energy-efficient than gRPC's. For the ARM platforms, the same is not happening since the number of their system calls are similar.

PHP's RPC—which has the best energy and run-time performance among PHP's implementations for the client instances—is mainly using system calls such as `connect, close, send, recv`, and `socket`, while the client-side gRPC is extensively using the `openat` and `mmap2` to map data on virtual memory. For the server instances for both platforms, REST implementation for PHP suffers from a great number of system calls.

JavaScript that has gRPC as the most energy- and performance-efficient implementations makes broad use of `writev` system calls that write data into multiple buffers. Also, JavaScript's gRPC is using mostly the `read, write, writev,` and `clock_gettime` system calls, while its REST implementation (the most efficient one) utilises considerably the `socket, connect,` and `close` system calls for the client and `accept` and `shutdown` for the server.

Our analysis shows the frugal opening, connecting, closing, accepting, and shutting down connections can impact the energy consumption and run-time performance of the IPC technologies. Moreover, an extensive number of context switches can severe implementations' performance. Besides, the usage of `writev` *system call appears in the most efficient implementations.*

4.2.3 IPC-RQ3. Is the energy consumption of the IPC technologies proportional to their run-time performance or resource usage?

In **IPC-RQ3**, we aim to identify if the energy consumption of different programming language implementations, in the context of IPC technologies, have proportional (1) run-time performance and (2) resource usage. To this line, a similar research question to ours was answered by Pereira et al. [127] where they compared the energy consumption, run-time performance, and memory usage of 27 different programming languages and showed that energy consumption is not proportional to the memory usage, while in some cases it is proportional to the execution time. We perform a similar experiment in the context of IPC technologies and we examine resource usage as described below. To answer **IPC-RQ3**, we break it down into two sub-questions and we answer separately as follows.

4.2.3.1 Energy consumption and run-time performance

Initially, we collected the median values of the energy consumption and run-time performance of each implementation from our experiment as shown in Tables 4.4 and 4.5 and we rendered bar plots. This offers a complete picture regarding the proportionality of measurements for each of our platforms. In Figures 4.5 and 4.6 the X-axis presents the IPC language implementations, while the Y-axis depicts the median energy consumption, for the server or client instance, and their run-time performance.

From the outcome, we observe cases where lower execution time is not associated with reduced energy consumption. For instance, for the Intel platform, the C#'s REST client and server implementations have execution time 1.2 times lower than the C#'s gRPC; however, the REST implementations energy consumption is 7.5 times higher than the gRPC's. Similarly, Java's RPC implementations are 1.3 times slower than PHP's REST; nevertheless, PHP's client and server consume 1.3 to 7.5 times more energy, respectively. Also, Ruby's REST server uses more energy than PHP's gRPC server but still, Ruby's implementation is 1.5 times faster. Go's server and client RPC instances have the same execution time as Python's server and client; however, Python's implementations consume almost three times more energy (see Figure 4.5).

For the ARM and Intel's platforms, we observe a similar behavior. Some cases depict that energy consumption is proportional to execution time, while other not (see Figure 4.6). For example, C#'s client and server implementations for RPC results to 1.2 times lower execution time than C#'s REST implementations. However, C#'s REST server and client instances are 4.8 and 3 times, respectively, more energy-efficient than C#'s RPC. Additionally, C#'s RPC server and client implementations are 1.3 times faster than Java's gRPC server and client; nevertheless, Java's implementations are 2.6–3.8 times more energy-efficient than C#'s RPC implementations. Also, PHP's REST implementations are 1.6 times faster than Java's RPC; but, Java's implementation uses 2.3 and 8.8 times less energy than PHP REST client and server, respectively.

In the context of IPC technologies, almost all the selected programming language implementations have proportional median values for the energy consumption and run-time performance. However, we found many cases where fast execution time did not result in energy savings.

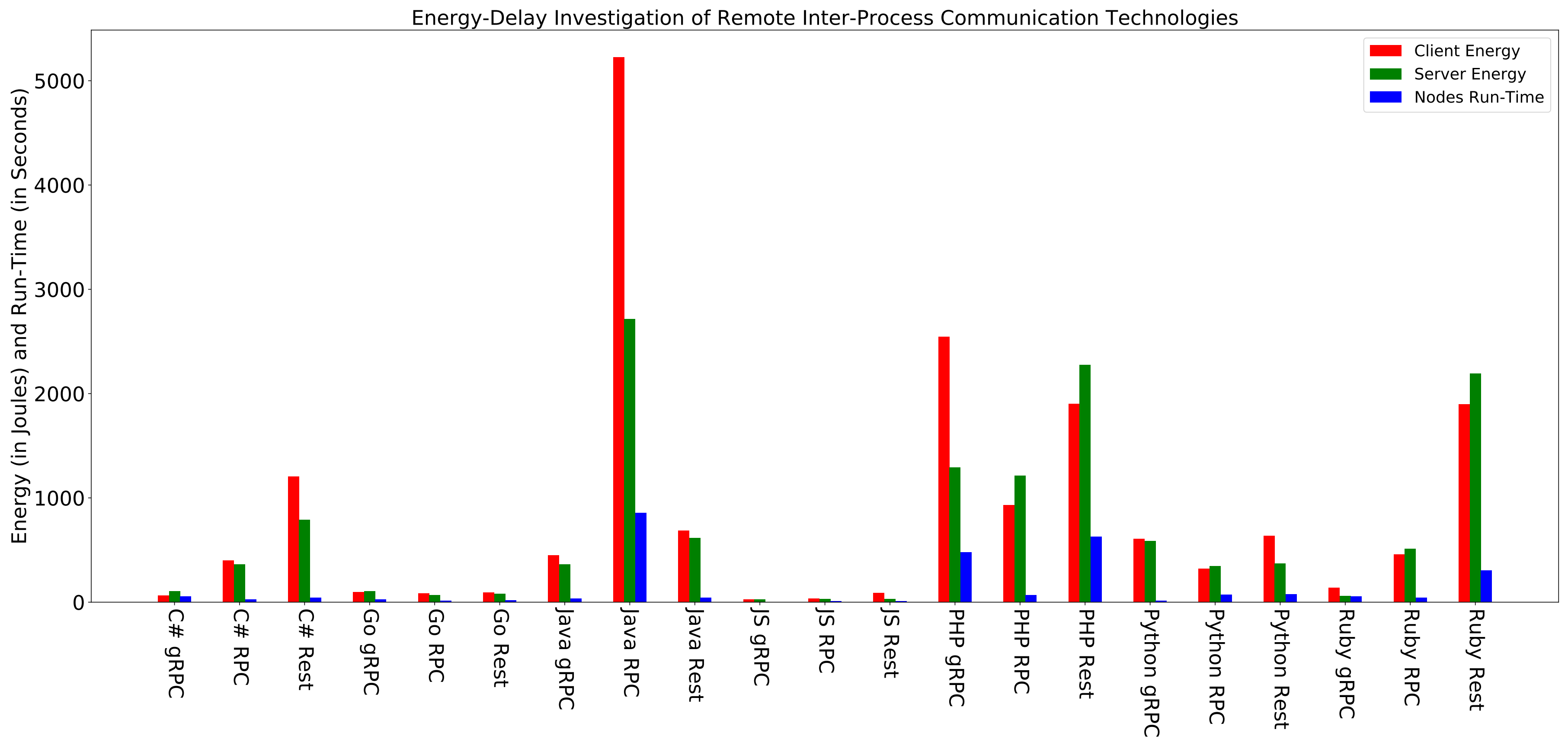


Figure 4.5: Intel platforms' Energy and Run-Time Performance Comparison

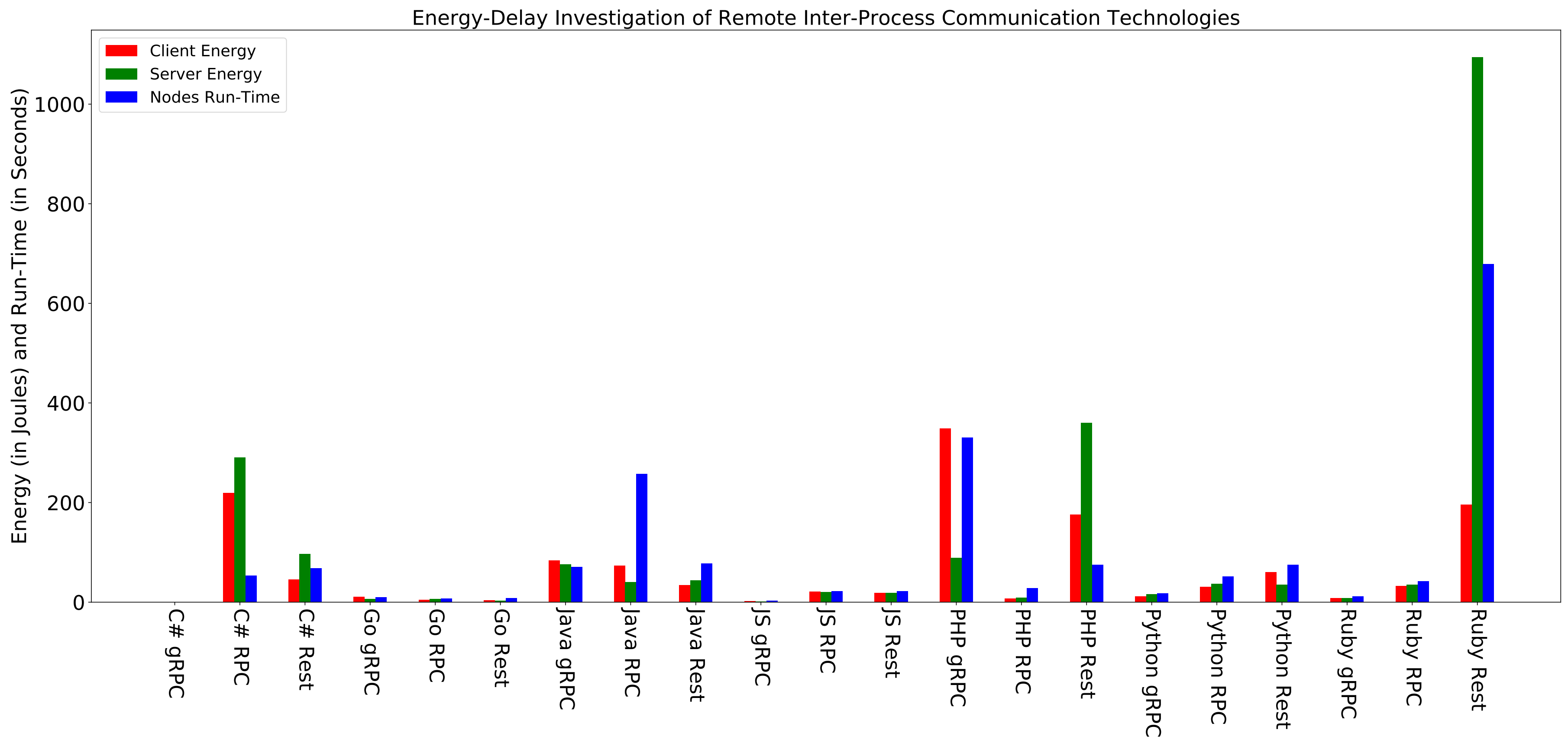


Figure 4.6: ARM platform's Energy and Run-Time Performance Comparison

Table 4.6: Intel platforms' MRSS (in KB), MPF, and VCS

					Server							Client			
IPC	Data Name	C#	Go	Java	JS	PHP	Python	Ruby	C#	Go	Java	JS	PHP	Python	Ruby
gRPC	MRSS	92	161	328	357	47	27	23	90	117	329	407	28	26	21
	MPF	81	55	90	86	51	4	5	77	216	95	99	4192	7	5
	VCS	167	137	121	0.5	41	266	42	25	136	125	1	470	20	40
RPC	MRSS	143	124	171	98	22	22	21	102	88	168	467	26	22	26
	MPF	128	45	50	21	1	26	6	83	34	50	113	1	6	6
	VCS	151	86	85	0.6	32	366	76	85	82	65	4	40	59	84
REST	MRSS	213	100	6	98	40	22	487	51	72	142	457	26	19	20
	MPF	114	38	2	21	5208	6	128	179	30	40	110	1	7	5
	VCS	174	121	0.05	0.6	29	215	119	824	149	48	4	45	58	113

4.2.3.2 Energy consumption and resource usage

In this section, we aim to observe whether the resource usage of the different implementations is proportional to energy consumption. To this end, we used the Linux `/usr/bin/time -v` command (version 1.9), which offers information regarding the implementation's resource use, such as context switching, page faults, and main memory usage. In this way, we can have a deeper understanding of the way each implementation is allocating memory and different operations that are causing peculiar system calls.

Tables 4.6 and 4.7 illustrate the Maximum Resident Set Size (MRSS), the Minor Page Faults (MPF), and the Voluntary Context Switches (VCS) for the client and server, respectively. We show the peak main memory usage for our implementations' processes by using the MRSS that is showing the portion of memory occupied by a process in the main memory. Additionally, we select the MPF since it can show the number of cases when our implementations are trying to access particular memory pages that are not currently mapped in the virtual address space. Also, we ignored the Major Page Faults because only a few instances occurred in our results. Finally, we use the VCS to observe the number of times an implementation's processes were context-switched while waiting for resources that were unavailable at that time. For the sake of simplicity, results reported in Tables 4.6 and 4.7 have been divided by 1,000 in order to improve readability.

For the Intel platforms, we observed, for the gRPC and RPC, that the most energy- and performance-efficient implementations which are C#, Go, Java, and JavaScript tend to have high MRSS, while PHP, Python, and Ruby have the lowest MRSS and overall performance among the implementations (see Table 4.6). We observe similar behavior for the REST implementations apart from Ruby that has the highest MRSS but still the poorest performance. Regarding the MPF metrics, we found a significant divergence within the programming language implementations. Therefore, finding an association between our results, it is not possible. Likewise to MPF, the number of VCSs do not affect the energy efficiency of the selected IPC implementations. We identified some of the most energy-efficient implementations having a very low VCS, while the same holds for inefficient implementations.

For the ARM platforms, we observed for the MRSS there is not a clear indication regarding the proportionality of memory usage for the most energy-efficient implementations against the least efficient ones. However, in Table 4.7, we observe that the implementations with high MRSS, such as Go and JavaScript, are the most energy- and run-time performance-efficient. Regarding the MPF and VCS for the ARM platform's implementations, we did not find any association with energy consumption as the results tend to be sparse among the

Table 4.7: ARM platforms' MRSS (in KB), MPF, and VCS

		Server							Client						
IPC	Data Name	C#	Go	Java	JS	PHP	Python	Ruby	C#	Go	Java	JS	PHP	Python	Ruby
gRPC	MRSS	0	93	66	79	34	17	12	0	91	66	94	20	16	10
	MPF	0	35	15	17	5	2	17	0	34	15	21	7974	2	16
	VCS	0	78	180	0.08	0.04	84	20	0	66	267	0.1	10	5	10
RPC	MRSS	84	68	32	67	15	18	12	84	69	30	94	20	17	13
	MPF	243	26	6	15	1	5	13	213	26	5	22	1	5	17
	VCS	243	33	4	0.4	8	24	13	108	27	17	3	10	14	26
REST	MRSS	82	57	2	70	31	18	126	49	57	29	96	20	20	14
	MPF	77	25	2	16	15	4	328	46	24	5	22	1	5	2
	VCS	81	37	0.03	0.4	8	5	57	146	50	11	3	18	19	36

most and least efficient implementations.

Neither the Minor Page Faults nor the Voluntary Context-Switches can be used to justify the energy consumption results in terms of IPC technologies. However, the number of Maximum Resident Set Size tends to be proportional to most of the cases. Therefore, drawing clear conclusions regarding energy consumption and resource usage is challenging.

4.2.4 Discussion

In this section, we examine the reasons behind our results and try to point out the implications. Moreover, we discuss their significance and potential for adoption of recommended practices.

4.2.4.1 Interpreting the Findings

From the collected results, JavaScript emerges as the programming language with the lowest energy consumption and best run-time performance implementation for gRPC. All the gRPC libraries, for all the selected programming languages, are using shared C as their core-library to build their implementations on top of it. However, JavaScript's gRPC library implementation is using C++ native addons[3] to achieve better performance, a pattern that is not applied for the other implementations. Additionally, Oliveira et al. showed that combining Java or JavaScript applications with native programming languages such as C/C++, can offer up to $100\times$ times less energy consumption and ten times better run-time performance for devices with ARM micro-processor [113]. Also, our study on Georgiou et al. [58] and that of Pereira et al. [128] have shown that C++ is among the most energy-efficient programming languages for servers and laptops. Therefore, using C++ with native addons helps JavaScript to reduce energy consumption.

[3] https://nodejs.org/api/addons.html

4.2.4.2 Lessons learned

Each of the selected implementations is making different use of the system calls in their lower level of abstraction. Employing tools such as `strace` to investigate them can provide hints to identify reasons behind the increased energy consumption and poor run-time performance. We have also observed that JavaScript had the lowest number of wait-like systems calls because Node.js—JavaScript's server-side run-time environment—uses an event-driven, asynchronous model. Therefore, JavaScript's server program is not waiting for a function's or an API's return data to start serving another request. However, when a function or API call returns the request's data, Node.js uses a notification mechanism to send immediately back data to the client. Therefore, compared to all the other implementations, Node.js fully utilises its execution time to serve client requests through its asynchronous nature and having less wait-like system calls during execution time.

By examining only the system calls of IPC implementations, we cannot always have a clear picture of their energy and run-time performance implications. For instance, Go implementations spend an important amount of time in kernel space; nevertheless, they are among the most energy and run-time performance-efficient implementations. This suggests that the type of system calls can affect the energy and run-time performance of IPC implementations.

4.2.4.3 Significance of Measurements

In this section, we demonstrate the significance of our measurements by illustrating the possibilities of energy savings while selecting a particular IPC implementation. Later on, we justify its feasibility for software practitioners to utilise our findings and gain a more energy-conscious development.

According to the work of Hindle [68], even minor optimisations on the energy consumption of smart phones for four million users per hour can result in significant worldwide energy savings equivalent to an American household's monthly power use per hour. According to WebFX,[4] there are more than 4.5 billion of Facebook posts from various ICT products on a daily basis, where clients use IPC implementations to interact with servers. If we consider that Facebook is built on PHP then we can assume—through rough calculations—that the energy cost of 20,000 post requests through gRPC can cost up to 22,560.9 and 12,834.4 Joules for an Intel server and client, respectively (see Table 4.4). That is translated to 1.39 and 0.76 Mega-Watt hours for 4.5 billion Facebook daily post requests for a server and client instance, respectively. However, switching to the most energy-friendly implementation of gRPC (*i.e*, JavaScript) that consumes 28.2 and 27 Joules for 20,000 post requests, leads to 0.0017 and 0.0016 Mega-Watt hours respectively, for an Intel server and client to serve, which amounts to the daily post requests of Facebook. The PHP IPC implementation of Facebook daily amount of post requests is slightly higher than the monthly power use of an American's household [1], while using JavaScript reduces power consumption significantly.

[4]`https://www.webfx.com/internet-real-time/`

4.3 Safeguards Results Analysis

We analyze the results produced by our experiments with respect to the research questions we set in section 2.1.1. We observe the behavior of the different safeguards, highlighting particular findings and pointing out ways to reduce energy consumption.

4.3.1 SG-RQ1. What are the energy and run-time performance implications of the safeguards on a computer system?

First, we examine the results related to the security mechanisms that protect against the attacks exploiting CPU-related vulnerabilities. Then, we describe the potential impact of using HTTPS, and discuss our findings on memory zeroing. Finally, we observe the implications associated with the GCC safeguards.

4.3.1.1 CPU Vulnerability Patches

We employed several benchmarks to measure the energy consumption and overhead of CPU-related vulnerability patches. In particular, we evaluated 128 benchmarks, consisting of 310 tasks. We discuss the obtained results in this section for each category in the following order: (1) video and audio encoding, (2) code compilation, (3) file compression, (4) database suites, (5) compute-intensive, (6) disk suites, (7) kernel operations, (8) machine learning, (9) memory suites, (10) network suites, (11) imaging benchmarks, (12) renderers, and (13) desktop graphics. Regarding the *computer-intensive* benchmark category, we listed the tasks with the highest performance degradation because we had too many results. Nevertheless, all measurements are publicly available in our GitHub repository. [5]

The highlighted rows in each table of this section, denote that the associated task's energy consumption is more than 1% higher when all the CPU vulnerability patches are disabled (*AllOff*) against the baseline scenario where all the CPU vulnerability patches are enabled (*Stock*). Specifically, the rows highlighted with orange color indicate an energy reduction between 1% and 4.9%. Additionally, the rows with yellow color denote a reduction between 5% and 9.9%. Finally, the green color illustrates tasks where we identified more than 10% of energy savings after disabling all CPU vulnerability patches. Due to the large amount of results in this particular section, we focus on the energy and run-time performance differences between the *Stock* and *AllOff* scenarios.

Video and Audio Encoding. Table 4.8 illustrates the results of the audio and video encoding/decoding benchmark suites. Our results suggest that the CPU vulnerability patches have a small impact on the majority of the corresponding tasks' energy and run-time performance. Only the tasks of *encode-flac*, *svt-av1*, and *vpxenc* gained energy savings of 1.2% on average, while their run-time performance remained unaffected.

Code Compilation: In Table 4.9, we illustrate the energy-delay implications of tasks associated with code compilation. Specifically, in this category, we obtained energy and run-time performance measurements of the corresponding tasks while they were compiling. Our

[5] `https://github.com/stefanos1316/resultsForEECSE20.git`

Table 4.8: Video and Audio Encoding/Decoding Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
dav1d_chimera	11522	267	11514	267	11473	266	11461	267	11478	267	0.7	0
dav1d_nature	2941	53	2935	53	2924	52	2918	52	2927	52	0.7	1.8
encode-mp3	666	28	667	28	665	28	668	28	661	28	0.7	0
encode-flac	424	20	420	20	420	20	419	20	418	20	1.4	0
ffmpeg	293	6	293	5	294	6	291	5	292	5	0.1	16.6
svt-av1	2192	37	2174	37	2151	36	2163	37	2166	37	1.2	0
svt-hevc	1584	27	1587	27	1570	27	1578	27	1576	27	0.5	0
svt-vp9	4260	77	4271	77	4254	76	4253	77	4238	76	0.5	1.2
vpxenc	1208	25	1201	25	1194	25	1196	25	1195	25	1.1	0
x264	733	13	732	13	730	13	734	13	732	13	0.1	0
x265	1017	17	1017	18	1010	17	1018	18	1016	17	0.1	0

Table 4.9: Code Compilation Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
build2	10419	196	10352	197	10329	197	10282	197	10247	195	1.6	0.5
gcc	17039	359	16830	353	16898	355	16796	350	16539	345	3	3.9
gdb	5791	134	5695	131	5729	132	5666	130	5587	128	3.5	4.4
kernel	81490	1555	73093	1382	73330	1393	72572	1371	72103	1374	11.5	11.6
llvm	45784	864	45569	861	45488	862	45302	858	45052	851	1.5	1.5
php	4403	99	4380	98	4366	98	4360	98	4318	97	1.9	2

results show that all CPU vulnerability patches affect the energy consumption and run-time performance compilation of the corresponding tasks, but not more than 5% for almost all cases. Notably, the Linux kernel version 5.3 compilation suffers more than any other test case for the corresponding category (see *kernel* in Table 4.9). In particular, the CPU vulnerability patches increased its energy consumption by 11.5% and execution time by 11.6%.

Table 4.10: File Compression Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
bzip2	375	17	377	17	375	17	373	17	371	17	1.1	0
lz_brotli	12516	573	12551	572	12505	573	12424	573	12476	572	0.3	0.1
lz_deflate	5820	262	5850	262	5810	262	5790	262	5808	261	0.2	0.4
lz_xz	13101	634	13028	633	13014	633	12864	633	12936	633	1.2	0.2
lz_zstd	13842	625	13721	625	13698	624	13680	623	13685	624	1.1	0.3
p7zip	1756	37	1749	37	1745	37	1748	37	1733	37	1.3	0
xz	1628	80	1617	80	1621	81	1612	80	1604	80	1.4	0
zstd	389	10	388	10	388	10	388	10	385	10	0.8	0

File Compression. Focusing on the results seen in Table 4.10, we observe that CPU vulnerability patches impact neither the energy consumption nor the run-time performance of file compression algorithms (affected by less than 2%).

Database Suites: CPU vulnerability patches seem to affect database suites the most. Table 4.11, illustrates the corresponding results. We observe that in-memory database systems such as *Redis* and *Memcached* [6] are the ones experiencing great energy and delay performance degradation. Specifically, the energy consumption of *mcperf* tasks ranges from 25.8% to 29.6%, while its execution time is reduced from 30% to 33.3%. In the case of the *Redis* benchmark tasks, we observed energy savings from 40.8% to 41.3% and increased run-time performance from 44.6% to 45.4%. In contrast to the in-memory database systems, *rocksdb*, *sqlite3*, and *cassandra* experienced energy and run-time performance gains from 3.4% to 5.3%, while *mongodb*'s (*pymongo*) energy consumption was only reduced by 1.8%.

Compute-Intensive: In this category, we first focus on the "green" highlighted cases and then discuss to the rest as we progress.

The *Apache* web server experienced an energy reduction of 18.6% and an increased run-time performance of 23.8% when we disabled the patches. Likewise, *Nginx* server's energy consumption and execution time decreased by 20.2% and 22.5%, respectively. Similarly to the above cases, the *node-express-loadtest* (a NodeJS web server written in the Express framework) achieved energy and run-time performance gains equal to 9.1% and 10%, respectively.

[6]Note that *mcperf* acts as the client of *Memcached*.

Table 4.11: Database Systems Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
cassandra_read	127	15	125	15	124	15	125	15	123	15	3.6	0
cassandra_write	115	13	112	13	112	13	111	13	111	13	3.6	0
mcperf_add	804	30	760	28	665	25	777	29	590	21	26.6	30
mcperf_append	752	29	710	27	632	24	712	27	555	20	26	31
mcperf_delete	550	18	521	17	441	14	508	17	387	12	29.6	33.3
mcperf_get	546	18	503	16	434	14	500	17	386	12	29.2	33.3
mcperf_prepend	748	29	710	27	631	23	711	27	555	20	25.8	31.1
mcperf_replace	747	30	710	28	630	25	720	29	553	20	25.8	31
mcperf_set	802	30	762	28	666	25	766	29	594	21	25.9	30
pymongo	883	38	890	39	866	37	870	38	867	38	1.8	0
redis_get	1972	66	1631	53	1761	58	1685	55	1166	36	40.8	45.4
redis_lpop	1985	65	1629	52	1756	58	1672	54	1167	36	41.1	45.4
redis_lpush	1988	66	1640	53	1763	58	1665	54	1166	36	41.3	45.4
redis_sadd	1981	66	1642	52	1764	58	1680	54	1169	36	40.9	45.5
redis_set	1972	66	1622	53	1761	58	1680	54	1167	36	40.8	44.6
rocksdb_fillrand	1281	31	1271	31	1243	31	1250	31	1212	31	5.3	0
rocksdb_fillseq	976	21	969	21	951	21	960	21	936	20	4.1	4.7
sqlitebench	944	134	962	136	878	125	930	134	908	130	3.8	3

The *brl-cad* benchmark evaluates the performance of various computer system components such as CPU, memory, cache coherency, kernel operations, and compiler. After dispensing the CPU vulnerability patches, the *brl-cad*'s energy and run-time performance increased by 6.4% and 7%, respectively. In the case of *bork*, which involved file encryption, the energy and the run-time performance were affected by 8.3% and 6.6%, respectively. Meanwhile, other benchmarks such as *openssl*, *john-the-ripper*, *botan*, and *blake2s* had less than a percentage of performance degradation after disabling the CPU vulnerability patches. *Glibc* tasks such as $atanh$, sin, $sincos$, $sinh$, $log2$, cos, $sqrt$, $tanh$ indicated energy reductions ranging from 1.8% to 6%. Notably, their execution time remained unaffected. Finally, *renaissance* (a benchmark suite for testing JVM's components, such as the JIT compiler and garbage collector) had energy and delay implications on certain operations, while others remained intact. Specifically, the *reactors* task (a set of message passing workloads encoded in the Reactors framework [141]) experienced 5.5% energy and 9.1% execution time reduction, while the *db shootout* task (a parallel shootout test on a Java in-memory database [141]) had 5.3% energy and 5.2% execution time reduction.

Several of the compute-intensive tasks were affected by less than 5%. Specifically, *blogbench*, a benchmark that uses multiple threads to perform read and write file-system operations, had its energy consumption and execution time reduced by 2.2% and 1.5% on average. *Byte* tasks (used for stressing multi-core systems through synthetic workloads) showed an energy reduction of 1.4%, on average, while their run-time performance was affected by less than a percentage. *DaCapo*, a benchmark suite with a variety of applications to stress computer systems processor and main memory, had 1% to 2.7% energy and 1.8% to 7.1% execution time reductions, respectively. *Cloverleaf*, a benchmark that stresses the CPU by solving Euler equations on a Cartesian grid, experienced 2.9% energy and 3.2% run-time performance gains. *Crafty*, a chess program that tests multi-core processor performance, had energy savings of 1.5%, while its run-time performance remained the

Table 4.12: Compute-Intensive Suites Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
apache	1507	42	1416	39	1370	38	1386	38	1224	32	18.7	23.8
blogbench_read	2435	63	2432	63	2403	63	2431	63	2385	63	2	1.5
blogbench_write	2326	52	2314	52	2285	52	2300	52	2269	52	2.4	0
brl-cad	44792	707	40782	641	40465	639	41321	664	41889	657	6.4	7
bork	256	15	248	15	267	17	251	15	234	14	8.3	6.6
byte_dhry2	1734	122	1730	121	1709	121	1728	121	1715	121	1.1	0.8
byte_float	1541	122	1530	121	1518	121	1521	121	1515	121	1.7	0
byte_int	1536	122	1531	121	1514	121	1525	121	1515	121	1.4	0
byte_register	1543	122	1531	121	1514	121	1536	122	1517	121	1.6	0
cloverleaf	3519	93	3432	90	3403	90	3424	90	3414	90	2.9	3.2
crafty	559	24	556	24	550	24	552	24	550	24	1.5	0
dacapo_eclipse	4168	109	4131	108	4091	108	4119	108	4093	107	1.7	1.8
dacapo_h2	1386	41	1383	41	1370	41	1367	41	1348	40	2.7	2.4
dacapo_tradeso	516	14	511	13	518	14	512	12	506	13	1.9	7.1
fhourstones	2488	117	2484	117	2461	117	2476	117	2437	117	2	0
glibc_ffs	221	10	227	10	220	10	225	10	227	10	2.9	0
glibc_log2	227	10	230	10	225	10	218	10	217	10	4.3	0
glibc_cos	475	21	476	21	471	21	467	21	460	21	3.1	0
glibc_sin	480	21	482	21	474	21	470	21	464	21	3.3	0
glibc_sincos	447	20	448	20	442	20	435	20	439	20	1.8	0
glibc_sinh	242	10	246	10	241	10	232	10	231	10	4.5	0
glibc_sqrt	232	10	240	10	230	10	237	10	242	10	3.9	0
glibc_tanh	235	10	235	10	232	10	223	10	221	10	4.2	0
gmpbench	9235	419	9208	418	9169	418	9203	418	9077	418	1.7	0.2
himeno	1715	73	1716	72	1692	73	1693	72	1672	72	2.4	1.3
nginx	936	31	926	30	848	28	870	28	747	24	20.2	22.5
node-loadtest	343	10	332	10	332	10	330	10	311	9	10	10
octave-bench	277	10	274	10	272	10	267	10	266	9	4.1	10
phpbench	365	15	367	15	362	15	359	15	361	15	1	0
pyperf_2to3	1485	61	1453	60	1467	60	1438	60	1436	60	3.2	1.6
pyperf_float	496	20	490	20	494	20	493	20	488	20	1.7	0
pyperf_go	1085	44	1075	44	1074	44	1070	44	1071	44	1.2	0
pyperf_pathlib	776	32	756	32	757	32	744	32	722	32	6.9	0
pyperf_pickle	704	28	701	28	698	28	698	28	697	28	1.1	0
pyperf_startup	1938	84	1902	82	1923	83	1894	82	1850	80	4.5	4.7
renais_shootout	596	19	614	19	598	19	585	18	565	18	5.3	5.2
renais_reactors	842	22	834	22	853	22	823	21	795	20	5.5	9.1
swet	296	14	294	14	296	14	296	14	289	14	2.1	0
sudokut	500	20	497	20	497	20	502	20	495	20	1	0
tiobench_write	2620	346	2587	342	2488	334	2547	334	2518	335	3.8	3.1
tiobench_read	1678	202	1638	197	1673	206	1686	203	1643	200	2	1

The tasks *renais_shootout* and *renais_reactors* stand for the tasks of *renaissance shoootout* and *reactors* respectively. Also, *node-loadtest* stands for the task *node-express-loadtest*.

same. *Fhourstone* is another CPU-intensive application that had 2% energy reduction, while its run-time performance remained unaffected. *Pyperformance* consists of various compute-intensive tasks such as math, regex, serialise, template libraries, and so on. By turning off the vulnerability patches, *pyperformance* tasks had energy savings ranging from 1% to 7% and an increased run-time performance from 1.6% to 9.3%. *Octave-bench*, a benchmark for numeric computations, had reduced energy consumption and execution time of 4.1% and 10%, respectively. *Gmpbench* aims to assess a processor's performance through integer multiplications; by dispensing the CPU vulnerability patches, it achieved energy savings of 1.7%, while its run-time performance did not change at all. *Himeno* performs a large number of computations to measure CPU performance; our results suggest that the patches can affect its energy and run-time performance up to 2.5% and 1.3%, respectively. *Swet* evaluates a system's performance through several math, logic, bitwise, and branch tests. Such operations had their energy performance degradation by 2.2%, while their run-time performance remained intact.

Table 4.13: Disk Suites Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
aio-stress	715	91	670	86	668	86	668	86	668	88	6.4	3.2
dbench_1	4921	720	4904	720	4898	720	4889	720	4908	720	0.2	0
dbench_6	5072	720	4982	720	5061	720	4986	720	5030	720	0.8	0
dbench_12	5266	720	5279	720	5292	720	5256	720	5206	720	1.1	0
dbench_48	5680	720	5695	720	5698	720	5659	720	5609	720	1.2	0
dbench_128	5919	720	5909	720	5933	720	5859	720	5821	720	1.6	0
dbench_256	6146	721	6136	721	6122	720	6069	721	6014	721	2.1	0
fs-mark_1K	336	47	346	48	339	47	331	47	333	47	0.8	0
fs-mark_4K	1127	155	1111	154	1109	153	1106	153	1111	154	1.4	0.6
fs-mark_5K	2985	398	2954	393	2923	387	2965	396	2957	395	1	0.7
iozone_2096	789	93	753	92	771	92	746	91	717	89	9	4.3
iozone_4096	1939	232	1894	232	1909	231	1888	231	1796	223	7.3	3.8
iozone_8126	5798	678	5733	679	5709	674	5624	670	5426	654	6.4	3.5
postmark	912	109	910	109	907	108	925	112	892	108	2.1	0.9

Disk Suites: We examined four benchmarks in total, that are composed of 13 tasks with different configurations that can stress a filesystem (see Tables 4.13). Our results suggest that CPU vulnerability patches impact filesystem operations. Specifically, after disabling all CPU vulnerability patches, *iozone* tasks' energy consumption was reduced from 6.4% to 9.1%. Also, their run-time performance was increased from 3.5% to 4.3%. *Aio-stress*, a benchmark to test asynchronous I/O operations on files, indicated a reduced energy consumption and execution time, *i.e.*, from 6.4% and 3.2% in both cases. Likewise, *dbench* showed energy savings, (around 3%), while its execution time remained intact. Similarly, *postmark*'s energy consumption was reduced by 2.2%, while its run-time performance remained unaffected. For the *fs-mark* benchmark suite, we observed less than 2% of energy and run-time performance gains.

Kernel Operations: By executing benchmarks that stress various Kernel operations, we ob-

Table 4.14: Kernel Operations Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
ctx_clock	6192	318	3523	174	6178	318	3165	169	1121	56	81.8	82.3
osb_files	659	29	631	27	593	26	622	27	542	23	17.6	20.6
osb_processes	887	22	852	20	889	22	870	21	803	19	9.4	13.6
osb_threads	588	19	549	17	574	19	550	18	494	15	15.9	21
osb_mem_alloc	514	23	503	22	508	23	501	22	480	21	6.6	8.6
osb_programs	584	11	553	10	576	11	546	10	511	9	12.4	18.1
schbench_2	868	30	868	30	860	30	867	30	866	30	0.2	0
schbench_4	1099	30	1099	30	1099	30	1099	30	1099	30	0	0
schbench_8	1113	30	1113	30	1114	30	1113	30	1112	30	0.1	0
stress-ng_fork	1232	24	1185	23	1222	24	1170	22	1108	21	10	12.5
stress-ng_matrix	799	14	759	13	775	13	786	14	752	13	5.9	7.1
stress-ng_msg	1024	22	862	18	988	21	636	13	519	10	49.2	54.5
stress-ng_sem	911	20	919	21	888	20	782	17	812	18	9.3	10
stress-ng_sock	1719	30	1392	28	1591	28	1367	28	1309	26	23.8	13.3
stress-ng_switch	1431	26	1411	25	1329	23	1204	21	1125	19	21.3	26.9
stress-ng_vec	1446	29	914	18	913	18	910	18	911	18	36.9	37.9

Osb is an abbreviation of osbench suite. For the *osbench_(files|processes|threads)* tasks it creates a number of files, processes, and threads.

served major energy and run-time performance gains as summarised in Table 4.14. Overall, our findings suggest that CPU vulnerability patches affect negatively most of the benchmarks' energy consumption and run-time performance.

The *stress-ng* benchmark suite is equipped with tasks to investigate the performance of operations such as fork (*stress-ng_fork*), send/receive messages via the System V message IPC (*stress-ng_msg*), semaphore wait and post operations (*stress-ng_sem*), socket open, send a message, and close (*stress-ng_sock*), context-switch (*stress-ng_switch*) and more. The tasks of context-switch, socket usage, messaging, and vector math operations (*stress-ng_vecmath*) had energy reductions of 21.3%, 23.8%, 49.2%, and 36.9%, while their run-time performance was increased by 26.9%, 13.3%, 54.5%, and 37.9%, respectively. The remaining tasks also experienced energy and run-time performance gains ranging from 5.9% to 10.8% and from 7.1% to 12.5%, respectively.

One of the test cases that is greatly affected by the CPU vulnerability patches is the *ctx-clock*, a micro-benchmark that performs numerous context-switches by invoking system calls. Our results suggest that *ctx-clock*'s energy savings and run-time performance increased by 81.8% and 82.3%, respectively, after disabling all CPU vulnerability patches. The performance impact on the context-switch operations was also visible from the results of the *stress-ng_switch* task.

Similarly to *stress-ng* and *ctx-clock*, *osbench* tasks experienced significant energy and run-time performance gains after disabling all the CPU vulnerability patches. The tasks of program launching,[7] memory allocation, file, processes, and threads creation had an energy and run-time performance boost ranging from 6.5% to 17.6% and 8.6% to 21%, respectively.

Among the test benchmark suites only *schbench*, a benchmarking tool for kernel thread latency, was affected the least from the CPU vulnerability patches (less than 1%).

[7]forking and starting programs using the `execlp` and waiting for them to finish

Table 4.15: Machine Learning Tasks Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
md_con_alexnet	1015	17	1017	17	1012	17	1006	17	1009	17	0.6	0
md_con_all	35053	639	35085	638	35024	639	35148	638	35116	637	0.1	0.3
md_con_gnet	7463	131	7437	131	7452	131	7486	131	7458	131	0	0
md_ip_1d	880	15	883	15	882	15	881	15	879	15	0.1	0
md_ip_all	2325	47	2323	47	2325	47	2347	47	2320	47	0.1	0
md_rnn_training	2107	37	2095	37	2090	37	2107	37	2099	37	0.4	0
rbenchmark	745	26	766	25	742	26	710	25	692	24	7.1	7.7
scikit-learn	1316	46	1335	46	1323	46	1319	45	1315	45	0	2.1

For the benchmark *mkl-dnn* we use the *md* abbreviation in the this Table.

Machine Learning: Our findings in the case of the machine learning category indicate that the patches do not affect neither the performance nor the energy consumption of related tasks. However, the *rbenchmark* was an exception with a 7.1% drop in the energy consumption and a corresponding reduction of its execution time as seen in Table 4.15.

Memory Suites: While examining memory benchmark suites, we have witnessed minor energy and run-time performance gains, that were less than 2% (see Table 4.16). However, for the *t-test1* and *tinymembench*, we observed energy savings of 12.3% and 8.8% respectively. Similarly, the above benchmarks' run-time performance was increased by 12.2% and 0.3%, respectively.

Network Suites: The limited number of results in Table 4.17 did not help on drawing clear conclusions regarding the energy and run-time performance impact of the CPU vulnerability patches have over network operations. Note that the obtained results illustrate that in the case of the *network_loopback*, both the energy and run-time performance were increased by 14.8% and 18.3% after dispensing all patches. Additionally, *iperf*'s test cases showed some minor changes in their energy and run-time performance.

Imaging Benchmarks: Table 4.18, indicates that all tasks had minor energy and run-time performance gains after disabling the CPU vulnerability patches. In particular, the *graphics_magick* tasks (*gm* in the Table 4.18) had energy and run-time performance gains from 0.1% to 4.3% and 0.4% to 4.1%, respectively. *Gegl* tasks experienced energy and run-time performance gains from 0.1% to 2.1% and 1.3% to 10%, correspondingly. Likewise, imaging

Table 4.16: Memory Suites Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
cb_memcpy	578	22	580	22	574	22	575	22	578	22	0	0
cb_memset	602	24	608	24	596	24	601	24	603	24	0	0
cb_mixed	1835	83	1853	83	1814	83	1832	83	1824	83	0.5	0
cb_read	532	27	533	27	528	27	528	27	532	27	0	0
cb_write	576	28	581	28	572	28	578	28	578	28	0	0
mbw_512	206	8	207	8	206	8	204	8	205	8	0.6	0
mbw_1024	414	17	410	17	411	17	406	17	408	17	1.5	0
mbw_4096	1648	70	1636	69	1641	70	1627	69	1627	69	1.2	1.4
rs_add_float	4031	140	4000	140	4014	140	4003	140	3961	140	1.7	0
rs_add_int	3914	141	3920	140	3908	140	3858	140	3882	140	0.8	0.7
rs_copy_float	4010	140	4009	140	3994	140	3986	140	3969	140	1	0
rs_copy_int	3914	141	3918	140	3896	140	3856	140	3882	140	0.8	0.7
rs_scale_float	4029	140	4002	140	4007	140	3996	140	3962	140	1.6	0
rs_scale_int	3921	141	3934	140	3909	141	3854	140	3884	140	0.9	0.7
rs_traid_float	4020	140	4018	140	4012	140	4000	140	3980	140	0.9	0
rs_traid_int	3932	141	3920	140	3917	140	3874	140	3876	140	1.4	0.7
stream	2279	65	2283	65	2272	65	2258	65	2267	65	0.5	0
sysbench_mem	261	5	261	5	258	5	261	5	259	5	1	0
t-test1	241	7	228	7	240	7	225	7	211	6	12.3	14.2
tinymembench	5969	275	5848	275	5950	275	5175	264	5440	270	8.8	0.4

For the benchmark *cachebench* we use the *cb* abbreviation, while for *ramspeed* the *rs*.

Table 4.17: Network Suites Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
iperf_tcp	290	33	296	33	291	33	293	33	291	33	0	0
iperf_udp	295	33	291	33	289	33	292	33	288	33	2.2	0
network_loopback	1488	49	1415	45	1365	45	1398	46	1267	40	14.8	18.3

Table 4.18: Imaging Tasks Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
dt_masskrug	498	12	499	12	496	12	495	12	493	12	1	0
dt_bench	1193	27	1193	27	1184	27	1191	27	1188	27	0.3	0
dt_room	400	9	400	9	399	9	399	9	396	9	1	0
dcraw	486	24	483	23	480	23	484	23	477	23	1.9	4.1
gegl_antialias	1178	47	1179	47	1168	47	1166	46	1166	46	1	2.1
gegl_cartoon	2817	122	2835	122	2780	121	2775	120	2771	120	1.6	1.6
gegl_color	1483	68	1495	68	1459	67	1473	67	1470	67	0.8	1.4
gegl_crop	241	10	244	10	237	9	239	9	241	9	0.8	10
gegl_reflect	791	35	799	35	786	35	792	35	790	35	0.1	0
gegl_rotate	1178	48	1177	48	1170	47	1170	47	1164	47	1.1	2
gegl_scale_size	175	6	176	6	172	6	171	6	174	6	0.1	0
gegl_tile_glass	852	35	858	35	842	35	839	34	834	34	2.1	2.8
gegl_wavelet	1665	73	1678	73	1644	72	1640	72	1636	72	1.7	1.3
gm_gaussian	8651	310	8672	307	8621	306	8627	307	8645	307	0	0.9
gm_minify	1090	48	1091	47	1086	47	1086	47	1076	46	1.2	4.1
gm_resize	2514	71	2510	71	2463	68	2504	70	2499	70	0.6	1.4
gm_rotate	2082	198	2043	195	2042	196	2026	194	1991	193	4.3	2.5
gm_sharpen	18257	482	18272	483	18281	482	18242	481	18236	480	0.1	0.4
inkscape	556	23	551	23	552	23	550	23	544	22	2	4.3
rawtherapee	3969	92	3966	91	3925	90	3949	91	3940	91	0.7	1
rsvg	369	16	360	15	363	16	362	15	353	15	4.2	6.2
tjbench	560	24	560	24	558	24	559	24	559	24	0.2	0

For the benchmark *graphics_magick* we use the *gm* abbreviation, while for *darktable* we use the *dt*.

tools such as *dcraw*, *inkscape*, and *rsvg* had energy savings equal to 2%, 2.1%, and 4.2% and run-time performance increase of 4.1%, 4.3%, and 6.2%, respectively. The benchmarks of *tjbench*, *rawtherapee*, and *darktable* had a minor energy and run-time performance deterioration.

Renderers: Renderers are computer programs that generate images from modeled objects. The results coming from the *Renderers* benchmarks depict that only *qgears* had major energy and run-time performance gains after disabling the CPU vulnerability patches (see Table 4.19). *Qgears* tasks had energy savings of 33.7% and execution time reduction of 23.3%, on average. Also, the tasks of *j2dbench text* and *paraview waveletvolumn* achieved energy gains of 1.1% and 4.3%, respectively. All the other tasks had less than a percentage of energy savings or run-time performance increase.

Desktop Graphics: Table 4.20 presents the results coming from three dimensional gaming benchmarks. All benchmarks ran specific, predefined scenarios. After analysing our findings, we observed minor differences regarding energy and run-time performance (less than 1%). In the case of the *nexuiz* tasks, we saw energy and run-time performance gains of 1.5% and 1.8%, on average.

Table 4.19: Renderer Suites Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
aobench	868	38	868	38	858	38	863	38	861	38	0.7	0
c-ray	9416	188	9437	188	9380	188	9430	188	9398	188	0.2	0
j2dbench_all	13232	361	13153	361	13168	361	13155	361	13207	361	0.1	0
j2dbench_graphics	9078	245	9019	245	9039	245	9039	245	9033	244	0.4	0.4
j2dbench_images	3796	105	3780	105	3780	105	3770	105	3791	105	0.1	0
j2dbench_text	393	14	387	14	392	14	391	14	389	14	1	0
jxrendermark	448	21	445	20	446	21	448	20	446	20	0.4	4.7
ospray_san_miguel	2329	55	2320	55	2343	55	2337	55	2344	55	0.6	0
ospray_xfrog_forest	10166	184	10138	184	10070	183	10111	184	10084	183	0.8	0.5
povray	6296	102	6292	102	6270	102	6309	102	6249	102	0.7	0
pv_manyshperes	7818	360	7842	360	7816	360	7886	362	7782	358	0.4	0.4
pv_waveletcontour	19423	784	19349	781	19408	784	19324	779	19356	779	0.3	0.2
pv_waveletvolumn	1367	31	1372	31	1375	31	1367	31	1308	31	4.3	0
qgears_image	4081	168	3799	162	3767	162	3768	162	3080	144	24.5	14.2
qgears_render	3900	160	3622	155	3593	155	3590	155	2970	140	23.8	12.5
qgears_compo	719	26	391	16	388	16	390	16	216	10	69.9	61.5
qgears_gearsfancy	3887	161	3617	155	3595	144	3590	155	2971	139	23.5	13.6
qgears_text	2595	106	2333	104	2322	104	2330	104	1824	90	29.7	15
rays1bench	1867	33	1862	33	1858	33	1878	33	1857	33	0.5	0
smallpt	1325	25	1327	24	1326	24	1335	25	1322	24	0.2	4
ts_hair	3314	60	3319	60	3292	60	3301	60	3306	60	0.2	0
ts_non_exponential	827	14	831	14	821	14	823	14	822	14	0.6	0
ts_volumetric_caustic	1224	19	1229	19	1215	19	1218	19	1220	19	0.2	0
ts_water_caustic	2134	48	2137	48	2124	48	2133	48	2129	48	0.2	0
ttsiod-renderer	2565	50	2578	50	2561	50	2559	50	2561	50	0.1	0

For the benchmark *tungsten* we use the *ts* abbreviation in this Table, while for *paraview* the *pv*.

4.3.1.2 Communication-related Security

The use of HTTP resulted in 2140 Joules of energy consumption and execution time of 93 seconds, while HTTPS required 4938 Joules and 211 seconds to execute the same scenario.[8] Note that in this case, we used a rather simple scenario to exchange multiple messages between a client and a server. By disabling the HTTPS protocol, we observed that both the energy consumption and the execution time are reduced by 56.66% and 55.92%, respectively. Our results can be explained if we consider the encryption overhead that is introduced during the initial handshake, and to encrypt every exchanged package.

4.3.1.3 Memory-related Protection

We performed an experiment to measure the energy consumption and run-time performance of a C program that allocates 100MiB of memory and utilises the `memset` function to zero-out the associated memory space.

[8] `https://github.com/stefanos1316/resultsForEECSE20/tree/master/secureNetworkCommunications`

Table 4.20: Desktop Graphics Energy (in Joules) and Delay (in seconds) Implications

	Stock		Meltdown		Spectre		MDS		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
glmark2_l	13149	332	13127	332	13023	332	13007	332	13138	332	0.1	0
glmark2_m	13273	332	13247	332	13173	332	13199	332	13273	332	0	0
glmark2_h	13152	332	13120	332	13019	332	13091	332	13124	332	0.2	0
glmark2_u	13456	332	13382	332	13312	332	13361	332	13417	332	0.3	0
nexuiz_l	962	24	943	24	941	24	941	24	943	24	1.9	0
nexuiz_m	1291	30	1275	30	1273	30	1272	30	1275	30	1.2	0
nexuiz_h	2496	53	2466	52	2458	52	2463	52	2466	52	1.2	1.8
nexuiz_u	2499	53	2465	52	2463	52	2461	52	2462	52	1.4	1.8
openarena_l	4992	106	5091	106	5057	108	4989	106	4991	106	0	0
openarena_m	7204	146	7168	146	7175	146	7181	146	7188	146	0.2	0
openarena_h	15811	305	15758	304	15697	304	15706	305	15694	305	0.7	0
openarena_u	3816	84	3807	84	3784	84	3800	84	3805	84	0.2	0
uv_l	10161	233	10181	232	10152	232	10136	232	10168	232	0	0.4
uv_m	10959	244	1099	243	11000	244	10920	243	10928	243	0.2	0.4
uv_h	19942	419	20047	417	20053	417	19978	419	19887	419	0.2	0
uv_u	32239	659	32382	656	32402	655	32276	656	31989	659	0.7	0
uh_l	12751	312	12745	312	12716	312	12704	312	12756	312	0	0
uh_m	13483	315	13439	315	13406	315	13400	315	13439	315	0.3	0
uh_h	20295	440	20361	438	20295	438	20302	438	20274	439	0.1	0.2
uh_u	32239	724	32382	720	32402	720	32276	720	31989	721	0.1	0.4
xonotic_l	45011	890	44993	887	44967	887	44926	886	44996	886	0	0.4
xonotic_m	44925	889	44889	887	44937	886	44938	885	44979	885	0	0.4
xonotic_h	44873	888	44904	887	44916	886	44911	886	44906	885	0	0.3
xonotic_u	44939	887	44992	886	44905	888	44934	886	44913	885	0	0.2

Next to each task we added letters that denote different screen resolutions for the game scenarios. We use the letters *l* (low), *m* (medium), *h* (high), and *u* (ultra-high) for 800x600, 1024x768, 1920x1080, and 2560x1440 resolutions, respectively.

To test the energy implications of memory zeroing, we used `redis-bench` [3] and obtained energy measures for the tasks of `set`, `hset`, `lpush`, and `lrange`. Moreover, we executed the above tasks using the `-d` command-line argument, to set a payload of 100MiB, and measured their energy consumption and run-time performance. Afterwards, we employed the `strace` command-line tool to obtain all the `mmap` system calls associated with memory allocations *i.e.*, unspecified file descriptor and no memory sharing options. Our results suggest that for the `set`, `hset`, `lpush`, and `lrange` operations, the memory zeroing was responsible of 4.5% and 4.55%, on average, of the benchmarks' total energy consumption and execution time, respectively.

Table 4.21: The *Phoronix* [163] Benchmark Suite Used for GCC

Category	Benchmark Suites
Audio Encoding	encode-mp3
Video Encode/Decode	ffmpeg, x264, x265
File Compression	compress-bzip2, compress-xz, lzbench, compress-zstd
Database Suite	rocksdb, sqlite, mcperf, redis
CPU Massive	nginx, apache, blogbench, m-queens, sysbench, byte, cloverleaf, tiobench, botan, xsbench, swet, john-the-ripper, himeno, ebizzy, cpp-perf-bench, brl-cad, blake2s, openssl, hmmer, nero2d, gmpbench, fhourstones, stockfish, crafty, aircrack-ng, hpcg
Disk Suite	postmark, iozone, fs-mark, dbench
Kernel	osbench, stress-ng, schbench, ctx-clock
Machine Learning	mkl-dnn
Memory Suite	stream, tinymembench, t-test1, mbw, ramspeed, cachebench
Networking Suite	iperf
Imaging	dcraw, tjbench, graphics-magick
Renderers	smallpt, tungsten, ttsiod-renderer, c-ray, aobench, povray, qgears, jxrendermark

4.3.1.4 Compiler-related Safeguards

To analyse the energy and run-time performance implications of compiler-related safeguards, we used the GCC toolkit. From the benchmarks defined in Section 3, we used only the benchmarks written in C and C++. To this end, we tested 73 out of the 128 benchmarks (see Table 4.21). We use the terms *Stock* and *AllOff* to refer to the scenarios where all the investigated GCC safeguards are enabled and disabled, respectively. Apart from the above two scenarios, we also examine the energy and delay implications of GCC safeguards such as (1) `-fno-stack-protector` (*Stack Protector* in Table 4.22) that disables the protection against stack smashing attacks for all functions, (2) `-z execstack` (*Execstack* in Table 4.22) that refers to the scenario where binaries are allowed to run in the stack, (3) `-no-pie` and `-fno-pic` (PIE & PIC in Table 4.22) which define the scenario where binaries, dependencies, and code are not randomized in memory, (4) `-Wl,-z,norelro` (RELRO in Table 4.22) that dispenses protection against Global Offset Table (GOT) overwrite attacks, and (5) `-U_FORTIFY_SOURCE` (*Fortify Source* in Table 4.22) that defines the scenario where buffer overflow checks for memory and string functions are disabled.

Table 4.22 illustrates the tasks affected the most from the examined GCC safeguards,

Table 4.22: GCC-related Safeguards Energy (in Joules) and Delay (in seconds) Implications

	Stock		Stack Protector		Execstack		PIE & PIC		RELRO		Fortify Source		AllOff		Difference (%)	
Tasks	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time
aio-stress	996	141	1001	141	911	131	1039	136	971	137	988	140	974	138	2.2	2.1
blogbench_write	1842	52	1843	52	1761	51	1830	52	1812	54	1869	52	1764	51	4.2	1.9
botan_aes	373	22	376	22	379	22	374	22	378	22	374	22	369	22	1.3	0
botan_blowfish	354	20	349	20	357	20	357	20	356	20	358	20	344	20	2.8	0
botan_twofish	374	20	371	20	376	20	374	20	376	20	375	20	366	20	2.1	0
byte_register	1373	121	1382	121	1373	121	1295	121	1381	121	1378	121	1273	121	7.2	0
byte_int	1376	121	1379	121	1372	121	1292	121	1374	121	1376	121	1271	121	7.6	0
byte_float	1376	121	1374	121	1371	121	1295	121	1380	121	1379	121	1274	121	7.4	0
bzip2	377	17	336	15	377	17	334	15	378	17	337	15	332	15	11.9	11.7
dcraw	429	25	436	25	429	24	428	24	430	24	428	24	421	24	1.8	4
gm_gaussian	24412	619	23289	610	24386	624	24523	618	24419	617	24302	623	23450	622	3.9	0.4
gm_sharpen	51607	1209	48798	1202	51298	1212	51035	1209	50949	1214	53957	1206	48903	1213	5.2	0.3
gm_resize	5747	141	5703	139	5736	141	5771	140	5775	141	5759	142	5640	141	1.8	0
gm_rotate	2158	229	2007	180	2009	186	2003	178	2169	229	2050	192	2063	192	4.4	17
hmmer	1252	24	1240	24	1248	24	1279	25	1254	25	1247	24	1208	24	3.5	0
lzbench_exz	11334	635	10959	621	11247	630	11191	636	11327	632	11293	633	10884	621	3.9	1.8
postmark	1050	138	1066	141	1002	132	1137	148	1022	134	1051	138	1018	133	3	3.6
povray	5772	105	5634	102	5801	105	5768	104	5832	105	5832	104	5633	102	2.4	2.8
rocksdb_fillrandom	1304	41	1263	41	1272	41	1314	41	1303	41	1274	41	1256	41	3.6	0
rocksdb_readrandom	1237	24	1136	23	1208	24	1244	25	1268	25	1208	24	1161	23	6.1	4.1
schbench_2	798	30	779	30	793	30	803	30	805	30	796	30	777	30	2.6	0
schbench_4	1018	30	998	30	1020	30	1024	30	1025	30	1024	30	993	30	2.4	0
schbench_8	1033	30	1011	30	1037	30	1041	30	1040	30	1030	30	1010	30	2.2	0
sqlitebench	1148	171	1101	166	1075	163	1092	166	1149	171	1121	170	1074	162	6.4	6.3
sysbench_cpu	1013	23	884	21	1027	23	895	22	1025	23	1017	23	883	22	12.8	4.3
tjbench	501	24	485	24	499	24	505	24	500	24	494	24	484	24	3.3	0
ts_hair	3130	60	3040	59	3121	60	3137	60	3131	60	3134	60	3007	59	3.9	1.6
ts_caustic	1998	48	2007	48	2044	48	1990	48	1990	48	1990	48	1953	47	2.2	2
ts_non_exponential	782	14	751	14	781	14	793	15	780	14	783	14	755	14	3.4	0
ts_volumetric	1123	20	1072	19	1123	20	1131	20	1122	20	1122	20	1078	19	4	5
tiobench_read	2328	311	2232	305	2262	310	2350	313	2284	305	2211	302	2217	301	4.7	3.2
tiobench_random_r	2274	311	2197	307	2208	310	2291	306	2278	310	2173	305	2175	304	4.3	2.2
tiobench_random_w	3850	557	3774	552	3780	555	3854	547	3832	551	3751	549	3743	546	2.7	1.9
xsbench	3072	81	2942	79	3087	81	2958	79	3079	81	3072	81	33774	79	3.4	2.4

For the benchmark *tungsten* we use the *ts* abbreviation in this Table, while for *graphics-magic* the *gm*. For *cpp_stepanov_abstraction* we use the *cpp_sa*, for *cpp_stepanov_vector* the *cpp_sv*, for *cpp_random_numbers* the *cpp_rn*.

while the colored rows denote the scale of the energy savings as defined in Section 4.3.1.1. In this section, we depict and discuss only the tasks with energy reduction over 1% after disabling all the security flags (*AllOff* scenario). Because of the limited number of tasks for some categories, we depict our measurements in a single table. However, all the raw values of the remaining tasks are publicly available in our repository.[9] Moreover, the category of desktop graphics did not have tasks written in C or C++; therefore, such a category is missing from this section. Also, we excluded the compilation tasks because GCC safeguards take effect at run-time and not at compile time.

Unaffected Categories. The tasks from the categories of *machine learning*, *memory benchmarks*, *network suites*, *video* and *audio encoding* are mostly unaffected in the context of this experiment.

File Compression. For this category, *bzip2* experienced energy savings equal to 11.9% and increased run-time performance of 11.7%. Also, *lzbench xz* and *libdeflate* tasks had average energy savings of 4.7% and reduced execution time of 2.2%, while the remaining tasks had less than a percentage of performance deterioration after disabling the associated GCC safeguards

Database Systems. Regarding database systems, only *rocksdb*'s and *sqlitebench*'s performance were affected. Specifically, *sqlitebench* exhibited energy savings of 6.4% and reduced execution time of 6.3% after dispensing all GCC safeguards. *Rocksdb* tasks experienced on average 4.8% of energy savings and 2.1% of reduced execution time. For the remaining systems, we observed a performance impact less than 1%.

Compute-Intensive. There were three compute-intensive benchmarks that were affected mostly in this experiment. In particular, *Sysbench*, a multi-threaded benchmark tool based on LuaJIT, introduced the most energy and run-time performance savings (12.3% and 4.3%, respectively). Also, *byte*, a benchmark used for stressing multi-core systems through synthetic workloads, had on average energy savings of 7.6%, while its tasks' execution time remained the same. The tasks of *cpp-performance-bench* (*cpp* abbreviation in Table 4.22)—a tool that benchmarks different C++ features, operations, and data structures—experienced on average energy savings of 1.3%. *Tiobench*, a multi-threaded I/O benchmark, had on average energy savings of 3.6% and increased run-time performance of 2.1%. Lastly, the benchmarks of *cloverleaf*, *hmmer*, *botan*, *blogbench*, and *xsbench* achieved on average energy savings of 3% and less than a percentage of run-time performance increase.

Kernel Operations. We found that only *schbench* was affected after dispensing the corresponding GCC safeguards, while the remaining benchmark suites had minor performance changes. Among the *osbench* tasks, only the *file creation* had minor performance gains after turning off the safeguards.

Disk Suites. There were two disk suites that produced interesting results. The benchmark of *aio-stress* had energy savings and run-time performance increase of 2.2% and 2.1%, respectively. Similarly, the energy and run-time performance of *Postmark* increased by 3%

[9] https://github.com/stefanos1316/resultsForEECSE20/tree/master/gcc

and 3.6%, respectively.

Imaging. The investigated benchmarks for the corresponding category show that *dcraw*, *tjbench*, and three out of the five examined tasks of *graphics-magick* (*gm* in Table 4.22) had energy savings ranging from 1.8% to 5.2% and zero to less than a percentage of run-time performance increase after dispensing the associated GCC safeguards.

Renderers. We examined four renderer benchmark suites. Our results suggest that three out of four benchmark suites were affected. Specifically, the tasks of *tungsten* (*ts* in Table 4.22) achieved 3.3% and 2.1% of energy reduction and run-time performance increase, on average. Also, the benchmarks of *c-ray* and *povray* had on average 2.2% of energy and 1.2% of run-time performance increase. There were no performance implications in the case of the *qgreas* benchmark.

Our analysis shows that the CPU vulnerability patches have a large energy and run-time performance impact on real-world applications. Notably, programs such as Apache, Nginx, Redis, and Memcached had energy and run-time performance gains from 18% to 45% after dispensing the CPU vulnerability patches. Similarly, GCC safeguards affect the energy and run-time performance of applications such as bzip2 and Sysbench by more than 10%. We observed similar results in the case of memory and communications-related security mechanisms too.

4.3.2 SG-RQ2. How do security mechanisms affect the energy consumption and the run-time performance of different applications and utilities?

We focus on the impact of security mechanisms on applications and utilities. Note that we do not discuss memory zeroing and encrypted network communications because they affect specific application types. In particular, encrypted network communications affect only web and application servers, while the memory zeroing impacts memory-intensive applications.

4.3.2.1 CPU-related Vulnerability Patches

The categories with the highest energy and run-time performance degradation from CPU vulnerability patches include code compilation, database systems, compute-intensive tasks, kernel operations, and disk I/O. In other cases, such as video and audio encoding, graphic suites, memory operations, and machine learning, the impact was not as high.

Specifically, for the category of databases, the in-memory database systems such as *Redis* and *Memcached* were affected the most. Also, web servers such as *Apache*, *Nginx*, and *node-express* had an important energy and run-time performance deterioration among the compute-intensive benchmarks category. From the kernel benchmark suites, the majority of tests were heavily affected by the CPU vulnerability patches. For example, operations such as *context-switches*, *message* IPC, *fork*, *memory allocation*, *threads*, *processes*, and *files creation* had a major energy and run-time performance deterioration. Last, the benchmark category of filesystem usage also showed that most of its applications were affected by the CPU vulnerability patches.

From the investigated CPU-related vulnerability patches, the patch of MDS tends to be on average the most energy-hungry, while the *Spectre* patch follows. Finally the *Meltdown* patch consumes the least energy among them. Regarding run-time performance, the MDS vulnerability patch introduces the most run-time performance deterioration, while *Meltdown* follows, and *Spectre* causes the lowest run-time performance degradation. However, the results do not suggest that a single CPU vulnerability patch is more performance-inefficient than the others. For instance, after disabling the *Meltdown* and MDS patches, we experienced more energy savings for the *redis* benchmark compared to *Spectre* (see Table 4.11). However, after disabling the *Spectre* patch for the *mcperf* benchmark, we witnessed more energy savings compared to *Meltdown* and MDS (see Table 4.11).

4.3.2.2 Compiler-related Safeguards

The GCC safeguards introduced performance degradation to most of the investigated categories. However, specific cases including *compute-intensive*, *database*, and *file compression* benchmarks seem to be affected more than the others. Likewise, three out four examined benchmarks from the *renderers* category illustrated energy savings after dispensing the compiler security flags. Similarly, most of the investigated applications from the *imaging* category indicated a better energy and run-time performance after disabling all the safeguards.

Focusing on the corresponding mechanisms, we observed that compiling with the `-fno-stack-protector` GCC safeguard led to the highest energy savings and increased run-time performance. By using the `-no-pie` and `-fno-pic` GCC safeguards, we had the second most energy and run-time performance gains. Then the flag of `-U_FORTIFY_SOURCE` had the third most energy and run-time performance gains, while the `-z,norelro` followed. Lastly, the `-z execstack` contributed least to the energy and run-time performance gains among the examined GCC safeguards.

By analysing our results, we point out that applications found under the categories of database systems, code compilation, compute-intensive, kernel operations, disk usage had the highest energy and run-time performance degradation from the CPU vulnerability patches. For the GCC safeguards, application types such as compute-intensive, databases systems, and file compression had the highest energy and run-time performance gains after disabling the associated security flags.

4.3.3 SG-RQ3. Is the energy consumption of the examined security mechanisms proportional to their run-time performance?

Several researchers [177, 39] have observed that the energy consumption of a computer platform tends to be proportional to its execution time. However, many others have provided opposing results [128, 58, 87, 135, 169]. To evaluate whether the above statement stands in the context of our study, we checked the tasks' energy consumption while dispensing security mechanisms and examined the way their run-time performance fluctuates.

4.3.3.1 CPU-related Vulnerability Patches

In the following, we offer an overview of the categories that illustrate results where energy consumption is not proportional to the run-time performance and mention benchmarks and scenarios that highlight the above fact.

Code Compilation. For this category, we observed that energy consumption is proportional to run-time performance for most of the tasks (see Table 4.9). However, it was not the case for the Linux kernel compilation for the scenarios of MDS and *AllOff*. Although MDS had 3 seconds lower execution time than the *AllOff* scenario, its energy consumption was higher by 469 Joules.

Compute-Intensive: In this category, we found cases where faster execution time is not associated with lower energy consumption (see Table 4.12). For instance, *brl-cad AllOff* had 1% faster execution time than in MDS scenarios, but MDS had 1.4% more energy savings than *AllOff* scenario. Likewise, for *dacapo_tradesoap*, MDS was 7.6% faster than the *AllOff* scenario. Nevertheless, *AllOff* was 1.2% more energy-efficient than the MDS scenario. For the *node-express-loadtest*, the execution time of the *Meltdown*, *Spectre*, and MDS was the same with the *Stock* scenario (*i.e.*, 10 seconds); yet, their energy consumption was on average 3.5% lower than the *Stock* scenario.

Kernel Operations: For the *stress-ng_sem* task (see Table 4.14), the scenarios of *Stock* and *Spectre* had the same execution time (*i.e.*, 20 seconds). Nevertheless, *Spectre* was 23 Joules more energy-efficient. For the *stress-ng_sock* task, the scenarios of *Meltdown* and MDS shared the same execution time with *Spectre* (*i.e.*, 28 seconds); yet, *Meltdown* and MDS were 10% more energy-efficient viz-a-viz the *Spectre* scenario. Regarding the *thread creation* task of the *osbench* benchmark, *Meltdown* had 5.5% lower execution time than the MDS scenario, but *Meltdown* was just 0.2% more energy-efficient compared to the MDS scenario. This fact illustrates that lower execution time is not always associated with lower energy consumption.

Memory Suites: We found few cases where energy consumption is not proportional to run-time performance. In particular, for the *t-test1* benchmark (see Table 4.16), the *Meltdown* and MDS scenarios had the same execution time as the *Stock* one (*i.e.*, 7 seconds), but *Meltdown* and MDS were 5.2% and 6.4% more energy-efficient than the *Stock* scenario, respectively.

Network Suites: For the *network_loopback* benchmark (see Table 4.17), the *Meltdown* and *Spectre* scenarios both took 45 seconds to run, but *Spectre* was 3.5% more energy-efficient. Moreover, for the same task, *Meltdown* was 2.1% faster than the MDS scenario, while the MDS scenario was 1.2% more energy-efficient than *Meltdown*.

Imaging Benchmarks: We mostly found that energy consumption is proportional to the run-time performance of the corresponding tasks (see Table 4.18). However, for the *rsvg* task, the MDS was 6.2% faster than the *Spectre* scenario, but *Spectre* was only 0.2% more energy-inefficient than the MDS scenario. Likewise, for *gegl_crop* task, *AllOff* was 10% faster than the *Meltdown* scenario, but *AllOff* was only 1.2% more energy efficient than the *Meltdown* scenario. This fact demonstrates again that run-time performance gains are not equal to energy savings.

Renderers: The *AllOff* scenario for the *jxrendermark* task had the same energy consumption with *Spectre* (*i.e.*, 446 Joules), but the *AllOff* scenario was 4.7% faster. Similarly, *AllOff* for the *paraview_waveletvolumn* task had the same execution time with the *Spectre* scenario (*i.e.*, 31 seconds), but was 4.8% more energy-efficient.

To sum up, we found that not all cases have energy consumption proportional to their run-time performance in the context of CPU-related vulnerability patches, we demonstrated cases where run-time performance gains are not tightly associated with energy savings.

4.3.3.2 Communication-related Security

According to the results in Section 4.3.1.2, we found that the energy consumption of the corresponding tasks is proportional to their run-time performance.

4.3.3.3 Memory-related Protection

To find out the impact of memory zeroing, we first measured the energy cost for zeroing out a specific size of memory. Then, we estimated the percentage cost that memory zeroing may have on *Redis* tasks. Because our results are based on estimations, for these security mechanisms, we cannot be certain if the energy consumption of the *Redis* tasks is proportional to their run-time performance.

4.3.3.4 Compiler-related Safeguards

In Table 4.22, for the *aio-stress* task, the PIE & PIC was 1.4% faster than the *AllOff* scenario, but the *AllOff* scenario was 6.2% more energy-efficient. Similarly, the *Fortify Source* for *blogbench_write* task was 3.7% faster than RELRO scenario, but RELRO scenario had 2% more energy savings. The *byte*'s tasks energy was reduced by 7.6% on average after disabling all the corresponding GCC safeguards, but the tasks' execution time remained unaffected. For the *graphics-magick_sharpen* task (*gm_sharpen* in Table 4.22) the RELRO scenario execution time was 8 seconds slower than *Fortify Source*, but *Fortify Source* was 5.9% more energy-efficient. For the *sysbench*, the *Stack Protector* scenario was 4.5% faster than *AllOff*, but both scenarios had the same energy consumption.

In conclusion, we find that the energy consumption tends to be proportional to the run-time performance for the majority of the examined tasks. However, we present many cases where they are not proportional.

Our findings suggest that energy consumption is not always proportional to run-time performance for all the investigated benchmarks in the case of the CPU vulnerability patches and the GCC safeguards. However, there are several exceptions. Therefore, we cannot clearly conclude that an application's energy consumption depends solely on its run-time performance.

4.3.4 Discussion

In this section, we discuss the factors that affect our results regarding the diverse security mechanisms.

4.3.4.1 CPU Vulnerability Patches and GCC Safeguards

According to the related work [147], the main reason of performance decline caused by the CPU vulnerability patches is the flushing of buffers after context-switching. Specifically, the flushing occurs in order to clean stale data from buffers to avoid data leakage through side-channel attacks. By using the Linux `time` command, we obtained measurements regarding the time that each benchmark spent inside the user- and kernel-space.[10] Our results suggest that the benchmarks with the highest performance deterioration, most of the times, were the ones that spent more time in kernel-space. This shows that the specific benchmarks invoked many system calls which cause context-switches and thus buffers flushing.

Apart from the performance cost of flushing buffers, there is another reason that may cause performance degradation to our benchmarks. The *Meltdown* vulnerability patch, is flushing the Translation Lookaside Buffers (TLBs) for each switching between the kernel- and user-space. Note that flushing TLBs, increases the number of TLB load misses [147, 29]. This, in turn, leads to a severe performance degradation due to expensive memory accesses, which result in an additional energy and run-time performance overhead. Regarding the Spectre vulnerability, Intel has updated the processors' microcode to flush out the Branch Target Buffers (BTB) after a branch misprediction. Flushing out BTBs reduces drastically the prediction rate of branches. Therefore, the CPU is forced to discard data for the associated mispredictions and waste resources executing unneeded instructions [147, 29]. In a similar manner, MDS vulnerability patch flushes out stale data from different CPU buffers (*i.e.*, line-fill buffers, store buffers, and load ports) that may have been loaded from previous operations [29].

We found that the associated GCC safeguards can affect many of the examined benchmarks. However, compared to the CPU vulnerability patches, GCC safeguards seem to have a much lower overall energy and run-time performance impact. Yet, we have found that the guard stack can affect significantly the energy consumption and run-time performance of selected applications. This stems from the fact that the corresponding tasks (affected the most from the guard stack) invoked many functions with guard stack protection which perform checks against buffer overflow attacks. This resulted in energy and run-time performance degradation for the associated tasks.

[10] `https://github.com/stefanos1316/resultsForEECSE20/tree/master/mitigations/kernelspace`

4.3.4.2 Associating Energy Consumption with Run-Time Performance

A task's energy consumption is the amount of energy, measured in *Joules*, required by a computer system to accomplish it, it is calculated by the formula $E = P \times T$, where P denotes the power consumption, in *Watts*, and T the total amount of time, in *seconds*, required to execute a task. Therefore, to achieve energy efficiency one might think that reducing the execution time of a computer task is enough. However, such a statement is only valid when power consumption is constant. Even though there are several studies related to computer systems and energy efficiency, it is still unclear how certain design decisions can alter the energy consumption of computer programs [37], or what their trade-offs are [94, 117]. As computer systems change and become more complex with an evolving memory hierarchy, processors of multiple cores and distinct power states, their power requirements (of such hardware components) begin to vary among the different memory hierarchy or processor's power states. Therefore, it is expected that the energy consumption of an applications to not be proportional to its execution time.

Chapter 5

Conclusions and Future Work

This thesis presents ways to achieve energy and run-time performance by making the proper selection for developing an application. In this chapter, we discuss this thesis contributions and future research directions.

5.1 Contributions of the Thesis

In this section, we summarise the contributions offered by this thesis. We first discuss the research- and then the development-oriented contributions for each of our studies.

In our survey study, we identified and discussed several research challenges. Such challenges could serve as a starting point to further improve the field of energy efficiency in software development. For developers, we point out available tools, with their features and limitations, that can be used to measure an application's energy consumption. Moreover, we discuss techniques and practices that can help reduce an application's energy consumption.

By investigating different programming languages, we identified which ones can contribute to better energy and run-time performance for specific programming tasks. We also examined how our results vary among different computer platforms and showed that there is no single winner for all cases. For developing purposes, we offer a programming language-based ranking catalogue of nine heatmaps [1] (three for each of server, laptop, and embedded system) showing the EDP implications of 14 programming languages against 25 generic programming tasks from Rosetta Code. These heatmaps could act as a guide on programming language selection for software practitioners seeking to gain energy and run-time performance when developing on a particular computer platform. Moreover, we offer publicly the data-set of the 25 tasks written in 14 programming languages along with the execution scripts that can be used as benchmarks for similar studies. Nevertheless, we advise developers in using the following:

- C for the development of computationally-intensive and concurrent applications
- Go for memory-intensive program that involves a lot of sorting
- Rust for I/O-intensive computer applications
- JavaScript for tasks that involve heavy string processing based on regular expressions

[1]see Appendix

- C++ for functional programming tasks

Furthermore, we performed an empirical study over diverse remote IPC technologies implemented in different programming languages to appraise their energy and run-time performance. Our results highlight JavaScript's and Go's implementations as the most energy- and run-time performance-efficient compared to PHP, Java, C#, Python, and Ruby. Moreover, we found that certain web-frameworks overuse system resources (through system calls) which implies reduced performance. Nevertheless, we have shown that the energy and run-time performance of the investigated IPC technologies are not necessarily proportional. Researchers can build on our study by comparing different web frameworks, using additional test cases, different database systems, and applying our findings in a real-world micro-services applications to evaluate their performance. Software practitioners can benefit from our study by noting the following.

- Energy consumption and run-time performance can vary significantly among different programming language implementations; therefore, making the right selection of IPC can benefit the applications' energy consumption.
- Neither the memory usage nor the number of context-switches can indicate the energy or run-time performance of a library's efficiency.

We studied how several security mechanisms affect the energy and run-time performance of multiple applications and utilities. Our results suggest that security mechanisms do not come for free because they use up a considerable amount of energy and degrade run-time performance. Specifically, we found that cpu vulnerability patches increase the energy consumption and execution time of many real-word applications such as *Apache*, *Nginx*, and *Redis*. Likewise, the use of http, instead of https in a secure environment, can reduce energy consumption and execution time by 56.66% and 55.92%, respectively. Similarly, doing away the memory zeroing can decrease the energy and increase the run-time performance of memory-bound applications up to 4.5%, on average. Finally, disabling compiler safeguards can lower the energy consumption of compute-intensive, imaging, and rendering benchmarks. Moreover, we found that the energy consumption is not always proportional to run-time performance.

To this end, we believe that many applications can benefit from energy savings and increased run-time performance by turning off diverse security mechanisms. However, this approach is only appropriate in a suitably sanitized, secure, sterile, and monitored environment, where activity by malicious users is an unlikely possibility. Creating such environments is difficult and expensive, but not impossible. For many good reasons organizations running large cloud data centers are already operating their core computing infrastructures inside secure perimeters [78, 79]. Given their huge scale, these same facilities are also excellent targets for achieving substantial energy savings and reducing CO_2 emissions by adopting the measures we propose.

5.2 Thesis Critiques

In this section, we discuss some of the weak points of this thesis in order to help researchers to set their focus on the right path. We first discuss the work on the various programming languages, then the remote IPC technologies study, and, finally, the safeguards research.

In Section 4.1, we discussed that different programming language implementations can affect computer task's energy and run-time performance in different ways. However, in our research, we compared implementations of computer tasks written solely in a specific programming language. For instance, we compared the energy and run-time performance of a compute-intensive task written in Python against the other implementations. Nevertheless, when performing compute-intensive tasks in Python, developers write such tasks in C to reduce the tasks' execution time. Also, real-world applications are usually developed by multiple programming languages. Therefore, it would make more sense to compare the energy and run-time performance of tasks that combine multiple programming languages.

In Section 4.2, we presented a study on popular remote IPC technologies where we appraised their energy and run-time performance. Moreover, we discussed that different IPC technologies affect the energy and run-time performance of Intel and ARM processors in different ways. In our experiments, we used a very simple test case to compare the performance of the selected IPC technologies. However, modern web-services and websites are developed by multiple frameworks, modules, programming languages, and software systems (*e.g.*, in-memory databases, database systems, servers). Therefore, to better understand the performance implications between different technologies, more tests are needed. Moreover, as discussed in the previous paragraph, more testing with applications that combine different programming languages, modules, and software systems are required in order to have a better picture of the energy and run-time performance implications of different technologies.

In Section 4.3, we explained the energy and run-time performance implications of security mechanisms on different application types (*e.g.*, im-memory database, kernel operations, server systems). In our experiments, we used a server computer to test the effects of security mechanisms. However, most companies host their applications on data centers where these applications run inside virtual machines. Therefore, we believe it is also important to run our experiments on virtual machines in order to investigate further the energy and run-time performance impact of security mechanisms. Moreover, we performed our experiments on a single OS (*i.e.*, Ubuntu). Nevertheless, data centers house more than one OS. To this end, we think it is also critical to see how much the energy and run-time performance of applications are affected by security mechanisms across different OSs.

5.3 Future Work

The findings presented in this thesis open new research directions in the context of energy-efficient practices in software engineering. The opportunities for future work, that we have identified, focus on four directions: (1) define rules to perform empirical studies for mobile applications energy efficiency, (2) energy and run-time performance evolution of Linux core tools and system calls, (3) investigation of software architecture styles energy requirements, and (4) search-based software engineering for database systems data structures.

When performing empirical studies to examine the energy consumption in software engineering, many researchers follow different approaches suggested by others or personal experience. However, the same guidelines do not apply to all cases (Cloud, IoT, controlled environment, etc.) or computer systems (workstation, server, smart-phone, embedded, and so on). To this end, we present a number of research questions that may formulate an empirical study regarding energy efficiency in the context of mobile applications. Note that the questions below can be also asked for different computer platforms.

- What are the publication trends of empirical studies on the energy consumption of mobile apps?
- What is the state of the art in relation to rigor and relevance of empirical studies on the energy consumption of mobile apps?
- Which metrics are considered when conducting empirical studies on the energy consumption of mobile apps?
- Which are the appropriate guidelines for researchers when conducting empirical studies on the energy consumption of mobile apps?
- What are the main emerging challenges for future empirical research on energy consumption of mobile apps?

Unix-like OS and its kernel have evolved tremendously over the years and it is considered among the most successful software. A study to appraise the energy and run-time implications of the Linux's core operations and tools to point out the reasons behind their fluctuations during the evolution of its software could help OS developers and software practitioners to better understand the energy consumption of their design or implementation decisions. Such a study is important because, by identifying energy hogs in OS and solving them, it directly impacts the energy consumption of many computer systems.

- What are the energy and run-time performance implications regarding the evolution of in Unix-line OS tools over the years?
- What are the energy and run-time performance implications regarding the evolution of in Unix-line OS system calls over the years?

Search-Based Software Engineering (SBSE) is a well-established research area that helps in solving effectively software engineering problems by using search and optimisation algorithms. However, the selection of appropriate data structures and the resulting energy efficiency improvements is a topic that has not been studied in the context of SBSE. For instance, most cyber-physical systems and cloud computing applications make use of database systems to retrieve data. In order to offer high performance, the database systems are using indexes to fetch data without searching all table rows. An application of SBSE here could help to point out which data structures to use in a database system's indexing mechanism in order to increase its energy savings. However, this can be also applied in a variety of other applications.

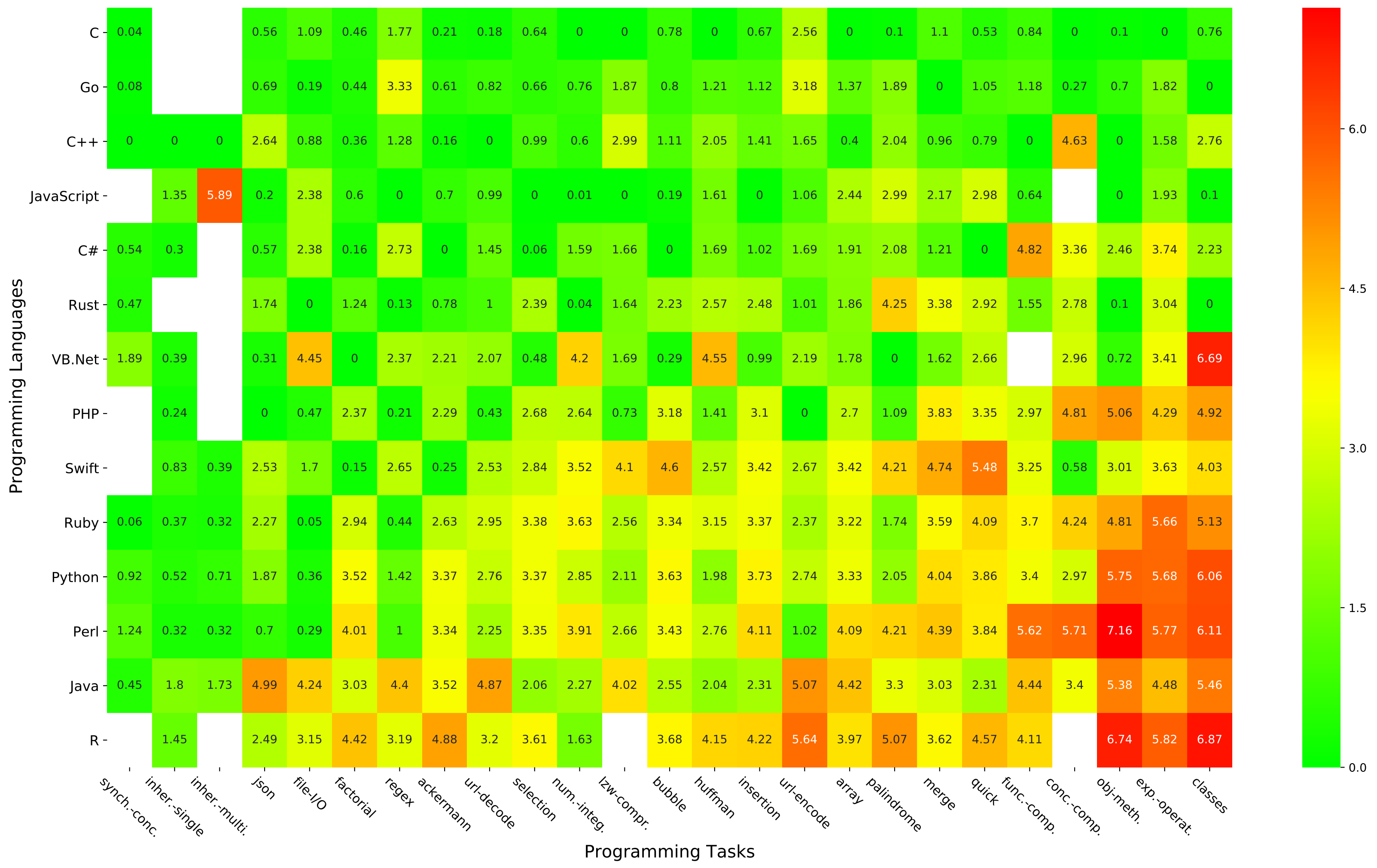
- What are the energy and run-time performance gains of different data structures in the context of database systems indexing?
- Can the same data structures offer energy and run-time performance gains on different database systems?

Selecting the suitable software architecture styles (SAS) for a project mainly depends on its requirements. However, there is very little knowledge regarding the energy consumption of different SAS. Micro-services SAS is considered to be the silver bullet for designing software. Although many large companies prefer to use micro-services to make their software scalable, reliable, polyglot, and so on, this may increase energy consumption due to the increased use of the network layer for services communication. However, many other SAS may offer reduced energy consumption and even better run-time performance. Therefore, investigating various SAS and their trade-off could help us suggest solutions, or even make changes in some of the most popular SAS to increase their energy and run-time performance. Moreover, such research could help to reduce the energy consumption of applications running in the cloud, IoT infrastructures, or hyper-physical systems if adopted in the early stage of the software development life cycle.

- Which are the most energy- and run-time performance-efficient among the selected SAS?
- Which changes can help in reducing the energy consumption and execution time of the selected SAS?
- Which are the reasons that make specific SAS more energy and run-time performance-efficient than others?

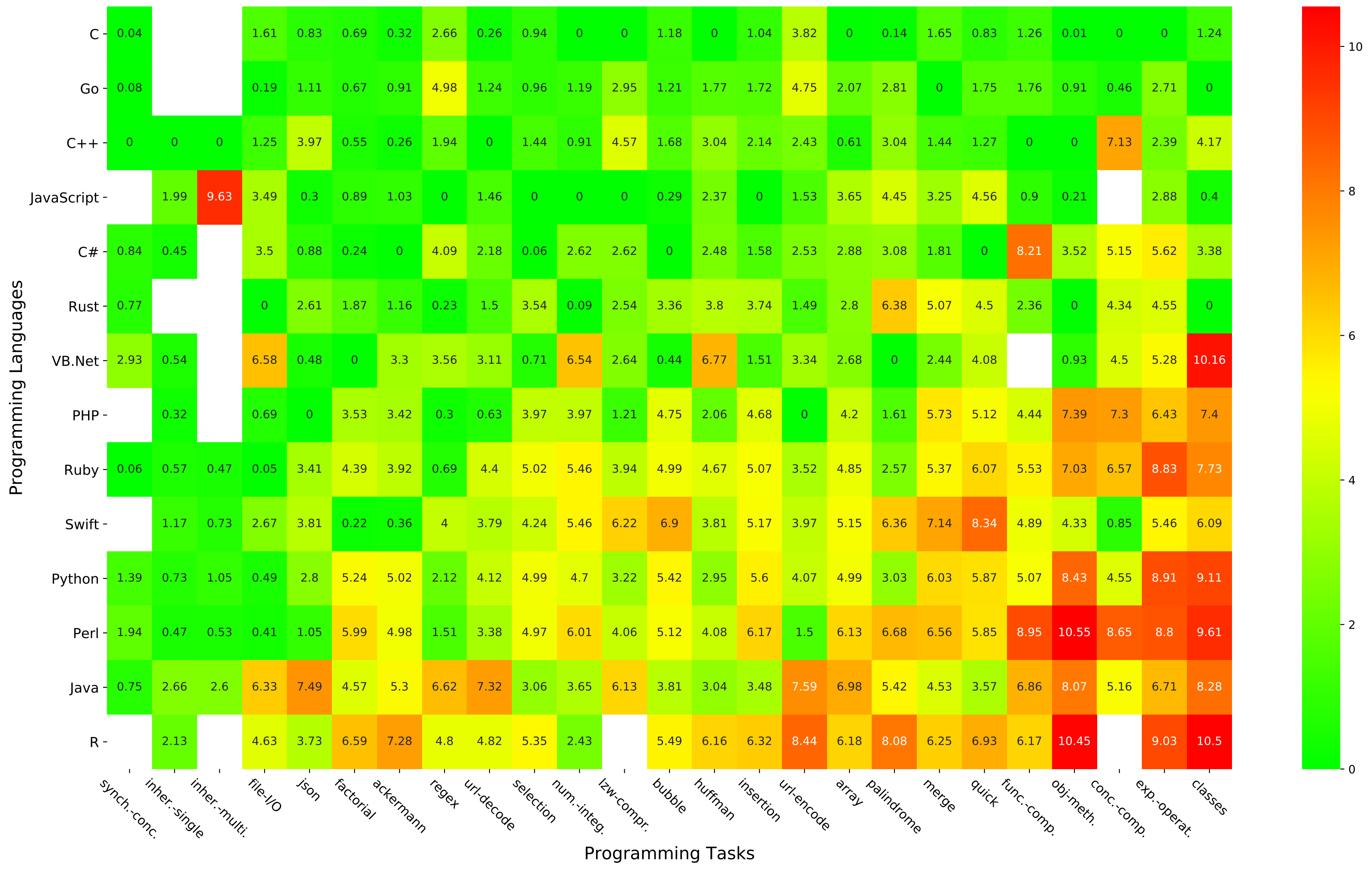
Part A

Appendix I: Heatmaps from Programming Languages Study



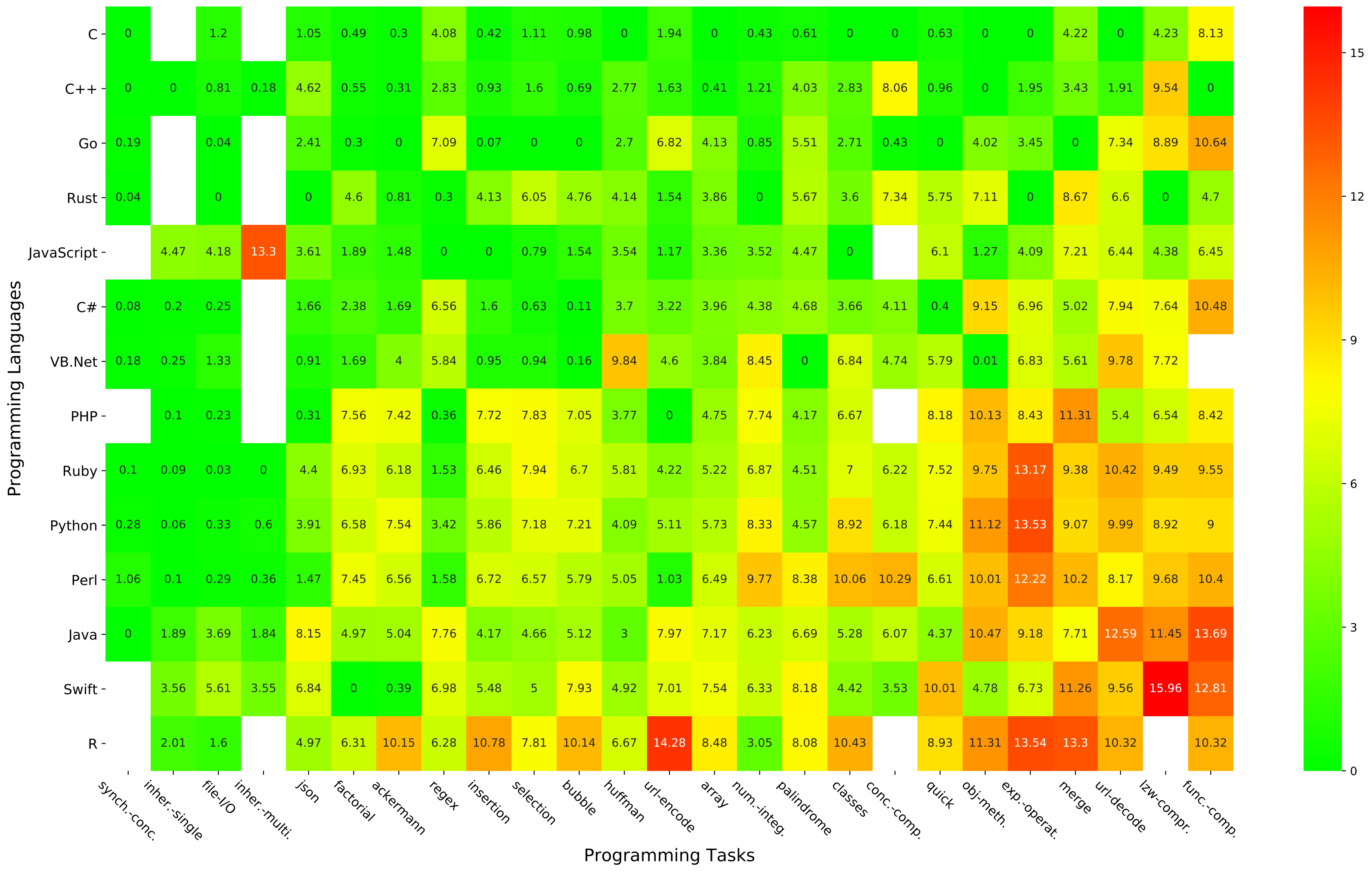
	synch.-conc.	inher.-single	inher.-multi.	json	file-I/O	factorial	regex	ackermann	url-decode	selection	num.-integ.	lzw-compr.	bubble	huffman	insertion	url-encode	array	palindrome	merge	quick	func.-comp.	conc.-comp.	obj-meth.	exp.-operat.	classes
C	0.04			0.56	1.09	0.46	1.77	0.21	0.18	0.64	0	0	0.78	0	0.67	2.56	0	0.1	1.1	0.53	0.84	0	0.1	0	0.76
Go	0.08			0.69	0.19	0.44	3.33	0.61	0.82	0.66	0.76	1.87	0.8	1.21	1.12	3.18	1.37	1.89	0	1.05	1.18	0.27	0.7	1.82	0
C++	0	0	0	2.64	0.88	0.36	1.28	0.16	0	0.99	0.6	2.99	1.11	2.05	1.41	1.65	0.4	2.04	0.96	0.79	0	4.63	0	1.58	2.76
JavaScript		1.35	5.89	0.2	2.38	0.6	0	0.7	0.99	0	0.01	0	0.19	1.61	0	1.06	2.44	2.99	2.17	2.98	0.64		0	1.93	0.1
C#	0.54	0.3		0.57	2.38	0.16	2.73	0	1.45	0.06	1.59	1.66	0	1.69	1.02	1.69	1.91	2.08	1.21	0	4.82	3.36	2.46	3.74	2.23
Rust	0.47			1.74	0	1.24	0.13	0.78	1	2.39	0.04	1.64	2.23	2.57	2.48	1.01	1.86	4.25	3.38	2.92	1.55	2.78	0.1	3.04	0
VB.Net	1.89	0.39		0.31	4.45	0	2.37	2.21	2.07	0.48	4.2	1.69	0.29	4.55	0.99	2.19	1.78	0	1.62	2.66		2.96	0.72	3.41	6.69
PHP		0.24		0	0.47	2.37	0.21	2.29	0.43	2.68	2.64	0.73	3.18	1.41	3.1	0	2.7	1.09	3.83	3.35	2.97	4.81	5.06	4.29	4.92
Swift		0.83	0.39	2.53	1.7	0.15	2.65	0.25	2.53	2.84	3.52	4.1	4.6	2.57	3.42	2.67	3.42	4.21	4.74	5.48	3.25	0.58	3.01	3.63	4.03
Ruby	0.06	0.37	0.32	2.27	0.05	2.94	0.44	2.63	2.95	3.38	3.63	2.56	3.34	3.15	3.37	2.37	3.22	1.74	3.59	4.09	3.7	4.24	4.81	5.66	5.13
Python	0.92	0.52	0.71	1.87	0.36	3.52	1.42	3.37	2.76	3.37	2.85	2.11	3.63	1.98	3.73	2.74	3.33	2.05	4.04	3.86	3.4	2.97	5.75	5.68	6.06
Perl	1.24	0.32	0.32	0.7	0.29	4.01	1	3.34	2.25	3.35	3.91	2.66	3.43	2.76	4.11	1.02	4.09	4.21	4.39	3.84	5.62	5.71	7.16	5.77	6.11
Java	0.45	1.8	1.73	4.99	4.24	3.03	4.4	3.52	4.87	2.06	2.27	4.02	2.55	2.04	2.31	5.07	4.42	3.3	3.03	2.31	4.44	3.4	5.38	4.48	5.46
R		1.45		2.49	3.15	4.42	3.19	4.88	3.2	3.61	1.63		3.68	4.15	4.22	5.64	3.97	5.07	3.62	4.57	4.11		6.74	5.82	6.87

Figure A.1: Embedded System’s EDP with Weighted Function 1 (Energy Efficiency)



	synch.-conc.	inher.-single	inher.-multi.	file-I/O	json	factorial	ackermann	regex	url-decode	selection	num.-integ.	lzw-compr.	bubble	huffman	insertion	url-encode	array	palindrome	merge	quick	func.-comp.	obj-meth.	conc.-comp.	exp.-operat.	classes
C	0.04			1.61	0.83	0.69	0.32	2.66	0.26	0.94	0	0	1.18	0	1.04	3.82	0	0.14	1.65	0.83	1.26	0.01	0	0	1.24
Go	0.08			0.19	1.11	0.67	0.91	4.98	1.24	0.96	1.19	2.95	1.21	1.77	1.72	4.75	2.07	2.81	0	1.75	1.76	0.91	0.46	2.71	0
C++	0	0	0	1.25	3.97	0.55	0.26	1.94	0	1.44	0.91	4.57	1.68	3.04	2.14	2.43	0.61	3.04	1.44	1.27	0	0	7.13	2.39	4.17
JavaScript		1.99	9.63	3.49	0.3	0.89	1.03	0	1.46	0	0	0	0.29	2.37	0	1.53	3.65	4.45	3.25	4.56	0.9	0.21		2.88	0.4
C#	0.84	0.45		3.5	0.88	0.24	0	4.09	2.18	0.06	2.62	2.62	0	2.48	1.58	2.53	2.88	3.08	1.81	0	8.21	3.52	5.15	5.62	3.38
Rust	0.77			0	2.61	1.87	1.16	0.23	1.5	3.54	0.09	2.54	3.36	3.8	3.74	1.49	2.8	6.38	5.07	4.5	2.36	0	4.34	4.55	0
VB.Net	2.93	0.54		6.58	0.48	0	3.3	3.56	3.11	0.71	6.54	2.64	0.44	6.77	1.51	3.34	2.68	0	2.44	4.08		0.93	4.5	5.28	10.16
PHP		0.32		0.69	0	3.53	3.42	0.3	0.63	3.97	3.97	1.21	4.75	2.06	4.68	0	4.2	1.61	5.73	5.12	4.44	7.39	7.3	6.43	7.4
Ruby	0.06	0.57	0.47	0.05	3.41	4.39	3.92	0.69	4.4	5.02	5.46	3.94	4.99	4.67	5.07	3.52	4.85	2.57	5.37	6.07	5.53	7.03	6.57	8.83	7.73
Swift		1.17	0.73	2.67	3.81	0.22	0.36	4	3.79	4.24	5.46	6.22	6.9	3.81	5.17	3.97	5.15	6.36	7.14	8.34	4.89	4.33	0.85	5.46	6.09
Python	1.39	0.73	1.05	0.49	2.8	5.24	5.02	2.12	4.12	4.99	4.7	3.22	5.42	2.95	5.6	4.07	4.99	3.03	6.03	5.87	5.07	8.43	4.55	8.91	9.11
Perl	1.94	0.47	0.53	0.41	1.05	5.99	4.98	1.51	3.38	4.97	6.01	4.06	5.12	4.08	6.17	1.5	6.13	6.68	6.56	5.85	8.95	10.55	8.65	8.8	9.61
Java	0.75	2.66	2.6	6.33	7.49	4.57	5.3	6.62	7.32	3.06	3.65	6.13	3.81	3.04	3.48	7.59	6.98	5.42	4.53	3.57	6.86	8.07	5.16	6.71	8.28
R		2.13		4.63	3.73	6.59	7.28	4.8	4.82	5.35	2.43		5.49	6.16	6.32	8.44	6.18	8.08	6.25	6.93	6.17	10.45		9.03	10.5

Figure A.2: Embedded System’s EDP with Weighted Function 2 (Balanced)



	synch.-conc.	inher.-single	file-I/O	inher.-multi.	json	factorial	ackermann	regex	insertion	selection	bubble	huffman	url-encode	array	num.-integ.	palindrome	classes	conc.-comp.	quick	obj-meth.	exp.-operat.	merge	url-decode	lzw-compr.	func.-comp.
C	0		1.2		1.05	0.49	0.3	4.08	0.42	1.11	0.98	0	1.94	0	0.43	0.61	0	0	0.63	0	0	4.22	0	4.23	8.13
C++	0	0	0.81	0.18	4.62	0.55	0.31	2.83	0.93	1.6	0.69	2.77	1.63	0.41	1.21	4.03	2.83	8.06	0.96	0	1.95	3.43	1.91	9.54	0
Go	0.19		0.04		2.41	0.3	0	7.09	0.07	0	0	2.7	6.82	4.13	0.85	5.51	2.71	0.43	0	4.02	3.45	0	7.34	8.89	10.64
Rust	0.04		0		0	4.6	0.81	0.3	4.13	6.05	4.76	4.14	1.54	3.86	0	5.67	3.6	7.34	5.75	7.11	0	8.67	6.6	0	4.7
JavaScript		4.47	4.18	13.3	3.61	1.89	1.48	0	0	0.79	1.54	3.54	1.17	3.36	3.52	4.47	0		6.1	1.27	4.09	7.21	6.44	4.38	6.45
C#	0.08	0.2	0.25		1.66	2.38	1.69	6.56	1.6	0.63	0.11	3.7	3.22	3.96	4.38	4.68	3.66	4.11	0.4	9.15	6.96	5.02	7.94	7.64	10.48
VB.Net	0.18	0.25	1.33		0.91	1.69	4	5.84	0.95	0.94	0.16	9.84	4.6	3.84	8.45	0	6.84	4.74	5.79	0.01	6.83	5.61	9.78	7.72	
PHP		0.1	0.23		0.31	7.56	7.42	0.36	7.72	7.83	7.05	3.77	0	4.75	7.74	4.17	6.67		8.18	10.13	8.43	11.31	5.4	6.54	8.42
Ruby	0.1	0.09	0.03	0	4.4	6.93	6.18	1.53	6.46	7.94	6.7	5.81	4.22	5.22	6.87	4.51	7	6.22	7.52	9.75	13.17	9.38	10.42	9.49	9.55
Python	0.28	0.06	0.33	0.6	3.91	6.58	7.54	3.42	5.86	7.18	7.21	4.09	5.11	5.73	8.33	4.57	8.92	6.18	7.44	11.12	13.53	9.07	9.99	8.92	9
Perl	1.06	0.1	0.29	0.36	1.47	7.45	6.56	1.58	6.72	6.57	5.79	5.05	1.03	6.49	9.77	8.38	10.06	10.29	6.61	10.01	12.22	10.2	8.17	9.68	10.4
Java	0	1.89	3.69	1.84	8.15	4.97	5.04	7.76	4.17	4.66	5.12	3	7.97	7.17	6.23	6.69	5.28	6.07	4.37	10.47	9.18	7.71	12.59	11.45	13.69
Swift		3.56	5.61	3.55	6.84	0	0.39	6.98	5.48	5	7.93	4.92	7.01	7.54	6.33	8.18	4.42	3.53	10.01	4.78	6.73	11.26	9.56	15.96	12.81
R		2.01	1.6		4.97	6.31	10.15	6.28	10.78	7.81	10.14	6.67	14.28	8.48	3.05	8.08	10.43		8.93	11.31	13.54	13.3	10.32		10.32

Figure A.3: Embedded System's EDP with Weighted Function 3 (Performance Efficiency)

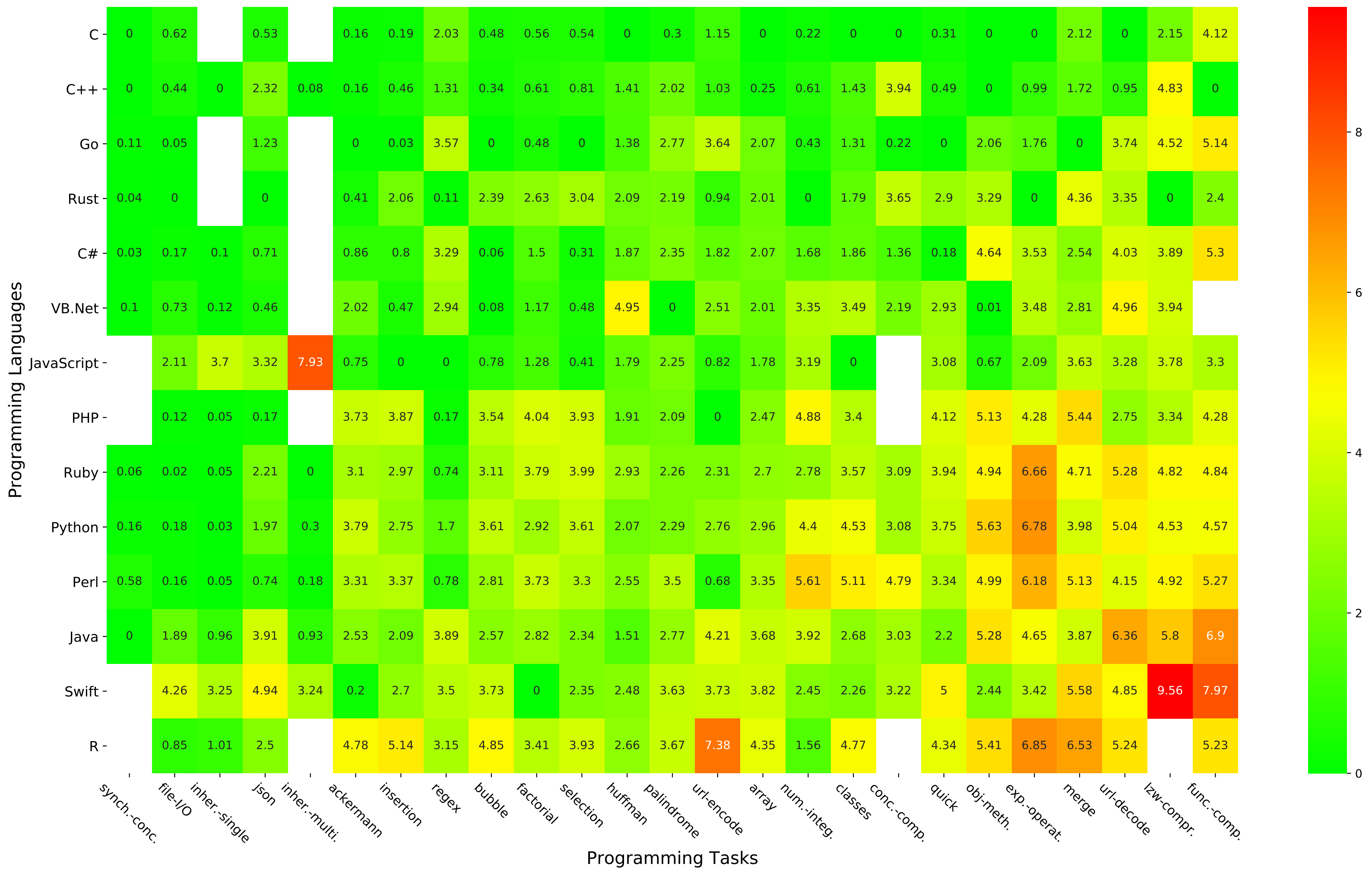
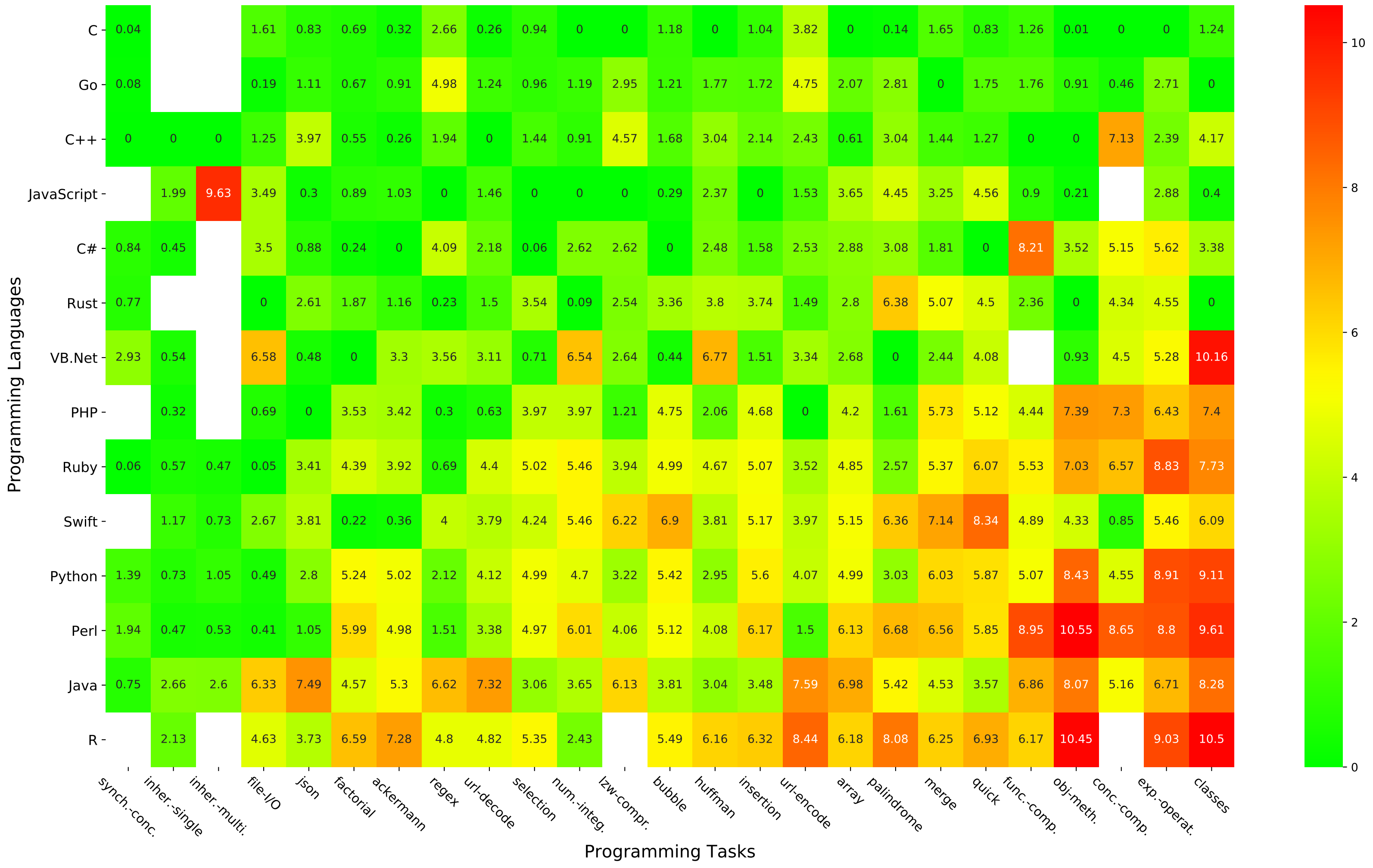
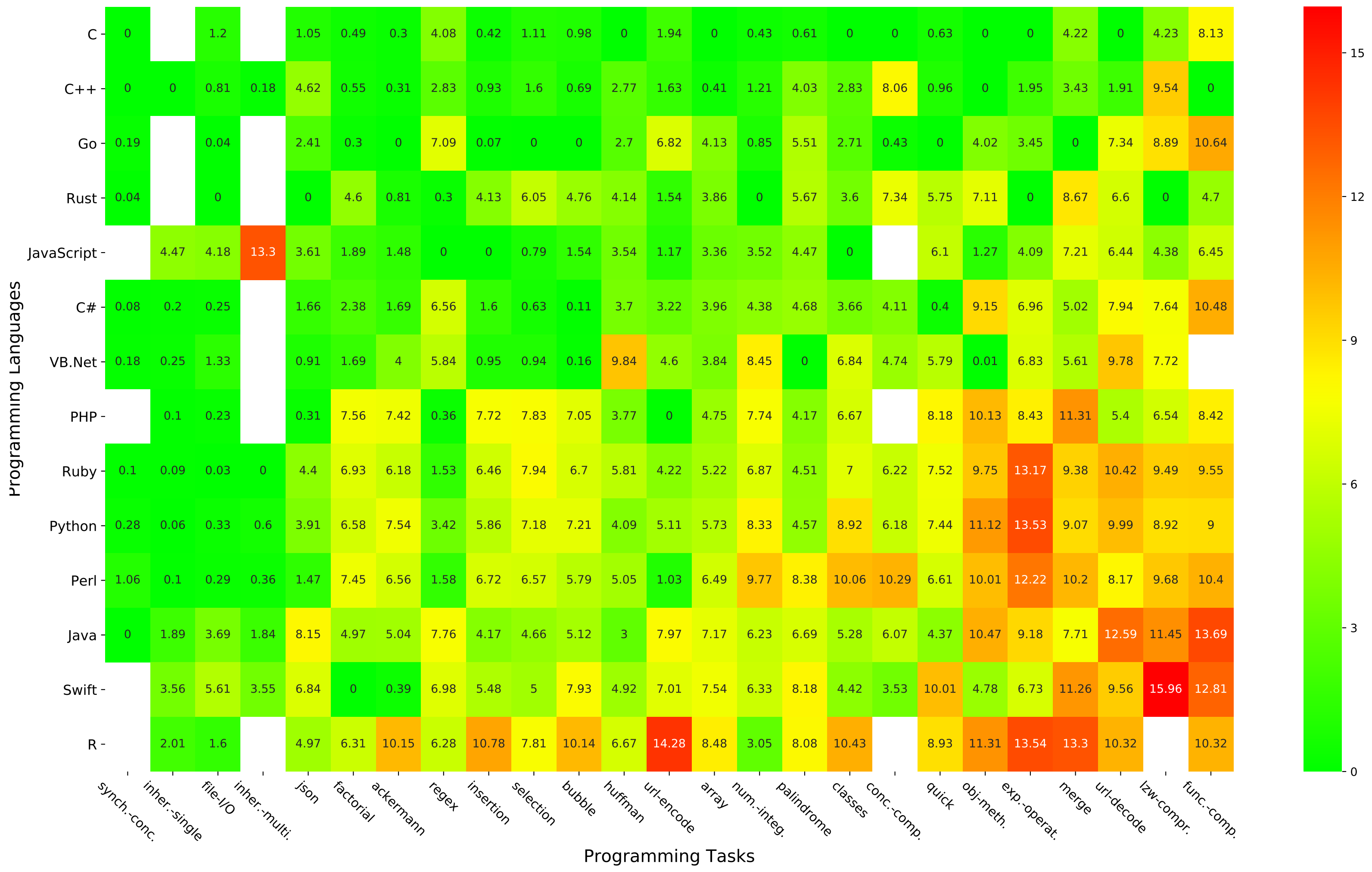


Figure A.4: Laptop's EDP with Weighted Function 1 (Energy Efficiency)



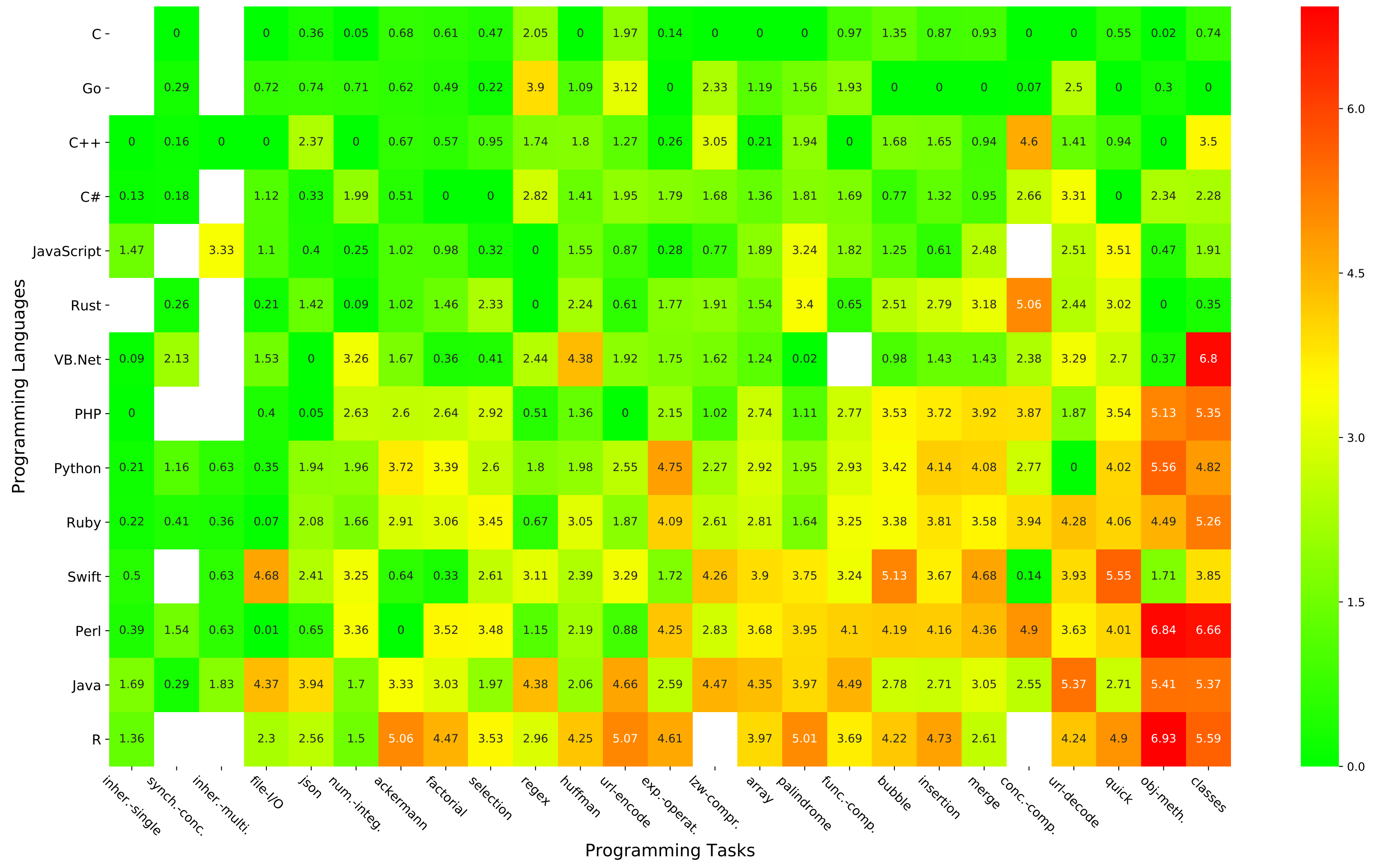
	synch.-conc.	inher.-single	inher.-multi.	file-I/O	json	factorial	ackermann	regex	url-decode	selection	num.-integ.	lzw-compr.	bubble	huffman	insertion	url-encode	array	palindrome	merge	quick	func.-comp.	obj.-meth.	conc.-comp.	exp.-operat.	classes
C	0.04			1.61	0.83	0.69	0.32	2.66	0.26	0.94	0	0	1.18	0	1.04	3.82	0	0.14	1.65	0.83	1.26	0.01	0	0	1.24
Go	0.08			0.19	1.11	0.67	0.91	4.98	1.24	0.96	1.19	2.95	1.21	1.77	1.72	4.75	2.07	2.81	0	1.75	1.76	0.91	0.46	2.71	0
C++	0	0	0	1.25	3.97	0.55	0.26	1.94	0	1.44	0.91	4.57	1.68	3.04	2.14	2.43	0.61	3.04	1.44	1.27	0	0	7.13	2.39	4.17
JavaScript		1.99	9.63	3.49	0.3	0.89	1.03	0	1.46	0	0	0	0.29	2.37	0	1.53	3.65	4.45	3.25	4.56	0.9	0.21		2.88	0.4
C#	0.84	0.45		3.5	0.88	0.24	0	4.09	2.18	0.06	2.62	2.62	0	2.48	1.58	2.53	2.88	3.08	1.81	0	8.21	3.52	5.15	5.62	3.38
Rust	0.77			0	2.61	1.87	1.16	0.23	1.5	3.54	0.09	2.54	3.36	3.8	3.74	1.49	2.8	6.38	5.07	4.5	2.36	0	4.34	4.55	0
VB.Net	2.93	0.54		6.58	0.48	0	3.3	3.56	3.11	0.71	6.54	2.64	0.44	6.77	1.51	3.34	2.68	0	2.44	4.08		0.93	4.5	5.28	10.16
PHP		0.32		0.69	0	3.53	3.42	0.3	0.63	3.97	3.97	1.21	4.75	2.06	4.68	0	4.2	1.61	5.73	5.12	4.44	7.39	7.3	6.43	7.4
Ruby	0.06	0.57	0.47	0.05	3.41	4.39	3.92	0.69	4.4	5.02	5.46	3.94	4.99	4.67	5.07	3.52	4.85	2.57	5.37	6.07	5.53	7.03	6.57	8.83	7.73
Swift		1.17	0.73	2.67	3.81	0.22	0.36	4	3.79	4.24	5.46	6.22	6.9	3.81	5.17	3.97	5.15	6.36	7.14	8.34	4.89	4.33	0.85	5.46	6.09
Python	1.39	0.73	1.05	0.49	2.8	5.24	5.02	2.12	4.12	4.99	4.7	3.22	5.42	2.95	5.6	4.07	4.99	3.03	6.03	5.87	5.07	8.43	4.55	8.91	9.11
Perl	1.94	0.47	0.53	0.41	1.05	5.99	4.98	1.51	3.38	4.97	6.01	4.06	5.12	4.08	6.17	1.5	6.13	6.68	6.56	5.85	8.95	10.55	8.65	8.8	9.61
Java	0.75	2.66	2.6	6.33	7.49	4.57	5.3	6.62	7.32	3.06	3.65	6.13	3.81	3.04	3.48	7.59	6.98	5.42	4.53	3.57	6.86	8.07	5.16	6.71	8.28
R		2.13		4.63	3.73	6.59	7.28	4.8	4.82	5.35	2.43		5.49	6.16	6.32	8.44	6.18	8.08	6.25	6.93	6.17	10.45		9.03	10.5

Figure A.5: Laptop’s EDP with Weighted Function 2 (Balanced)



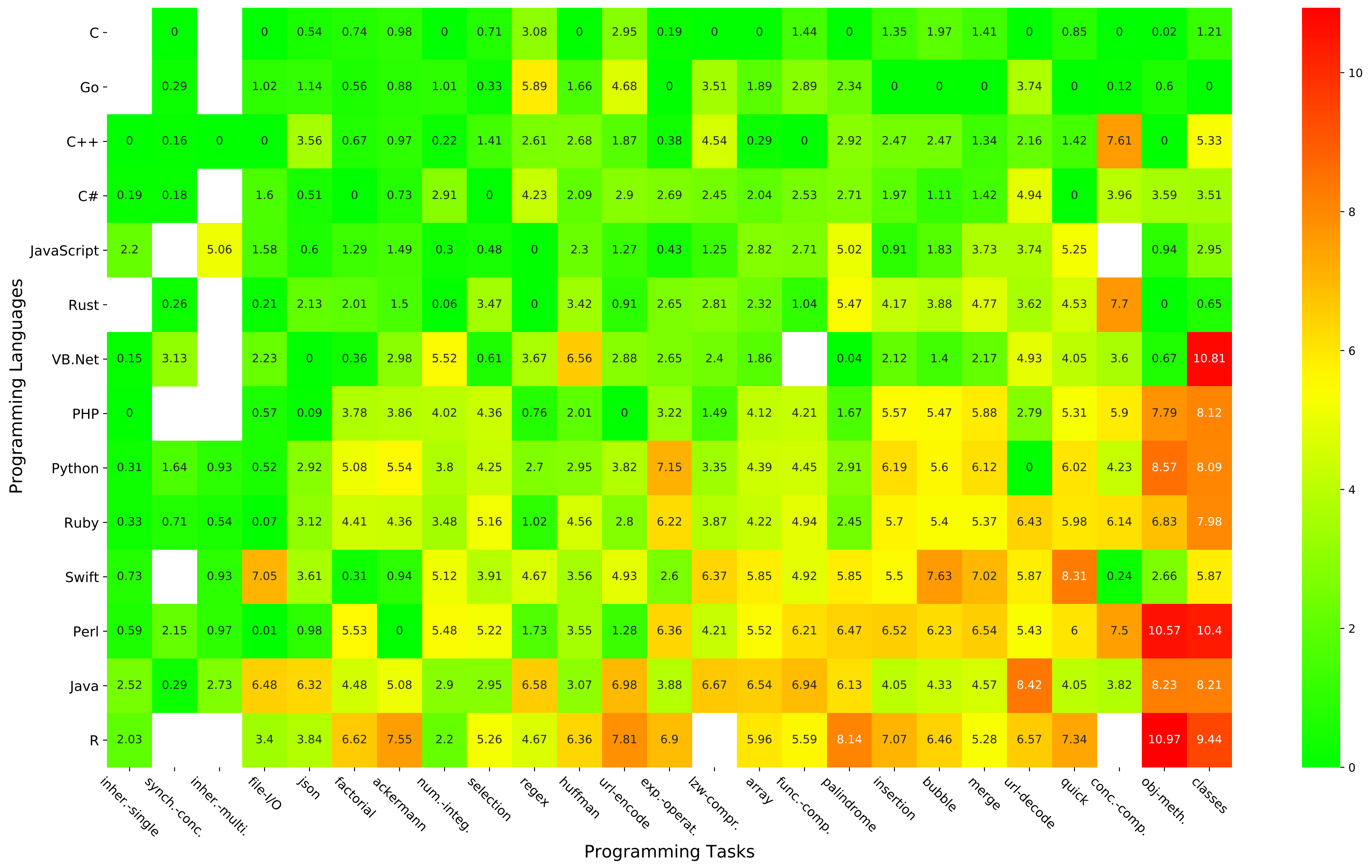
	synch.-conc.	inher.-single	file-I/O	inher.-multi.	json	factorial	ackermann	regex	insertion	selection	bubble	huffman	url-encode	array	num.-integ.	palindrome	classes	conc.-comp.	quick	obj-meth.	exp.-operat.	merge	url-decode	lzw-compr.	func.-comp.
C	0		1.2		1.05	0.49	0.3	4.08	0.42	1.11	0.98	0	1.94	0	0.43	0.61	0	0	0.63	0	0	4.22	0	4.23	8.13
C++	0	0	0.81	0.18	4.62	0.55	0.31	2.83	0.93	1.6	0.69	2.77	1.63	0.41	1.21	4.03	2.83	8.06	0.96	0	1.95	3.43	1.91	9.54	0
Go	0.19		0.04		2.41	0.3	0	7.09	0.07	0	0	2.7	6.82	4.13	0.85	5.51	2.71	0.43	0	4.02	3.45	0	7.34	8.89	10.64
Rust	0.04		0		0	4.6	0.81	0.3	4.13	6.05	4.76	4.14	1.54	3.86	0	5.67	3.6	7.34	5.75	7.11	0	8.67	6.6	0	4.7
JavaScript		4.47	4.18	13.3	3.61	1.89	1.48	0	0	0.79	1.54	3.54	1.17	3.36	3.52	4.47	0		6.1	1.27	4.09	7.21	6.44	4.38	6.45
C#	0.08	0.2	0.25		1.66	2.38	1.69	6.56	1.6	0.63	0.11	3.7	3.22	3.96	4.38	4.68	3.66	4.11	0.4	9.15	6.96	5.02	7.94	7.64	10.48
VB.Net	0.18	0.25	1.33		0.91	1.69	4	5.84	0.95	0.94	0.16	9.84	4.6	3.84	8.45	0	6.84	4.74	5.79	0.01	6.83	5.61	9.78	7.72	
PHP		0.1	0.23		0.31	7.56	7.42	0.36	7.72	7.83	7.05	3.77	0	4.75	7.74	4.17	6.67		8.18	10.13	8.43	11.31	5.4	6.54	8.42
Ruby	0.1	0.09	0.03	0	4.4	6.93	6.18	1.53	6.46	7.94	6.7	5.81	4.22	5.22	6.87	4.51	7	6.22	7.52	9.75	13.17	9.38	10.42	9.49	9.55
Python	0.28	0.06	0.33	0.6	3.91	6.58	7.54	3.42	5.86	7.18	7.21	4.09	5.11	5.73	8.33	4.57	8.92	6.18	7.44	11.12	13.53	9.07	9.99	8.92	9
Perl	1.06	0.1	0.29	0.36	1.47	7.45	6.56	1.58	6.72	6.57	5.79	5.05	1.03	6.49	9.77	8.38	10.06	10.29	6.61	10.01	12.22	10.2	8.17	9.68	10.4
Java	0	1.89	3.69	1.84	8.15	4.97	5.04	7.76	4.17	4.66	5.12	3	7.97	7.17	6.23	6.69	5.28	6.07	4.37	10.47	9.18	7.71	12.59	11.45	13.69
Swift		3.56	5.61	3.55	6.84	0	0.39	6.98	5.48	5	7.93	4.92	7.01	7.54	6.33	8.18	4.42	3.53	10.01	4.78	6.73	11.26	9.56	15.96	12.81
R		2.01	1.6		4.97	6.31	10.15	6.28	10.78	7.81	10.14	6.67	14.28	8.48	3.05	8.08	10.43		8.93	11.31	13.54	13.3	10.32		10.32

Figure A.6: Laptop’s EDP with Weighted Function 3 (Performance Efficiency)



	inher.-single	synch.-conc.	inher.-multi.	file-I/O	json	num.-integ.	ackermann	factorial	selection	regex	huffman	url-encode	exp.-operat.	lzw-compr.	array	palindrome	func.-comp.	bubble	insertion	merge	conc.-comp.	url-decode	quick	obj-meth.	classes
C		0		0	0.36	0.05	0.68	0.61	0.47	2.05	0	1.97	0.14	0	0	0	0.97	1.35	0.87	0.93	0	0	0.55	0.02	0.74
Go		0.29		0.72	0.74	0.71	0.62	0.49	0.22	3.9	1.09	3.12	0	2.33	1.19	1.56	1.93	0	0	0	0.07	2.5	0	0.3	0
C++	0	0.16	0	0	2.37	0	0.67	0.57	0.95	1.74	1.8	1.27	0.26	3.05	0.21	1.94	0	1.68	1.65	0.94	4.6	1.41	0.94	0	3.5
C#	0.13	0.18		1.12	0.33	1.99	0.51	0	0	2.82	1.41	1.95	1.79	1.68	1.36	1.81	1.69	0.77	1.32	0.95	2.66	3.31	0	2.34	2.28
JavaScript	1.47		3.33	1.1	0.4	0.25	1.02	0.98	0.32	0	1.55	0.87	0.28	0.77	1.89	3.24	1.82	1.25	0.61	2.48		2.51	3.51	0.47	1.91
Rust		0.26		0.21	1.42	0.09	1.02	1.46	2.33	0	2.24	0.61	1.77	1.91	1.54	3.4	0.65	2.51	2.79	3.18	5.06	2.44	3.02	0	0.35
VB.Net	0.09	2.13		1.53	0	3.26	1.67	0.36	0.41	2.44	4.38	1.92	1.75	1.62	1.24	0.02		0.98	1.43	1.43	2.38	3.29	2.7	0.37	6.8
PHP	0			0.4	0.05	2.63	2.6	2.64	2.92	0.51	1.36	0	2.15	1.02	2.74	1.11	2.77	3.53	3.72	3.92	3.87	1.87	3.54	5.13	5.35
Python	0.21	1.16	0.63	0.35	1.94	1.96	3.72	3.39	2.6	1.8	1.98	2.55	4.75	2.27	2.92	1.95	2.93	3.42	4.14	4.08	2.77	0	4.02	5.56	4.82
Ruby	0.22	0.41	0.36	0.07	2.08	1.66	2.91	3.06	3.45	0.67	3.05	1.87	4.09	2.61	2.81	1.64	3.25	3.38	3.81	3.58	3.94	4.28	4.06	4.49	5.26
Swift	0.5		0.63	4.68	2.41	3.25	0.64	0.33	2.61	3.11	2.39	3.29	1.72	4.26	3.9	3.75	3.24	5.13	3.67	4.68	0.14	3.93	5.55	1.71	3.85
Perl	0.39	1.54	0.63	0.01	0.65	3.36	0	3.52	3.48	1.15	2.19	0.88	4.25	2.83	3.68	3.95	4.1	4.19	4.16	4.36	4.9	3.63	4.01	6.84	6.66
Java	1.69	0.29	1.83	4.37	3.94	1.7	3.33	3.03	1.97	4.38	2.06	4.66	2.59	4.47	4.35	3.97	4.49	2.78	2.71	3.05	2.55	5.37	2.71	5.41	5.37
R	1.36			2.3	2.56	1.5	5.06	4.47	3.53	2.96	4.25	5.07	4.61		3.97	5.01	3.69	4.22	4.73	2.61		4.24	4.9	6.93	5.59

Figure A.7: Server's EDP with Weighted Function 1 (Energy Efficiency)



	inher.-single	synch.-conc.	inher.-multi.	file-I/O	json	factorial	ackermann	num.-integ.	selection	regex	huffman	url-encode	exp.-operat.	lzw-compr.	array	func.-comp.	palindrome	insertion	bubble	merge	url-decode	quick	conc.-comp.	obj-meth.	classes
C		0		0	0.54	0.74	0.98	0	0.71	3.08	0	2.95	0.19	0	0	1.44	0	1.35	1.97	1.41	0	0.85	0	0.02	1.21
Go		0.29		1.02	1.14	0.56	0.88	1.01	0.33	5.89	1.66	4.68	0	3.51	1.89	2.89	2.34	0	0	0	3.74	0	0.12	0.6	0
C++	0	0.16	0	0	3.56	0.67	0.97	0.22	1.41	2.61	2.68	1.87	0.38	4.54	0.29	0	2.92	2.47	2.47	1.34	2.16	1.42	7.61	0	5.33
C#	0.19	0.18		1.6	0.51	0	0.73	2.91	0	4.23	2.09	2.9	2.69	2.45	2.04	2.53	2.71	1.97	1.11	1.42	4.94	0	3.96	3.59	3.51
JavaScript	2.2		5.06	1.58	0.6	1.29	1.49	0.3	0.48	0	2.3	1.27	0.43	1.25	2.82	2.71	5.02	0.91	1.83	3.73	3.74	5.25		0.94	2.95
Rust		0.26		0.21	2.13	2.01	1.5	0.06	3.47	0	3.42	0.91	2.65	2.81	2.32	1.04	5.47	4.17	3.88	4.77	3.62	4.53	7.7	0	0.65
VB.Net	0.15	3.13		2.23	0	0.36	2.98	5.52	0.61	3.67	6.56	2.88	2.65	2.4	1.86		0.04	2.12	1.4	2.17	4.93	4.05	3.6	0.67	10.81
PHP	0			0.57	0.09	3.78	3.86	4.02	4.36	0.76	2.01	0	3.22	1.49	4.12	4.21	1.67	5.57	5.47	5.88	2.79	5.31	5.9	7.79	8.12
Python	0.31	1.64	0.93	0.52	2.92	5.08	5.54	3.8	4.25	2.7	2.95	3.82	7.15	3.35	4.39	4.45	2.91	6.19	5.6	6.12	0	6.02	4.23	8.57	8.09
Ruby	0.33	0.71	0.54	0.07	3.12	4.41	4.36	3.48	5.16	1.02	4.56	2.8	6.22	3.87	4.22	4.94	2.45	5.7	5.4	5.37	6.43	5.98	6.14	6.83	7.98
Swift	0.73		0.93	7.05	3.61	0.31	0.94	5.12	3.91	4.67	3.56	4.93	2.6	6.37	5.85	4.92	5.85	5.5	7.63	7.02	5.87	8.31	0.24	2.66	5.87
Perl	0.59	2.15	0.97	0.01	0.98	5.53	0	5.48	5.22	1.73	3.55	1.28	6.36	4.21	5.52	6.21	6.47	6.52	6.23	6.54	5.43	6	7.5	10.57	10.4
Java	2.52	0.29	2.73	6.48	6.32	4.48	5.08	2.9	2.95	6.58	3.07	6.98	3.88	6.67	6.54	6.94	6.13	4.05	4.33	4.57	8.42	4.05	3.82	8.23	8.21
R	2.03			3.4	3.84	6.62	7.55	2.2	5.26	4.67	6.36	7.81	6.9		5.96	5.59	8.14	7.07	6.46	5.28	6.57	7.34		10.97	9.44

Figure A.8: Server's EDP with Weighted Function 2 (Balanced)

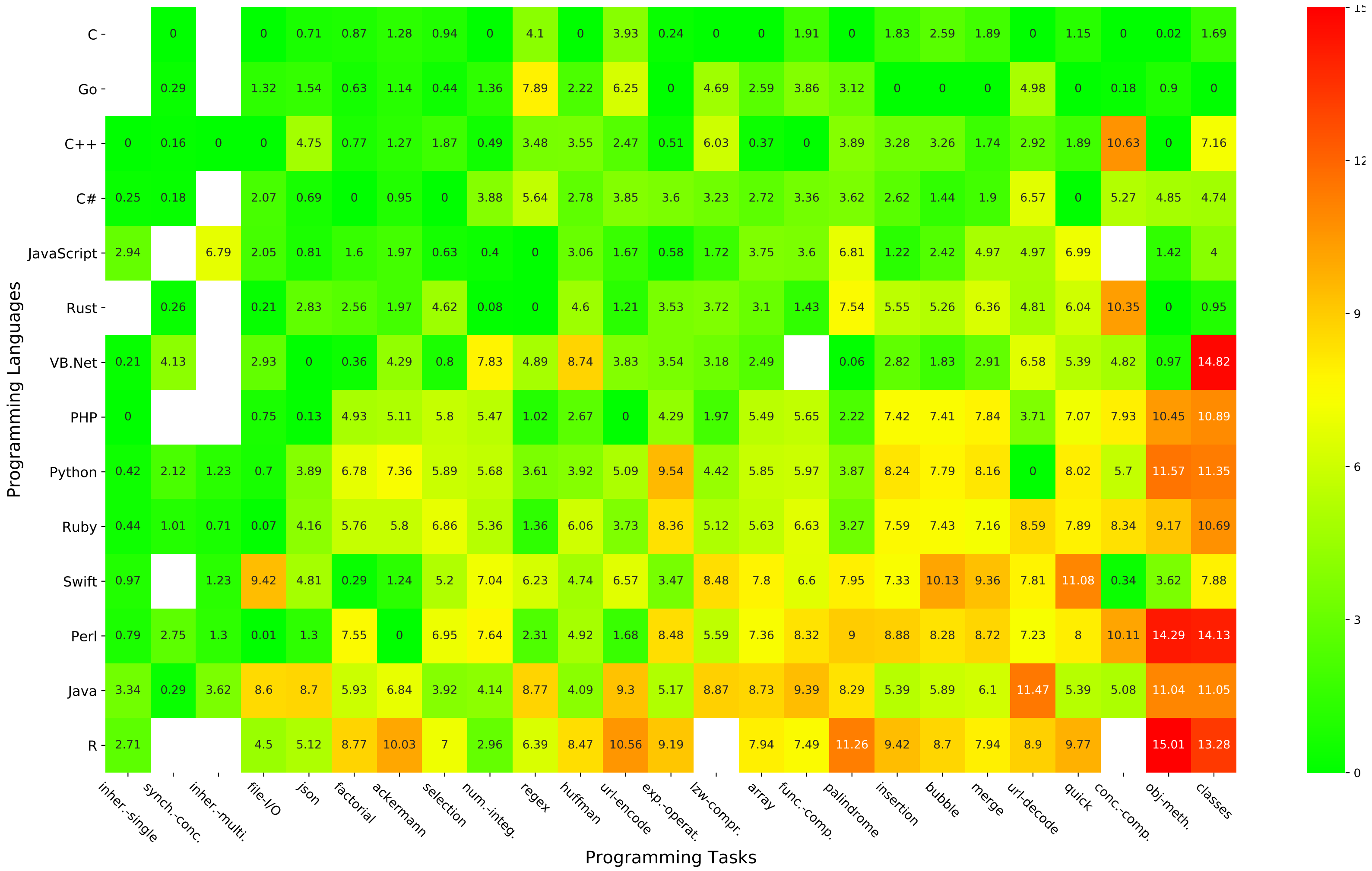


Figure A.9: Server’s EDP with Weighted Function 3 (Performance Efficiency)

Bibliography

[1] How much electricity does an American home use? - FAQ - U.S. Energy Information Administration (EIA). URL https://www.eia.gov/tools/faqs/faq.php?id=97.

[2] TIOBE Index | TIOBE - The Software Quality Company, 2017. [Online]. Available: = https://www.tiobe.com/tiobe-index/. [Accessed: 2017-09-12].

[3] How fast is Redis? – Redis, 2018. URL https://redis.io/topics/benchmarks.

[4] Usage Statistics and Market Share of Linux for Websites, December 2019, 2019. URL https://w3techs.com/technologies/details/os-linux.

[5] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong. Program energy efficiency: The impact of language, compiler and implementation choices. In *Green Computing Conference (IGCC), 2014 International*, pages 1–6, November 2014. doi: 10.1109/IGCC.2014.7039169.

[6] S. Abdulsalam, Z. Zong, Q. Gu, and Meikang Qiu. Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency. In *6th International Green and Sustainable Computing Conference*, IGSC '15, pages 1–8, Dec 2015. doi: 10.1109/IGCC.2015.7393699.

[7] Acmeism. RosettaCodeData: RosettaCode Data Project, nov 2017. URL https://github.com/acmeism/RosettaCodeData.

[8] K. Aggarwal, A. Hindle, and E. Stroulia. GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 311–320, September 2015. doi: 10.1109/ICSM.2015.7332477.

[9] Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. 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, pages 219–233, Riverton, NJ, USA, 2014. IBM Corp.

[10] G. Agosta, M. Bessi, E. Capra, and C. Francalanci. Dynamic memoization for energy efficiency in financial applications. In *2011 International Green Computing Conference and Workshops*, pages 1–8, July 2011. doi: 10.1109/IGCC.2011.6008559.

[11] Anders S. G. Andrae and Tomas Edler. On Global Electricity Usage of Communication Technology: Trends to 2030. *Challenges*, 6(1):117–157, June 2015. doi: 10.3390/challe6010117. URL `https://www.mdpi.com/2078-1547/6/1/117`.

[12] Anys Bacha and Radu Teodorescu. Dynamic Reduction of Voltage Margins by Leveraging On-chip ECC in Itanium II Processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 297–307, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485948. URL `http://doi.acm.org/10.1145/2485922.2485948`.

[13] Anys Bacha and Radu Teodorescu. Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 306–318, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6998-2. doi: 10.1109/MICRO.2014.54. URL `http://dx.doi.org/10.1109/MICRO.2014.54`.

[14] Woongki Baek and Trishul M. Chilimbi. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806620.

[15] Peter Bailey. Watts Up Pro power meter interface utility for Linux, September 2017. URL `https://github.com/pyrovski/watts-up`.

[16] Jayant Baliga, Robert Ayre, Kerry Hinton, and Rodney S. Tucker. Energy consumption in wired and wireless access networks. *IEEE Communications Magazine*, 49(6):70–77, June 2011. ISSN 1558-1896. doi: 10.1109/MCOM.2011.5783987.

[17] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 588–598, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635871.

[18] Mohamed Amine Beghoura, Abdelhak Boubetra, and Abdallah Boukerram. Green software requirements and measurement: random decision forests-based software energy consumption profiling. *Requirements Engineering*, pages 1–14, July 2015. ISSN 0947-3602, 1432-010X. doi: 10.1007/s00766-015-0234-2.

[19] J. Martin Bland and Douglas G. Altman. Multiple significance tests: the Bonferroni method. *BMJ*, 310(6973):170, January 1995. ISSN 0959-8138, 1468-5833. doi: 10.1136/bmj.310.6973.170. URL `https://www.bmj.com/content/310/6973/170`.

[20] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, September 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL `http://doi.acm.org/10.1145/324133.324234`.

[21] A. E. Husain Bohra and V. Chaudhary. VMeter: Power modelling for virtualized clouds. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010. doi: 10.1109/IPDPSW.2010.5470907.

[22] J. Bornholt, T. Mytkowicz, and K. S. McKinley. The model is not enough: Understanding energy consumption in mobile devices. In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–3, August 2012.

[23] A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier. PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level, 2012. [Online]. Available: https://ercim-news.ercim.eu/en92/special/powerapi-a-software-library-to-monitor-the-energy-consumed-at-the-process-level. [Accessed: 2016-06-13].

[24] Gerome Bovet and Jean Hennebert. Communicating With Things - An Energy Consumption Analysis. In *2012 IEEE Tenth International Conference on Pervasive Computing (Pervasive '2012)*, June 2012.

[25] D. Branco and P. R. Henriques. Impact of GCC optimization levels in energy consumption during C/C++ program execution. In *2015 IEEE 13th International Scientific Conference on Informatics*, pages 52–56, November 2015. doi: 10.1109/Informatics.2015.7377807.

[26] Christian Bunse, Zur Schwedenschanze, and Sebastian Stiemer. On the energy consumption of design patterns. In *Proceedings of the 2nd Workshop EASED@ BUIS Energy Aware Software-Engineering and Development*, pages 7–8. Citeseer, 2013.

[27] Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González. Thread shuffling: Combining DVFS and thread migration to reduce energy consumptions for multi-core systems. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 379–384, August 2011. doi: 10.1109/ISLPED.2011.5993670.

[28] K. W. Cameron, Rong Ge, and Xizhou Feng. High-performance, power-aware distributed computing for scientific applications. *Computer*, 38(11):40–47, November 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.380.

[29] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 769–784, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3363219. URL https://doi.org/10.1145/3319535.3363219.

[30] Eugenio Capra, Chiara Francalanci, and Sandra A. Slaughter. Is Software "Green"? Application Development Environments and Energy Efficiency in Open Source Applications. *Inf. Softw. Technol.*, 54:60–71, January 2012. ISSN 0950-5849.

[31] Tim Caswell, nov 2017. URL https://github.com/creationix/libco.

[32] Tim Caswell. nvm: Node Version Manager - Simple bash script to manage multiple active node.js versions, jan 2018. URL https://github.com/creationix/nvm.

[33] C. L. Chamas, D. Cordeiro, and M. M. Eler. Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: An energy cost analysis. In *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*, pages 1–6, November 2017.

[34] X. Chen and Z. Zong. 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)*, pages 485–492, October 2016. doi: 10.1109/BDCloud-SocialCom-SustainCom.2016.77.

[35] Y. K. Chen, J. Chhugani, P. Dubey, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, and M. Smelyanskiy. Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications. *Proceedings of the IEEE*, 96(5):790–807, May 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917729.

[36] Shaiful Alam Chowdhury and Abram Hindle. GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 49–60, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2901763. URL http://doi.acm.org/10.1145/2901739.2901763.

[37] Shaiful Alam Chowdhury, Abram Hindle, Rick Kazman, Takumi Shuto, Ken Matsui, and Yasutaka Kamei. GreenBundle: An Empirical Study on the Energy Impact of Bundled Processing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1107–1118, May 2019. doi: 10.1109/ICSE.2019.00114. ISSN: 1558-1225.

[38] Maxime Colmant, Mascha Kurpicz, Pascal Felber, Loïc Huertas, Romain Rouvoy, and Anita Sobe. Process-level Power Estimation in VM-based Systems. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 14:1–14:14, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741971.

[39] Luis Corral, Anton B. Georgiev, Alberto Sillitti, and Giancarlo Succi. Can Execution Time Describe Accurately the Energy Consumption of Mobile Apps? An Experiment in Android. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 31–37, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2844-9. doi: 10.1145/2593743.2593748. URL http://doi.acm.org/10.1145/2593743.2593748. event-place: Hyderabad, India.

[40] Ivan Tomas Cotes-Ruiz, Rocio P. Prado, Sebastian Garcia-Galan, Jose Enrique Munoz-Exposito, and Nicolas Ruiz-Reyes. Dynamic Voltage Frequency Scaling Simulator for Real Workflows Energy-Aware Management in Green Cloud Computing. *PLOS ONE*, 12(1):e0169803, January 2017. ISSN 1932-6203. doi: 10.1371/journal.pone.0169803. URL https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0169803.

[41] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, SSYM'98, page 5, San Antonio, Texas, January 1998. USENIX Association.

[42] Thurston H. Y. Dang, Petros Maniatis, and David A. Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *ASIA CCS '15*, 2015. doi: 10.1145/2714576.2714635.

[43] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, Aug 2010. doi: 10.1145/1840845.1840883.

[44] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0146-6. doi: 10.1145/1840845.1840883. URL http://doi.acm.org/10.1145/1840845.1840883.

[45] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 455–470, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343053. doi: 10.1145/2989081.2989088. URL https://doi.org/10.1145/2989081.2989088.

[46] Developer.android. Android NDK | Android Developers, 2018. URL https://developer.android.com/ndk/index.html.

[47] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-Based Energy Profiling of Android Apps: Simple, Efficient and Reliable? *Software Analysis Evolution and Reengineering (SANER)*, February 2017. URL http://orbilu.uni.lu/handle/10993/29378.

[48] Die.net. strace(1): trace system calls/signals - Linux man page, jan 2018. URL https://linux.die.net/man/1/strace.

[49] Die.net. time(1) - Linux man page, jan 2018. URL https://linux.die.net/man/1/time.

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

[51] eLinux. RPI vcgencmd usage - eLinux.org, aug 2017. URL https://elinux.org/RPI_vcgencmd_usage.

[52] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460, December 2012. doi: 10.1109/MICRO.2012.48.

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

[54] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[55] Hot Frameworks. Web framework rankings | HotFrameworks, May 2018. URL https://hotframeworks.com/.

[56] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 978-0-201-63361-0.

[57] Stefanos Georgiou. Rosetta_code_research_msr: Exploiting Programming Languages Energy Consumption, mar 2018. URL https://github.com/stefanos1316/Rosetta_Code_Research_MSR/Scripts.

[58] Stefanos Georgiou and Diomidis Spinellis. Energy-Delay Investigation of Remote Inter-Process Communication Technologies. *Journal of Systems and Software*, page 110506, December 2019. ISSN 0164-1212. doi: 10.1016/j.jss.2019.110506. URL http://www.sciencedirect.com/science/article/pii/S0164121219302808.

[59] Jakob Gruber. Speeding up V8 Regular Expressions, jan 2018. URL https://v8project.blogspot.com/2017/01/speeding-up-v8-regular-expressions.html.

[60] Richard F. Gunst and Robert L. Mason. Fractional factorial design. *WIREs Computational Statistics*, 1(2):234–244, 2009. ISSN 1939-0068. doi: 10.1002/wics.27. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/wics.27.

[61] Michael Hablich. v8: The official mirror of the V8 Git repository, jan 2018. URL https://github.com/v8/v8/wiki/Design-Elements.

[62] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 92–101, May 2013. doi: 10.1109/ICSE.2013.6606555.

[63] IJ Haratcherev, GP Halkes, TEV Parker, OW Visser, and KG Langendoen. *PowerBench: A Scalable Testbed Infrastructure for Benchmarking Power Consumption*, pages 37–44. s.n., 2008. ISBN 978-90-9023209-6.

[64] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy Profiles of Java Collections Classes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 225–236, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1.

[65] Hesham H. M. Hassan, Ahmed Shawky Moussa, and Ibrahim Farag. Performance vs. Power and Energy Consumption: Impact of Coding Style and Compiler. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 8(12), 2017. doi: 10.14569/IJACSA.2017.081217. URL https://thesai.org/Publications/ViewPaper?Volume=8&Issue=12&Code=IJACSA&SerialNo=17.

[66] Alexander A. Hernandez and Sherwin E. Ona. A Qualitative Study of Green IT Adoption Within the Philippines Business Process Outsourcing Industry: A Multi-Theory Perspective. *Int. J. Enterp. Inf. Syst.*, 11(4):28–62, October 2015. ISSN 1548-1115. doi: 10.4018/IJEIS.2015100102. URL http://dx.doi.org/10.4018/IJEIS.2015100102.

[67] V. Herwig, R. Fischer, and P. Braun. Assessment of REST and WebSocket in regards to their energy consumption for mobile applications. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, volume 1, pages 342–347, September 2015.

[68] Abram Hindle. Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, 20(2):374–409, April 2015. ISSN 1573-7616. doi: 10.1007/s10664-013-9276-6.

[69] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 12–21, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0.

[70] Timo Honig, Heiko Janker, Christopher Eibel, Oliver Mihelic, and Rüdiger Kapitza. Proactive Energy-Aware Programming with PEEK. 2014.

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

[72] HP. HP EliteBook 840 G3 Notebook PC| HP® United States, Mar 2018. URL http://www8.hp.com/us/en/products/laptops/product-detail.html?oid=7815294.

[73] GitHub Info. GitHut - Programming Languages and GitHub, June 2018. URL http://githut.info/.

[74] Melanie Kambadur and Martha A. Kim. An Experimental Survey of Energy Management Across the Stack. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 329–344, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1.

[75] Karan Aggarwal. *The Power of System Call Traces: Predicting the Software Energy Impact of Changes*. November 2014. URL http://archive.org/details/Cascon2014.

[76] Richard Kavanagh and Karim Djemame. Rapid and accurate energy models through calibration with ipmi and rapl. *Concurrency and Computation: Practice and Experience*, 31(13):e5124, 2019. doi: 10.1002/cpe.5124. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5124. e5124 cpe.5124.

[77] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), March 2018. ISSN 2376-3639. doi: 10.1145/3177754. URL https://doi.org/10.1145/3177754.

[78] Won Kim and S. Korea. Computing: Today and tomorrow. *Journal of Object Technology*, pages 65–72, 2009.

[79] Won Kim, Soo Dong Kim, Eunseok Lee, and Sungyoung Lee. Adoption issues for cloud computing. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '09, pages 2–5, Kuala Lumpur, Malaysia, December 2009. Association for Computing Machinery. ISBN 978-1-60558-659-5. doi: 10.1145/1821748.1821751. URL https://doi.org/10.1145/1821748.1821751.

[80] la.dell. Vostro 470 Mini Tower Desktop Details – Ready For i7 QC Processors | Dell St. Kitts & Nevis, dec 2017. URL http://www1.la.dell.com/kn/en/corp/Desktops/vostro-470/pd.aspx?refid=vostro-470&s=corp.

[81] P. Lago, Q. Gu, and P. Bozzelli. A systematic literature review of green software metrics. 2014.

[82] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi. Safe limits on voltage reduction efficiency in GPUs: A direct measurement approach. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 294–307, December 2015. doi: 10.1145/2830772.2830811.

[83] lenovo thinkcentre. ThinkCentre M910 Tower | Power Your Business | Lenovo Australia, May 2018. URL https://www3.lenovo.com/au/en/desktops-and-all-in-ones/thinkcentre/.

[84] Tim Lewis. Openmp home, dec 2017. URL http://www.openmp.org/.

[85] Ding Li and William G. J. Halfond. An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 46–53, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2844-9. doi: 10.1145/2593743.2593750. URL http://doi.acm.org/10.1145/2593743.2593750.

[86] Xueliang Li and John P. Gallagher. A Source-level Energy Optimization Framework for Mobile Applications. In *16th International Working Conference on Source Code Analysis and Manipulation*, Raleigh, North Carolina, USA, October 2016. IEEE Computer Society.

[87] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 517–528, March 2016. doi: 10.1109/SANER.2016.85.

[88] Luís Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, Gilberto Melfe, and João Paulo Fernandes. On Haskell and energy efficiency. *Journal of Systems*

and Software, 149:554–580, March 2019. ISSN 0164-1212. doi: 10.1016/j.jss.2018.12.014.

[89] Jian Lin, Yu-dong Guo, Y. Man, and Shao-Huang Zhou. Executable Program Code Segment Address Randomization. *2015 International Conference on Computer Science and Applications (CSA)*, 2015. doi: 10.1109/CSA.2015.69.

[90] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 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, pages 2–11, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597085. URL http://doi.acm.org/10.1145/2597073.2597085.

[91] Jie Liu, Feng Zhao, and Aman Kansal. Virtual Machine Power Metering and Provisioning. *Microsoft Research*, June 2010.

[92] Kenan Liu, Gustavo Pinto, and Yu David Liu. Data-Oriented Characterization of Application-Level Energy Optimization. In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, number 9033 in Lecture Notes in Computer Science, pages 316–331. Springer Berlin Heidelberg, apr 2015. ISBN 978-3-662-46674-2 978-3-662-46675-9. DOI: 10.1007/978-3-662-46675-9_21.

[93] Irene Manotas, Lori Pollock, and James Clause. SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5.

[94] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An Empirical Study of Practitioners' Perspectives on Green Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 237–248, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1.

[95] Leo A. Meyerovich and Ariel S. Rabkin. Empirical Analysis of Programming Language Adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 1–18, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509515. event-place: Indianapolis, Indiana, USA.

[96] Junya Michanan, Rinku Dewri, and Matthew J. Rutherford. GreenC5: An adaptive, energy-aware collection for green software development. *Sustainable Computing: Informatics and Systems*, 13:42–60, November 2016. ISSN 2210-5379. doi: 10.1016/j.suscom.2016.11.004. URL http://www.sciencedirect.com/science/article/pii/S2210537916300403.

[97] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels.

In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 309–328, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1.

[98] Subrata Mitra, Manish K. Gupta, Sasa Misailovic, and Saurabh Bagchi. Phase-aware Optimization in Approximate Computing. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 185–196, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4931-8. URL http://dl.acm.org/citation.cfm?id=3049832.3049853.

[99] Sparsh Mittal and Jeffrey S. Vetter. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, May 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2442980. URL http://ieeexplore.ieee.org/document/7120149/.

[100] R. Mizouni, M. A. Serhani, R. Dssouli, A. Benharref, and I. Taleb. Performance Evaluation of Mobile Web Services. In *2011 IEEE Ninth European Conference on Web Services*, pages 184–191, September 2011.

[101] Mike Mol. Rosetta Code, Jan 2016. URL http://rosettacode.org/wiki/Rosetta_Code.

[102] Maurizio Morisio, Luca Ardito, Antonio Vetro', and Giuseppe Procaccianti. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. pages 34–39, 2013.

[103] I. Moura, G. Pinto, F. Ebert, and F. Castor. Mining energy-aware commits. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 56–67, May 2015. doi: 10.1109/MSR.2015.13.

[104] Lev Mukhanov, Dimitrios S. Nikolopoulos, and Bronis R. de Supinski. ALEA: Fine-Grain Energy Profiling with Basic Block Sampling. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 87–98, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-9524-3. doi: 10.1109/PACT.2015.16.

[105] S. Murugesan. Harnessing Green IT: Principles and Practices. *IT Professional*, 10(1): 24–33, jan 2008. ISSN 1520-9202. doi: 10.1109/MITP.2008.10.

[106] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2012. ISBN 161197206X, 9781611972061.

[107] A. Noureddine and A. Rajan. Optimising Energy Consumption of Design Patterns. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 2, pages 623–626, May 2015. doi: 10.1109/ICSE.2015.208.

[108] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. Runtime monitoring of software energy hotspots. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 160–169, sep 2012. doi: 10.1145/2351676.2351699.

[109] Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, and Lionel Seinturier. A Preliminary Study of the Impact of Software Engineering on GreenIT. pages 21–27, June 2012.

[110] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. A Review of Energy Measurement Approaches. *SIGOPS Oper. Syst. Rev.*, 47(3):42–49, November 2013. ISSN 0163-5980. doi: 10.1145/2553070.2553077. URL http://doi.acm.org/10.1145/2553070.2553077.

[111] L. H. Nunes, L. H. V. Nakamura, H. d F. Vieira, R. M. d O. Libardi, E. M. de Oliveira, J. C. Estrella, and S. Reiff-Marganiec. Performance and energy evaluation of RESTful web services in Raspberry Pi. In *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, pages 1–9, December 2014.

[112] PYPL PopularitY of Programming Language. PYPL PopularitY of Programming Language index, June 2018. URL http://pypl.github.io/PYPL.html.

[113] Wellington Oliveira, Renato Oliveira, and Fernando Castor. A Study on the Energy Consumption of Android App Development Approaches. MSR '17, pages 42–52, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-1544-7.

[114] Goland org. Go synchronization, May 2019. URL https://golang.org/pkg/sync/.

[115] James Pallister, Simon J. Hollis, and Jeremy Bennett. Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms. *The Computer Journal*, 58(1): 95–109, January 2015. ISSN 0010-4620. doi: 10.1093/comjnl/bxt129.

[116] S. Pandruvada. Running average power limit — rapl textbar 01.org, 2014. [Online]. Available: https://01.org/blogs/tlcounts/2014/running-average-power-limit---rapl. [Accessed: 2016-06-28].

[117] C. Pang, A. Hindle, B. Adams, and A. E. Hassan. What Do Programmers Know about Software Energy Consumption? *IEEE Software*, 33(3):83–89, May 2016. ISSN 0740-7459. doi: 10.1109/MS.2015.83.

[118] Juan Manuel Paniego, Silvana Gallo, Martín Pi Puig, Franco Chichizola, Laura De Giusti, and Javier Balladini. Analysis of RAPL Energy Prediction Accuracy in a Matrix Multiplication Application on Shared Memory. In Armando Eduardo De Giusti, editor, *Computer Science – CACIC 2017*, pages 37–46, Cham, 2018. Springer International Publishing. ISBN 978-3-319-75214-3.

[119] Thomas Pantels. Optimizing Power for Interactions between Virus Scanners and Pre-bundled Software, Jun 2015. URL https://software.intel.com/en-us/articles/optimizing-power-for-interactions-between-virus-scanners-and-pre-bundled-software.

[120] Thomas Pantels, Sheng Guo, and Rajshree Chabukswar. Touch Response Measurement, Analysis, and Optimization for Windows* Applications, apr 2014. URL https://software.intel.com/en-us/articles/touch-response-measurement-analysis-and-optimization-for-windows-applications.

[121] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. Harnessing Voltage Margins for Energy Efficiency in Multicore CPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 503–516, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4952-9. doi: 10.1145/3123939.3124537. URL http://doi.acm.org/10.1145/3123939.3124537.

[122] Jae Jin Park, Jang-Eui Hong, and Sang-Ho Lee. Investigation for Software Power Consumption of Code Refactoring Techniques. In Marek Reformat, editor, *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013*, pages 717–722. Knowledge Systems Institute Graduate School, 2014.

[123] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168841. URL http://doi.acm.org/10.1145/2168836.2168841.

[124] Tomasz Patyk, Harri Hannula, Pertti Kellomaki, and Jarmo Takala. Energy consumption reduction by automatic selection of compiler options. In *2009 International Symposium on Signals, Circuits and Systems*, pages 1–4, July 2009. doi: 10.1109/ISSCS.2009.5206106.

[125] Mathias Payer and Thomas R. Gross. Fine-grained User-space Security Through Virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 157–168, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0687-4. doi: 10.1145/1952682.1952703. URL http://doi.acm.org/10.1145/1952682.1952703. event-place: Newport Beach, California, USA.

[126] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, pages 15–21, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4161-5. doi: 10.1145/2896967.2896968. URL http://doi.acm.org/10.1145/2896967.2896968.

[127] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Fernandes, and João Saraiva. 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*, pages 256–267. ACM, October 2017. doi: 10.1145/3136014.3136031.

[128] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 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 2017, pages 256–267, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5525-4. doi: 10.1145/3136014.3136031.

URL http://doi.acm.org/10.1145/3136014.3136031. event-place: Vancouver, BC, Canada.

[129] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. jStanley: Placing a Green Thumb on Java Collections. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 856–859, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5937-5. doi: 10.1145/3238147.3240473.

[130] Peter A. H. Peterson, Digvijay Singh, William J. Kaiser, and Peter L. Reiher. Investigating Energy and Security Trade-offs in the Classroom with the Atom LEAP Testbed. In *Proceedings of the 4th Conference on Cyber Security Experimentation and Test*, CSET'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[131] S. Petridou and S. Basagiannis. Towards energy consumption evaluation of the SSL handshake protocol in mobile communications. In *2012 9th Annual Conference on Wireless On-Demand Network Systems and Services (WONS)*, January 2012. doi: 10.1109/WONS.2012.6152219.

[132] G. Pinto, F. Soares-Neto, and F. Castor. Refactoring for Energy Efficiency: A Reflection on the State of the Art. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software (GREENS)*, pages 29–35, May 2015. doi: 10.1109/GREENS.2015.12.

[133] G. Pinto, A. Canino, F. Castor, G. Xu, and Y. D. Liu. Understanding and overcoming parallelism bottlenecks in ForkJoin applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 765–775, October 2017. doi: 10.1109/ASE.2017.8115687.

[134] Gustavo Pinto. Refactoring Multicore Applications Towards Energy Efficiency. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, pages 61–64, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1995-9.

[135] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding Energy Behaviors of Thread Management Constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 345–360, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660235. URL http://doi.acm.org/10.1145/2660193.2660235.

[136] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining Questions About Software Energy Consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 22–31, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597110. URL http://doi.acm.org/10.1145/2597073.2597110.

[137] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding Energy Behaviors of Thread Management Constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 345–360, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660235.

[138] Gustavo Pinto, Kenan Liu, and Fernando Castor. A Comprehensive Study on the Energy Efficiency of Java Thread-Safe Collections. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution*, Raleigh, North Carolina, USA, 2016. IEEE Computer Society.

[139] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha. A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *IEEE Transactions on Mobile Computing*, 5(2), February 2006. ISSN 1536-1233. doi: 10.1109/TMC.2006.16.

[140] Giuseppe Procaccianti, Héctor Fernández, and Patricia Lago. Empirical evaluation of two best practices for energy-efficient software development. *Journal of Systems and Software*, 117:185–198, July 2016. ISSN 0164-1212. doi: 10.1016/j.jss.2016.02.035. URL http://www.sciencedirect.com/science/article/pii/S0164121216000777.

[141] Aleksandar Prokopec, Andrea Rosa, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomir Bulej, Yudi Zheng, Alex Villazon, Doug Simon, Thomas Wurthinger, and Walter Binder. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 31–47, Phoenix, AZ, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314637. URL https://doi.org/10.1145/3314221.3314637.

[142] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Rosa, S. Samsi, C. Yee, A. Reuther, and J. Kepner. Measuring the Impact of Spectre and Meltdown. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, September 2018. doi: 10.1109/HPEC.2018.8547554.

[143] L.B. Rall. *The Arithmetic of Differentiation*. MRC TECHNICAL SUMMARY REPORT. Mathematics Research Center, University of Wisconsin-Madison, 1984. URL https://books.google.gr/books?id=dIHZNQAACAAJ.

[144] M. Rashid, L. Ardito, and M. Torchiano. Energy Consumption Analysis of Algorithms Implementations. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–4, oct 2015. doi: 10.1109/ESEM.2015.7321198.

[145] M. Rashid, L. Ardito, and M. Torchiano. Energy Consumption Analysis of Algorithms Implementations. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–4, October 2015. doi: 10.1109/ESEM.2015.7321198.

[146] Raspberry.org. Raspberry Pi 3 Model B, Mar 2018. URL https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[147] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19,

pages 554–569, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359640. URL http://doi.acm.org/10.1145/3341301.3359640. event-place: Huntsville, Ontario, Canada.

[148] Haris Ribic and Yu David Liu. Energy-efficient Work-stealing Language Runtimes. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 513–528, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541971.

[149] Haris Ribic and Yu David Liu. Energy-efficient Work-stealing Language Runtimes. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 513–528, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5.

[150] Guenter Roeck. lm-sensors, jan 2018. URL https://github.com/groeck/lm-sensors.

[151] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 978-0-89791-216-7. URL http://dl.acm.org/citation.cfm?id=41765.41801.

[152] Rubén Saborido, Venera Arnaoudova, Giovanni Beltrame, Foutse Khomh, and Giuliano Antoniol. On the impact of sampling frequency on software energy measurements. *PeerJ PrePrints*, 3:e1219, 2015. doi: 10.7287/peerj.preprints.1219v2.

[153] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. In *2012 First International Workshop on Green and Sustainable Software (GREENS)*, pages 55–61, June 2012. doi: 10.1109/GREENS.2012.6224257.

[154] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How Does Code Obfuscation Impact Energy Usage? In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, September 2014. doi: 10.1109/ICSME.2014.35.

[155] Cagri Sahin, Lori Pollock, and James Clause. How Do Code Refactorings Affect Energy Usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 36:1–36:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2774-9. doi: 10.1145/2652524.2652538.

[156] Cagri Sahin, Lori Pollock, and James Clause. From benchmarks to real apps. *Journal of Systems and Software*, 117(C):307–316, July 2016. ISSN 0164-1212.

[157] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 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, pages 164–174, New York, NY, USA,

2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993518. URL http://doi.acm.org/10.1145/1993498.1993518.

[158] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. doi: 10.1145/2312005.2312018. URL http://doi.acm.org/10.1145/2312005.2312018.

[159] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM. ISBN 9781450304436. doi: 10.1145/2025113.2025133. URL http://doi.acm.org/10.1145/2025113.2025133.

[160] Nikolay A. Simakov, Martins D. Innus, Matthew D. Jones, Joseph P. White, Steven M. Gallo, Robert L. DeLeon, and Thomas R. Furlani. Effect of Meltdown and Spectre Patches on the Performance of HPC Applications. *arXiv:1801.04329 [cs]*, January 2018. URL http://arxiv.org/abs/1801.04329. arXiv: 1801.04329.

[161] Balaji Subramaniam and Wu-chun Feng. GBench: benchmarking methodology for evaluating the energy efficiency of supercomputers. *Computer Science - Research and Development*, 28(2-3):221–230, May 2012. ISSN 1865-2034, 1865-2042. doi: 10.1007/s00450-012-0218-0.

[162] Marek Suchanek, Milan Navratil, Don Domingo, and Laura Bailey. 4.4. CPU Frequency Governors, July 2018. URL https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-cpufreq.

[163] Phoronix Test Suite. Phoronix open-source, automated benchmarking, may 2019. URL https://www.phoronix-test-suite.com/.

[164] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, May 2013. doi: 10.1109/SP.2013.13. ISSN: 1081-6011.

[165] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, USA, 3rd edition, 2007. ISBN 978-0-13-600663-3.

[166] Tiobe. TIOBE Index | TIOBE - The Software Quality Company, Oct 2017. URL https://www.tiobe.com/tiobe-index/.

[167] Tiobe. Programming Languages Definition | TIOBE - The Software Quality Company, jan 2018. URL https://www.tiobe.com/tiobe-index/programming-languages-definition/.

[168] A. R. Tonini, L. M. Fischer, J. C. B. d Mattos, and L. B. d Brisolara. Analysis and Evaluation of the Android Best Practices Impact on the Efficiency of Mobile Applications.

In *2013 III Brazilian Symposium on Computing Systems Engineering*, pages 157–158, December 2013. doi: 10.1109/SBESC.2013.39.

[169] Anne E. Trefethen and Jeyarajan Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444–449, November 2013. ISSN 1877-7503. doi: 10.1016/j.jocs.2013.01.005. URL http://www.sciencedirect.com/science/article/pii/S1877750313000173.

[170] Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. Trends in worldwide ICT electricity consumption from 2007 to 2012. *Computer Communications*, 50:64–76, sep 2014. ISSN 0140-3664.

[171] Vassilis Vassiliadis, Charalampos Chalios, Konstantinos Parasyris, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Exploiting Significance of Computations for Energy-Constrained Approximate Computing. *International Journal of Parallel Programming*, 44(5): 1078–1098, October 2016. ISSN 1573-7640. doi: 10.1007/s10766-016-0409-6. URL https://doi.org/10.1007/s10766-016-0409-6.

[172] Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. Towards Automatic Significance Analysis for Approximate Computing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 182–193, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3778-6. doi: 10.1145/2854038.2854058. URL http://doi.acm.org/10.1145/2854038.2854058.

[173] WattsUpMeter. Watts up? Products: Meters, Oct 2017. URL https://www.wattsupmeters.com/secure/products.php?pn=0.

[174] Oliveira Wellington, Oliveira Renato, Castor Fernando, Fernandes Benito, and Pinto Gustavo. Recommending energy-efficient java collections. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.*, pages 160–170, 2019.

[175] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. Axilog: Language support for approximate hardware design. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 812–817, March 2015. doi: 10.7873/DATE.2015.0513.

[176] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design Test*, 34(2): 60–68, April 2017. ISSN 2168-2356. doi: 10.1109/MDAT.2016.2630270.

[177] Tomofumi Yuki and Sanjay Rajopadhye. Folklore Confirmed: Compiling for Speed = Compiling for Energy. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 169–184. Springer, Cham, September 2013. ISBN 978-3-319-09966-8 978-3-319-09967-5. doi: 10.1007/978-3-319-09967-5_10. URL https://link.springer.com/chapter/10.1007/978-3-319-09967-5_10.

[178] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-905-3. doi: 10.1145/1878961.1878982.

[179] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. *ACM SIGPLAN Notices*, 49(7):129–140, March 2014. ISSN 0362-1340. doi: 10.1145/2576195.2576208. URL `http://dl.acm.org/citation.cfm?id=2576195.2576208`.

Part B

List of Publications

B.1 Accepted publications based on this thesis

- Stefanos Georgiou and Diomidis Spinellis. Energy-Delay Investigation of Remote Inter-Process Communication Technologies. Accepted for publication in Elsevier Journal of Systems and Software 2019. https://doi.org/10.1016/j.jss.2019.110506
- Stefanos Georgiou, Stamatia Rizou, and Diomidis Spinellis. Software Development Life Cycle for Energy-Efficiency: Techniques and Tools. Accepted for publication in ACM Computing Surveys 2019. https://doi.org/10.1145/3337773
- Stefanos Georgiou, Maria Kechagia Panos Louridas, and Diomidis Spinellis. What Are Your Programming Language's Energy-Delay Implications? In 15th International Conference on Mining Software Repositories: Technical Track, MSR '18. New York, NY, USA, May 2018. Association for Computing Machinery. https://doi.org/10.1145/3196398.3196414

B.2 Submitted articles based on this thesis

- Stefanos Georgiou, Dimitris Mitropulos, Diomidis Spinellis. Energy-Efficient Computing in a Safe Environment.

B.3 Relevant accepted publications not part of the thesis

- Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. Analyzing Programming Languages' Energy Consumption: An Empirical Study. In PCI 2017: Proceedings of the 21st Pan-Hellenic Conference on Informatics, ACM International Conference Proceeding Series. ACM Press, September 2017. https://doi.org/10.1145/3139367.3139418