Universidade do Minho
Escola de Engenharia

Rui Alexandre Afonso Pereira

# Energyware Engineering: Techniques and Tools for Green Software Development

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

Universidade do Minho

universidade de aveiro

U.PORTO

Trabalho efectuado sob a orientação do
**Professor Doutor João Alexandre Saraiva**
e do
**Professor Doutor Jácome Cunha**

Julho de 2018

# Acknowledgements

First and foremost, I would like to express my most sincere and deepest gratitude to my excellent supervisors, Prof. João Saraiva and Prof. Jácome Cunha, for the endless support, knowledge, dedication, professionalism and, above all, patience throughout these last several years. They were the most significant factor in leading me on this amazing path during this thesis, with continuous motivation, which kept pushing me to go further than I expected to go, and continuous confidence showed in me and my capabilities. From a MSc to a PhD, I hope these were just the initial collaborations on a long road ahead.

I would also like to thank João Paulo Fernandes for all the helpful and insightful comments, ideas, and collaborations throughout this amazing journey. A very special and meaningful thanks to Marco Couto who was a great teammate, and collaborated with me alongside many projects, ideas, and discussions during these years. Couldn't had hoped for a better research team. I too hope to continue future collaborations with both of you. I would also like to thank Luis Cruz for the help he provide often times and the discussions we had.

A very heartfelt thank you to my parents, grandparents, and girlfriend for all the endless support, confidence, and happiness given to me throughout these years, in face of countless struggles. All have contributed to where I am today, and each have given me motivation at different points in time to continue pushing through when it was much needed.

Obviously, last but not least, a gracious thank you to all my lab and non-lab friends whom have put up with my antics on a regular basis and kept me from going insane. You all contributed to the amazing and relaxed atmosphere I was subject to through these years, and contribute to the hilarious and memorable moments in this chapter of my life.

Finally, this work is partially funded by FCT – Foundation for Science and Technology, the Portuguese Ministry of Science, Technology and Higher Education, through national funds, and co-financed by the European Social Fund (ESF) through the Operacional Programme for Human Capital (POCH), with scholarship reference SFRH/BD/112733/2015. Additionally, funding was also provided the ERDF – European Regional Development Fund – through the Operational Programmes for Competitiveness and Internationalisation COMPETE and COMPETE 2020, and by the Portuguese Government through FCT project Green Software Lab (ref. POCI-01-0145-FEDER-016718), by the project GreenSSCM - Green Software for Space Missions Control, a project financed by the Innovation Agency, SA, Northern Regional Operational Programme, Financial Incentive Grant Agreement under the Incentive Research and Development System, Project No. 38973, and by the Luso-American Foundation in collaboration with the National Science Foundation with grant FLAD/NSF ref. 300/2015 and ref. 275/2016.

REPÚBLICA PORTUGUESA

POCH PROGRAMA OPERACIONAL CAPITAL HUMANO

UNIÃO EUROPEIA
Fundo Social Europeu

FCT
Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

COMPETE
PROGRAMA OPERACIONAL FACTORES DE COMPETITIVIDADE

QREN
QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL
PORTUGAL 2007.2013

COMPETE 2020

PORTUGAL 2020

UNIÃO EUROPEIA
Fundo Europeu de Desenvolvimento Regional

fundação
LUSO-AMERICANA
PARA O DESENVOLVIMENTO

# Abstract

*Energy consumption is nowadays one of the most important concerns worldwide. While hardware is generally seen as the main culprit for a computer's energy usage, software too has a tremendous impact on the energy spent, as it can cancel the efficiency introduced by the hardware. Green Computing is not a new field of study, but the focus has been, until recently, on hardware. While there has been advancements in Green Software techniques, there is still not enough support for software developers so they can make their code more energy-aware, with various studies arguing there is both a lack of knowledge and lack of tools for energy-aware development.*

*This thesis intends to tackle these two problems and aims at further pushing forward research on Green Software. This software energy consumption issue is faced as a software engineering question. By using systematic, disciplined, and quantifiable approaches to the development, operation, and maintenance of software we defined several techniques, methodologies, and tools within this document. These focus on providing software developers more knowledge and tools to help with energy-aware software development, or Energyware Engineering.*

*Insights are provided on the energy influence of several stages performed during a software's development process. We look at the energy efficiency of various popular programming languages, understanding which are the most appropriate if a developer's concern is energy consumption. A detailed study on the energy profiles of different Java data structures is also presented, along with a technique and tool, further providing more knowledge on what energy efficient alternatives a developer has to choose from. To help developers with the lack of tools, we defined and implemented a technique to detect energy inefficient fragments within the source code of a software system. This technique and tool has been shown to help developers improve the energy efficiency of their programs, and even outperforming a runtime profiler.*

*Finally, answers are provided to common questions and misconceptions within this field of research, such as the relationship between time and energy, and how one can improve their software's energy consumption.*

*This thesis provides a great effort to help support both research and education on this topic, helps continue to grow green software out of its infancy, and contributes to solving the lack of knowledge and tools which exist for Energyware Engineering.*

# Resumo

*Hoje em dia o consumo energético é uma das maiores preocupações a nível global. Apesar do hardware ser, de uma forma geral, o principal culpado para o consumo de energia num computador, o software tem também um impacto significativo na energia consumida, pois pode anular, em parte, a eficiência introduzida pelo hardware. Embora Green Computing não seja uma área de investigação nova, o foco tem sido, até recentemente, na componente de hardware. Embora as técnicas de Green Software tenham vindo a evoluir, não há ainda suporte suficiente para que os programadores possam produzir código com consciencialização energética. De facto existem vários estudos que defendem que existe tanto uma falta de conhecimento como uma escassez de ferramentas para o desenvolvimento energeticamente consciente.*

*Esta tese pretende abordar estes dois problemas e tem como foco promover avanços em green software. O tópico do consumo de energia é abordado duma perspectiva de engenharia de software. Através do uso de abordagens sistemáticas, disciplinadas e quantificáveis no processo de desenvolvimento, operação e manutencão de software, foi possível a definição de novas metodologias e ferramentas, apresentadas neste documento. Estas ferramentas e metodologias têm como foco dotar de conhecimento e ferramentas os programadores de software, de modo a suportar um desenvolvimento energeticamente consciente, ou Energyware Engineering.*

*Deste trabalho resulta a compreensão sobre a influência energética a ser usada durante as diferentes fases do processo de desenvolvimento de software. Observamos as linguagens de programação mais populares sobre um ponto de vista de eficiência energética, percebendo quais as mais apropriadas caso o programador tenha uma preocupação com o consumo energético.*

*Apresentamos também um estudo detalhado sobre perfis energéticos de diferentes estruturas de dados em Java, acompanhado por técnicas e ferramentas, fornecendo conhecimento relativo a quais as alternativas energeticamente eficientes que os programadores dispõem. Por forma a ajudar os programadores, definimos e implementamos uma técnica para detetar fragmentos energicamente ineficientes dentro do código fonte de um sistema de software. Esta técnica e ferramenta têm demonstrado ajudar programadores a melhorarem a eficiência energética dos seus programas e em algum casos superando um runtime profiler.*

*Por fim, são dadas respostas a questões e conceções erradamente formuladas dentro desta área de investigação, tais como o relacionamento entre tempo e energia e como é possível melhorar o consumo de energia do software.*

*Foi empregue nesta tese um esforço árduo de suporte tanto na investigação como na educação relativo a este tópico, ajudando à maturação e crescimento de green computing, contribuindo para a resolução da lacuna de conhecimento e ferramentas para suporte a Energyware Engineering.*

# Contents

# Acronyms

**CLBG** Computer Language Benchmark Game

**IoT** Internet of Things

**IT** Information Technology

**JCF** Java Collection Framework

**RAPL** Runtime Average Power Limiter

**SFL** Spectrum-based Fault Localization

**SPELL** Spectrum-based Energy Leak Localization

**SWEBOK** Software Engineering Body of Knowledge

**TRQ** Thesis Research Question

# List of Figures

# List of Tables

# Chapter 1

# Introduction

*This chapter briefly introduces the concept and research area of this thesis known as Green Computing. We look at how the energy consumption problem is a major concern for our environment, with a specific emphasis on the negative contributions from information technology (IT).*

*It follows with a look on the specific motivation behind our research, a sub-area known as Green Software, where we aim to tackle the energy consumption problem in IT through a software-based approach. By focusing on reducing energy consumption through software analysis and optimization, one can heavily increase IT energy efficiency. Additionally, we see how the lack of knowledge and tools for energy efficient software development are also an issue.*

*Finally, this thesis's research questions are presented along with an explanation of their reasoning and importance, a list of all the contributions which originated during this thesis's development, and this document's structure.*

## 1.1 Green Computing

Ever since the industrial revolution, society has built its intensive development pace on top of the widespread use of energy resources. The problem is that the growing energy demands have significant side-effects such as global warming, but even more importantly that it is simply not able to match such demands on the production side.

For the last couple of years, the world has begun to express heavy concern over the impact it has done to the environment with activities such as oil-drilling, CO2 emissions, and energy consumption. Yet, even though environmentalism and computer sciences are usually mentioned separately, the world has come to realize the annually increasing impact that information technology has on the environment with its emissions of greenhouse gases and electrical costs, something we cannot ignore. This has raised awareness in multiple contexts (Tiwari et al., 1994; Yuan and Nahrstedt, 2003), that recently include the community acknowledgment of the need for sustainable software development (Becker et al., 2014).

This can be attributed to the exponential growth we are witnessing in the Information and Communications Technology (ICT) sector. Almost everyone has access to a computer, and the Internet is virtually accessible everywhere. While undoubtedly a milestone in our era, all of this occurs at the expense of high energy costs needed to supply servers, data centers, and any use of computers (Guelzim and Obaidat, 2013; Gelenbe and Caseau, 2015). This cost continues to increase as the demand for more cloud services rises and the traditional technologies are migrating from local servers to remote servers (Ricciardi et al., 2013; Mouftah and Kantarci, 2013).

Koomey has reported that electrical energy consumption by IT was estimated to be between 1.1% and 1.5% of the world's emissions, nearly 1 gigaton of emissions a year (Koomey, 2011), and with the increasing demand for computation and data storage, emissions will increase to 1.54 gigatons, or 3% of global emissions by 2020.

IT businesses have also come to realize this impact, along with another major concern: energy costs. There are studies suggesting that in average, close to 50% of the energy costs of an organization can be attributed to the IT departments (Harmon and Auseklis, 2009).

Step-by-step, some businesses have begun to promote *Green* initiatives, in hopes of reducing the emissions and energy costs. For example, looking at the data center sector, Symantec Corporation[1], a software manufacturing company with massive data-centers around the globe, decided to reduce CO2 emissions by 15% by 2012 (Symantec, 2008). To accomplish this goal, Symantec decided to look at the Sunnyvale Data Center, and after finding out 60% of end user's computers were left powered on overnight, they decided to place users' computers in stand-by mode after four hours of inactivity (Thompson, 2008). These two steps helped reduce approximately $2 million and over 6 million kilowatts of energy.

But sometimes, businesses cannot just physically reduce consumption as Symantec did, so now there are two options to turn to: physical IT components (hardware) and code (software). Much research has been done with hardware and other IT components in regards of reducing energy, such as new hardware level designs and energy efficient architecture (Su et al., 1994; Douglis et al., 1995; Kravets and Krishnan, 1998; Delaluz et al., 2001; Iyer and Marculescu, 2001; Chandrakasan et al., 1992; David et al., 2010), system level designs (Ribic and Liu, 2014; Bartenstein and Liu, 2014), optimizing operating system's energy usage (Flautner et al., 2001; Pettis et al., 2006; Pering et al., 2000), or even data center layouts for efficient cooling. For example, Google uses a customized evaporative cooling to reduce its data centers' energy consumption (Hooper, 2008), and Shukla (2012) proposes some hardware design techniques for power reduction, and shows how using OLEDs instead of other light sources also significantly reduces power reduction in data centers. Kerstens and DuChene (2008) presented a way to use Ganglia Moab to save power in data centers by detecting hotspots in clusters and spreading out the workload to unused nodes, or even shutting them down if they are not needed at certain times.

Yet while great advances have been made to decrease energy consumption and emissions of hardware, these Green IT initiatives stem from reducing energy loss in the power supply chain (Brown et al., 2008). And even so, energy consumption keeps rising steeply, which shows that the rising demand is exceeding efficiency improvement, and the window for optimization is closing.

[1] https://www.symantec.com/

Additionally, energyefficiency in the hardware level is canceled out by poorly optimized software, where "energy used by ICT hardware can be attributed to software" (Standard, 2017). This is not shocking, since software development has always focused on response efficiency/functionality, rather than the minimization of power consumption (Wirth, 1995), making software design/construction energy-unaware.

Such as how goals of optimizing performance or memory usage cannot be achieved by purely looking at low-level systems and hardware, optimizing energy consumption cannot take the same route. Thus, there is a great need to turn to software code, and apply practices of using our computational resources in a more efficient manner, while maintaining or increasing their overall performance. In other words, we need to look into **green software** practices.

## 1.2 Motivation - Green Software

This section details the motivation for working on *Green Software*, including the work presented in this thesis.

The *Green Software* field focuses on reducing energy consumption through software analysis and optimization. It is known that "even small inefficiencies in apps add up across the system, significantly affecting battery life" (Pinto and Castor, 2017). While still in its early years, *Green Software* research has grown significantly. There are countless works showing how having knowledge on software energy efficiencies and inefficiencies can make a significant impact on the energy consumption. Studies have shown how different programming languages (Couto et al., 2017b; Oliveira et al., 2017; Pereira et al., 2017b; Georgiou et al., 2018), design patterns (Sahin et al., 2012; Bunse and Stiemer, 2013), sorting algorithms (Bunse et al., 2009b,a), multicore smartphones (Li and Mishra, 2016), software testing (Li et al., 2014; Jabbarvand et al., 2016), Android API and advertisements (Linares-Vásquez et al., 2014; Jabbarvand et al., 2015; Rasmussen et al., 2014; Cruz and Abreu, 2017; Couto et al., 2015), software version changes (Hindle, 2015), code obfuscations (Sahin et al., 2016), machine-learning algorithms (McIntosh et al., 2018), refactorings and transformations (Brandolese et al., 2002;

Sahin et al., 2014; Park et al., 2014), and different Java based collections (Pereira et al., 2016; Manotas et al., 2014; Pinto et al., 2016; Hasan et al., 2016) have a statistically significant impact on energy usage.

Various energy consumption measurement and estimation software tools exist (Grech et al., 2015; Stulova et al., 2016; Li et al., 2013; Liqat et al., 2013; Noureddine et al., 2015; Hao et al., 2013; Pathak et al., 2012; Hoque et al., 2015; Chowdhury and Hindle, 2016). Unfortunately, most do not provide guidance or insights on understanding what they can do to reduce energy consumption, and most programmers don't even know these exist or how they can use them (Pang et al., 2016).

As researchers, we continuously face these problems of how to analyze, interpret, and optimize energy consumption. However, these problems also extend to programmers. In fact, studies (Manotas et al., 2014, 2016; Pang et al., 2016; Pinto et al., 2014a) have shown that programmers are very concerned with the energy consumption power of their applications, many times seeking help. In fact, there are many misconceptions within the programming community as to what causes high energy consumption, how to solve these issues, and have expressed a heavy lack of support and knowledge for energy-aware development.

Pinto and Castor (2017) and Manotas et al. (2016) argue that there are two main problems in regards to energy efficient software development: **the lack of knowledge** and **the lack of tools**. These two problems are what motivates this thesis. Additionally, they also discuss twelve different paths for future research in green software, organized according to the Software Engineering Body of Knowledge structure (SWEBOK) (Bourque et al., 1999). Of these listed paths, the work produced during this thesis also touches on:

- *Software Tools & Methods: Static Analysis Tools*
- *Software Maintenance: Refactoring*
- *Software Design & Construction*
- *Software Design & Construction: Data structures*
- *Software Quality & Testing: Software Debugging*

## 1.3 Research Questions

Since the beginning of this thesis, research in green-computing, green-software, and energy efficient software systems has been both in its infancy and yet, also growing rapidly.

We wish to provide developers knowledge on what practices can improve or harm the energy efficiency of their programs. The main strategy is to approach the software energy consumption problem as a software engineering problem. The solution comes by using systematic, disciplined, and quantifiable approaches to the development, operation, and maintenance of software, supplemented with a focus on energy consumption. We define this software engineering discipline as *Energyware Engineering*.

While many questions have collaterally and spontaneously arisen during this work, the following Thesis Research Questions (TRQ) have been the main focus:

- **TRQ1**: *What influence do different programming languages have on energy consumption?* Properly understanding the differences in energy consumption between different programming languages will allow both researchers and programmers the means to compare the energy efficiency of popular languages. In turn, this will allow energy-aware decision making during the initial steps of developing software: choosing a language.

- **TRQ2**: *Can fault localization techniques be adapted to detect energy hotspots in source code?* Fault localization is traditionally used to statically identify program faults or bugs within a program's source code. By drawing a parallel between fault detection and energy hotspot detection, a programmer would be able to statically locate what sections within their program are causing energy inefficiency. Having such a technique and tool available would both help developers in Energyware Engineering, and also further close the absence of tools aimed at energy-aware programming.

- **TRQ3**: *What influence do different Java data structures and their methods have on energy consumption?* As in **TRQ1**, fully understanding the energy impact coming from different data structures, or Collections in this case, will allow one to be more energy conscientious during the development of a program. If we were able to know which Collections are energy-greedy, alternatives can be easily suggested. Additionally, if looking down one level at the Collection's methods, we can have an even deeper understanding of what scenarios (methods) contribute more or less to energy consumption, and offer more fine-tuned suggestions.

## 1.4 Contributions

The main scientific contributions of this thesis are divided in three principle topics:

**Programming Language Energy Efficiency** A study on the energy efficiency of 27 popular programming languages, across 10 different set of problems. This study focuses on the *lack of knowledge*, and helps developers choose a programming language if energy consumption is a concern. Additionally, performance and memory usage are also analyzed, and the relationship between these three (energy, performance, and memory) categories are also compared. This work, described in Chapter 3, was also published in:

- **Towards a Green Ranking for Programming Languages** *(Best Paper)* – Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, João Saraiva. In *Brazilian Symposium on Programming Languages*, 2017. (Couto et al., 2017b)

- **Energy Efficiency across Programming Languages** *(Submitted)*– Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva. In *International Conference on Software Language Engineering*, 2017. (Pereira et al., 2017b)

- **Ranking Programming Languages by Energy Efficiency** – Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva. Submitted to *Journal of Science of Computer Programming*, 2018.

Additionally, a public repository containing the complete set of tools and programs used is available at:

- `https://github.com/greensoftwarelab/Energy-Languages`

**Spectrum-based Energy Leak Localization** The development of a language and context independent technique, based on using Spectrum-based Fault Localization, to point to energy inefficient blocks, or what we call *energy leaks*, within source-code. The result is an energy ranking of energy inefficient source code fragments. This technique has been implemented and evaluated in a prototype toolkit. This work, described in Chapter 4, focused on the *lack of tools*, and has been partially published in:

- **Locating Energy Hotspots in Source Code** – Rui Pereira. In *International Conference on Software Engineering - ACM Student Research Competition*, 2017. (Pereira, 2017)

- **Helping Programmers Improve the Energy Efficiency of Source Code** – Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, João Saraiva. In *International Conference on Software Engineering Companion* 2017. (Pereira et al., 2017a)

The *SPELL* prototype toolkit can be found at:

- `https://github.com/greensoftwarelab/SPELL`

**Java Collection Framework Energy Efficiency** A study on the energy efficiency of 43 methods, across 22 different Java collections, divided into three groups (Sets, Lists, and Maps), and varying population sizes. This work allows us to give developers more information on which data structures to choose if their concern is energy efficiency, based on what methods will be used within the program. Additionally, we developed a tool, jStanley, to automatically detect all the used collections and their used methods, and suggest a better (energy or time) alternative based on our methodology. This work focused on the *lack of knowledge* and *lack of tools*, and has been published in:

- **The Influence of the Java Collection Framework on Overall Energy Consumption** – Rui Pereira,Marco Couto, Jácome Cunha, João Paulo Fernandes, João Saraiva. In *International Workshop on Green and Sustainable Software*, 2016. (Pereira et al., 2016)

- **jStanley: Placing a Green Thumb on Java Collections** – Rui Pereira, Pedro Simão, Jácome Cunha, João Saraiva. In *33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018. (Pereira et al., 2018)

The interactive data tables, the jStanely prototype, and public repository for future research can be respectively found at:

- `http://greenlab.di.uminho.pt/collections/`

- `https://github.com/greensoftwarelab/jStanley`

- `https://github.com/greensoftwarelab/Collections-Energy-Benchmark`

**Green Computing** In addition to the prior, further work and activities on *Green Computing* included:

- **GreenDroid: A tool for Analyzing Energy Consumption in the Android Ecosystem** – Marco Couto, Jácome Cunha, João Paulo Fernandes, Rui Pereira, João Saraiva In *International Scientific Conference on Informatics* 2015. (Couto et al., 2015)

- **Static Energy Consumption Analysis in Variability Systems** – Marco Couto, Jácome Cunha, João Paulo Fernandes, Rui Pereira, João Saraiva. In *2nd Green in Software Engineering Workshop*, 2016. (Couto et al., 2016)

- **Products go Green: Worst-Case Energy Consumption in Software Product Lines** – Marco Couto, Rui Pereira, Paulo Borba, Jácome Cunha, João Paulo Fernandes, João Saraiva. In *International Systems and Software Product Line Conference*, 2017. (Couto et al., 2017a)

In summary, the list of contributions of this thesis include:

- A large-scale analysis on the energy efficiency of 27 programming languages.
- Public repository for future research on the energy efficiency of programming languages.
- Development of a language and context independent technique to detect energy inefficiencies within source-code.
- Development and evaluation of a prototype toolkit, termed *SPELL*, implementing the previously mentioned technique.
- An analysis on the energy profiles of methods from the Java Collection Framework (JCF).
- Public repository for future research on the energy efficiency of the Java Collection Framework.
- A methodology to use the previously defined JCF energy profiles to choose a more efficient solution.
- Online interactive energy data tables for the Java Collection Framework.
- Development and evaluation of a prototype, termed *jStanely*, implementing the previous methodology, suggesting more efficient solutions, and automatically changing the source code to the suggested alternative.

## 1.5 Document Structure

This document describes the work accomplished while developing this thesis. The contents in this document are organized as follows:

**Chapter 1** Introduces the topics and motivation of this thesis, the goals which this thesis aims to accomplish, the research questions which we aim to answer, and the list of contributions accomplished during this thesis.

**Chapter 2** Touches on the State of the Art directly pertaining to topics presented in this document.

**Chapter 3** Presents a large scale analysis on the energy efficiency on 27 popular programming languages.

**Chapter 4** Describes a language and context independent technique to localize energy hotspots in source code, along with an implementation of the technique in a tool called *SPELL*.

**Chapter 5** Details a study on the energy efficiency of the Java Collection Framework on a method level, presenting a methodology to help programmers choose the most appropriate collection for their program, and the implementation of the *jStanely* tool.

**Chapter 6** Discusses some of the interesting observations found during work on this thesis, and empirical studies. The analyses and tests presented in this chapter are preliminary in nature. They show very promising energy efficient approaches to certain issues, but nevertheless, a more thorough validation is needed.

**Chapter 7** Final thoughts, conclusions, and possible future lines of work based on the contributions from this thesis.

# Chapter 2

# State of the Art

*In this chapter, we will look into what has been done thus far on the three main research topics worked on during this thesis. The first section will focus on research into understanding the energy efficiency and costs of programming languages. Afterwards, we will look into what techniques and tools exist to help developers detect and understand where are problematic (energy inefficient) code fragments in their source code. Finally, we will look into research in regards to understanding how different data structures compare to each other in terms of energy consumption, and what solutions exist to help produce more energy efficient programs.*

## 2.1 Programming Languages and Energy Efficiency

There has been previous works based on analyzing the performance of programming languages (Nanz and Furia, 2015; Pankratius et al., 2012; Williams et al., 2010; St-Amour et al., 2012), but only recently has research been done on programming language energy efficiency.

Researchers (Nobre et al., 2018) have observed how different compiler optimizations impact the energy consumption within the `C` programming language. They were able to observe a reduction in the energy consumption by up to 24%, and showed how only some of these improvements could be explained by improvements in execution time. Additionally, their experiments show cases where applications were faster, yet consumed more energy.

Lima et al. (2016) looked at the energy behavior of a purely functional and lazy language, Haskell, using extracted examples from *Rosetta Code* [1] and *The Computer Language Benchmark Game* [2] (CLBG). In this work, the authors chose to look at the energy efficiency of Haskell from both a strictness and concurrency perspective, while also analyzing the energy influence of small implementation changes (which will be detailed in Subsection 2.3).

Oliveira et al. (2017) took a different approach, and compare `JavaScript`, `Java`, and `C++` in an Android setting. As with the previous work, these authors also used examples from both Rosetta Code and the CLBG along with a few examples from the *F-Droid* repository [3]. In this setting, they found that in terms of energy consumption there is no overall winner. Depending on the tasks at hand, the three languages fluctuated. They also showed that development across different devices had very little differences, supporting that an approach consuming less energy on one device should translate to another. Additionally, they also found instances where a faster implementation did not translate into a more energy efficient one, which is also something reoccurring throughout this thesis.

A more recent effort by Georgiou et al. (2018) calculated the Energy Delay Product on 14 programming languages performing 25 different tasks. For this study, they obtained the programs from *Rosetta Code*, and ran the study on three different platforms: a server, a laptop,

[1] Rosetta Code: `http://rosettacode.org`

[2] The Computer Language Benchmarks Game: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

[3] F-Droid: `https://f-droid.org/en/`

and an embedded system. Their analysis was focused on the different types of tasks such as file handling, I/O-intensive operations, arithmetic, etc. They showed that while the different platforms often times produced different rankings, there was no statistical evidence that there is a difference between the embedded and the server or the laptop and the server. Additionally, their study showed that compiled languages outperformed the others both in terms of energy consumption and performance.

## 2.2 Energy Inefficient Code Detection

It is common for software developers to use debugging tools and profilers to help detect bugs or performance inefficient code fragments. Applying these concepts to help detect energy inefficient code fragments is a much more challenging task. There is still very little knowledge as to what can be directly done, from a software developers position, to manipulate and improve energy consumption. Even if a developer takes the steps and effort to use one of the many energy/power measuring devices, a lot has to be taken into account such as the contextual information about what the program is supposed to be doing, or where it was executed. Thus, this challenging problem has attracted several researchers to propose solutions, but with a focus on mobile applications.

Ma et al. (2013) presented a tool, *eDoctor*, for mobile users to troubleshoot any irregular battery draining issues they were having on their smartphones. The authors' tool analyzes a mobile application's behavior, and identify abnormalities. It then suggests the user the most appropriate repair solutions, such as disabling device locations, downgrade to previous versions, turn on airplane mode, etc. A different approach was done by Oliner et al. (2013), where a black-box diagnostic is performed. The client application sends coarse-grained measurements to a server where the data is correlated with client properties (for example running applications). It then suggests actions the user may take to improve battery life.

While the previous tool was for end-users, Pathak et al. (2012) focused their *eProf* tool for developers. *eProf* is a fine-grained energy profiler for smartphone apps. It instruments app binaries with system-call and routine tracing, and uses this information with their self proposed power model to understand where the energy is spent in the app. The developer is then shown what external modules are used, and attributes a percentage to different tasks and events performed such as HTTP requests, game rendering, map downloading, user tracking, etc. The developer can then use this information to modify their application to optimize the energy inefficient blocks. Additionally, the authors found that "wakelock bugs" and I/O events were the most energy draining issues.

Linares-Vásquez et al. (2014) conducted a large empirical study on API calls and usage patterns, within the Android development framework, to find which have a tendency to have high consumption costs. Their study was conducted on 55 different apps, looking into 807 different API methods and defined 131 as *energy-greedy APIs*. Similarly, Liu et al. (2014) analyzed 402 different Android applications and found that there were two main causes of energy problems: missing deactivation of sensors or wake locks, and cost ineffective use of sensory data. In response, they developed *GreenDroid*, a tool to identify these two problems to further help developers find these issues.

Two similar and complementary works (Cruz and Abreu, 2017; Banerjee and Roychoudhury, 2016), also within the Android domain, defined energy efficient guidelines for mobile development. The former was based on performance guidelines for mobile and focused on code smells affecting CPU usage. The latter focus on resource usage, leakage, and sensors.

Banerjee et al. (2014) constructed an automated test generation framework to produce tests simulating user interactions, such as touches or taps, to heavily stress I/O components. As these tend to be one of the main causes of high energy consumption in mobile, they are able to capture possible energy hotspots/energy leaks within the application.

Couto et al. (2014) presented a technique where they relate the energy consumption to the source code of the application while giving classifications of methods as Red, Yellow, or Green. They do so by running each test case twice on the program, where first they log the stack trace of each test, and then they log the energetic values for a test. By correlating the stack trace with the energy values, and using thresholds, they classify the tests as Red, Yellow, or Green. Finally, depending on what methods were called in those tests, they also classify each test as Red, Yellow, or Green.

Recently, Verdecchia et al. (2018) presented a naive spectrum-based fault localization technique aimed to efficiently locate energy hotspots in source code. Their work is very closely related to the one presented in Chapter 4. The authors state that while our contribution lies more in providing the means to precisely locate energy hotspots in source code, their work aims to investigate if more naive approaches can be used to locate them. Thus, understanding both sides, research can be further done on finding the best balance of performance and precision.

## 2.3 Data Structures and Energy Efficiency

When developing software for commercial use, research, or toy programs, a programmer almost always has at least one data structure involved. These common programming storage formats are able to be implemented in many different ways, each with its own benefits, drawbacks, and performance. In recent years however, analyzing data structures and their energy efficiency has attracted the attention of many researchers.

Pinto et al. (2016) specifically studied the energy efficiency on Java's thread-safe collections, based on traversal, insertion, and removal operations. They were able to improve up to 17% energy savings by switching out collections, showing how such simple changes can reduce the energy consumption considerably.

Another study specifically focused on different map data structures in Android Saborido et al. (2018). They analyzed the CPU time, memory usage, and energy consumption in `HashMap`, `ArrayMap`, and `SpareArray` variants. Finally, they offered guidelines for Android developers for choosing the most appropriate choice if the developer is worried about energy usage.

Lima et al. (2016) analyzed the energy behavior of various Haskell sequential and concurrent data structures. They too were able to show how making changes on which data structures are used can have large impacts, saving up to 60% of energy in one of their settings. Finally, they argue that tools to support developers in quickly refactoring a program to switch between such primitives can be of great help if energy is a concern.

Hasan et al. (2016) looked at Java collections from the Java Collections Framework (8 collections), Apache Commons Collections (5 collections), and Trove (4). They measured the energy costs of iterations, insertions (beginning, middle, and end for *Lists*), and random access/query. They divided the collections into three groups, *Sets*, *Lists*, and *Maps*, representing the three possible collection interfaces. They were able to see that there are differences in the energy consumption profiles of collections within a group, and showed how one can use those profiles to choose a more energy efficient alternative. A study they performed showed how switching out one *List* for a worse one can decrease energy consumption by 300%, or improve the energy consumption by 36%.

Finally, Manotas et al. (2014) developed the *SEEDS* framework. This was the first automated support for optimizing the energy usage of applications by making code-level changes. A specific instantiation of this framework was presented by the authors to improve the energy consumption of projects using Java's Collections API, producing good results. *SEEDS* is a dynamic approach which follows a trial and error method, testing each possible alternative, until the most energy efficient one is found.

# Chapter 3

# Energy Efficiency Across Programming Languages

*This chapter presents a study of the runtime, memory usage and energy consumption of twenty seven well-known software languages. We monitor the performance of such languages using ten different programming problems, expressed in each of the languages. Our results show interesting findings, such as, slower/faster languages consuming less/-more energy, and how memory usage influences energy consumption. We show how to use our results to provide software engineers support to decide which language to use when energy efficiency is a concern.*

## 3.1 Introduction

Software language engineering provides powerful techniques and tools to design, implement and evolve software languages. Such techniques aim at improving programmers productivity - by incorporating advanced features in the language design, like for instance powerful modular and type systems - and at efficiently execute such software - by developing, for example, aggressive compiler optimizations. Indeed, most techniques were developed with the main goal of helping software developers in producing faster programs. In fact, in the last century *performance* in software languages was in almost all cases synonymous of *fast execution time* (embedded systems were probably the single exception).

In this century, this reality is quickly changing and software energy consumption is becoming a key concern for computer manufacturers, software language engineers, programmers, and even regular computer users. Nowadays, it is usual to see mobile phone users (which are powerful computers) avoiding using CPU intensive applications just to save battery/energy. While the concern on the computers' energy efficiency started by the hardware manufacturers, it quickly became a concern for software developers too (Pinto et al., 2014a). In fact, this is a recent and intensive area of research where several techniques to analyze and optimize the energy consumption of software systems are being developed. Such techniques already provide knowledge on the energy efficiency of data structures (Pereira et al., 2016; Hasan et al., 2016) and android language (Oliveira et al., 2017), the energy impact of different programming practices both in mobile (Li and Halfond, 2014; Sahin et al., 2012; Linares-Vásquez et al., 2014) and desktop applications (Sahin et al., 2014; Pereira et al., 2017a), the energy efficiency of applications within the same scope (Chowdhury and Hindle, 2016; Jabbarvand et al., 2015), or on how to predict energy consumption in several software systems (Hao et al., 2013; Couto et al., 2017a), among with several other works.

An interesting question that frequently arises in the software energy efficiency area is whether *a faster program is also an energy efficient program*, or not. If the answer is yes, then optimizing a program for speed also means optimizing it for energy, and this is exactly what the compiler construction community has been hardly doing since the very beginning of software languages. However, energy consumption does not depend only on execution time, as shown in the equation:

$$E_{nergy} = T_{ime} \times P_{ower} \tag{3.1}$$

In fact, there are several research works showing different results regarding this subject (Yuki and Rajopadhye, 2014; Pinto et al., 2014b; Trefethen and Thiyagalingam, 2013a; Lima et al., 2016; Pereira et al., 2016; Abdulsalam et al., 2015).

A similar question arises when comparing software languages: *is a faster language, a greener one?* Comparing software languages, however, is an extremely complex task, since the performance of a language is influenced by the quality of its compiler, virtual machine, garbage collector, available libraries, etc. Indeed, a software program may become faster by improving its source code, but also by "just" optimizing its libraries and/or its compiler. This chapter focuses on answering **TRQ1**: *What influence do different programming languages have on energy consumption?*.

In this chapter we analyze the performance of twenty seven software languages. We consider ten different programming problems that are expressed in each of the languages, following exactly the same algorithm, as defined in the CLBG (Gouy, 2018). We compile/execute such programs using the state-of-the-art compilers, virtual machines, interpreters, and libraries for each of the 27 languages. Afterwards, we analyze the performance of the different implementation considering three variables: execution time, memory consumption and energy consumption. Moreover, we analyze those results according to the languages' execution type (compiled, virtual machine and interpreted), and programming paradigm (imperative, functional, object oriented, scripting) used. For each of the execution types and programming paradigms, we compiled a software language ranking according to each variable considered. Our results show interesting findings, such as, slower/faster software languages consuming less/more energy, and how memory usage influences energy consumption. Finally, we discuss how to use such results to provide software engineers support to decide which language to use when energy efficiency is a concern.

This chapter is organized as follows: Section 3.2 exposes the detailed steps of our methodology to measure and compare energy efficiency in software languages, followed by a presentation of the results. Section 3.3 contains the analysis and discussion on the obtained results, where we first analyze whether execution time performance implies energy efficiency, then we examine the relation between memory usage and memory energy consumption, and finally we present a discussion on how energy, time and memory relate in the 27 software languages. In Section 3.4 we discuss the threats to the validity of our study. Finally, in Section 3.5 we present the conclusions of our work.

## 3.2 Measuring Energy in Software Languages

The initial motivation and primary focus of this work is to understand the energy efficiency across various programming languages. This might seem like a simple task, but it is not as trivial as it sounds.

To have a fair and proper analysis, we need a good representation of different programming problems (and their solutions) written across a large set of programming languages. Additionally, the different solutions should follow the same guidelines and rules to be as fair as possible in such a comparison. It would not be enough just to get from point A to point B in a solution, they would also need to follow the same path in between while also performing as best as possible.

With this in mind, we begin by trying to answer the following research question:

- **RQ1**: *Can we compare the energy efficiency of software languages?* This will allow us to have results in which we can in fact compare the energy efficiency of popular programming languages. In having these results, we can also explore the relations between energy consumption, execution time, and memory usage.

The following subsections will detail the methodology used to answer this question, and the results we obtained.

### 3.2.1 The Computer Language Benchmarks Game

In order to obtain a comparable, representative and extensive set of programs written in many of the most popular and most widely used programming languages we have explored CLBG (Gouy, 2018).

The CLBG initiative includes a framework for running, testing and comparing implemented coherent solutions for a set of well-known, diverse programming problems. The overall motivation is to be able to compare solutions, within and between, different programming languages. While the perspectives for comparing solutions have originally essentially analyzed runtime performance, the fact is that CLBG has recently also been used in order to study the energy efficiency of software (Lima et al., 2016; Couto et al., 2017b; Oliveira et al., 2017). It is to note, that implementations submitted are written by experts in each of the languages, with the intention of being the fastest performing solution.

In its current stage, the CLBG has gathered solutions for 13 benchmark problems, such that solutions to each such problem must respect a given algorithm and specific implementation guidelines. Solutions to each problem are expressed in, at most, 28 different programming languages.

The complete list of benchmark problems in the CLBG covers different computing problems, as described in Table 3.1. Additionally, the complete list of programming languages in the CLBG is shown in Table 3.2, sorted by their paradigms. The sorting of the paradigms are based on a combination of the self definition of each language acquired from their official site and the community-defined paradigms for each language (for example on Wikipedia [1]).

### 3.2.2 Design and Execution

Although CLBG includes 28 languages, we excluded *Smalltalk* since the compiler for that language is proprietary. Also, for comparability, we have discarded benchmark problems whose language coverage is below the threshold of 80%. By language coverage we mean, for each benchmark problem, the percentage of programming languages (out of 27) in which

[1] `https://en.wikipedia.org/`

Table 3.1: CLBG corpus of programs

| Benchmark | Description | Input |
|---|---|---|
| `n-body` | Double precision N-body simulation | 50M |
| `fannkuch-redux` | Indexed access to tiny integer sequence | 12 |
| `spectral-norm` | Eigenvalue using the power method | 5,500 |
| `mandelbrot` | Generate Mandelbrot set portable bitmap file | 16,000 |
| `pidigits` | Streaming arbitrary precision arithmetic | 10,000 |
| `regex-redux` | Match DNA 8mers and substitute magic patterns | fasta output |
| `fasta` | Generate and write random DNA sequences | 25M |
| `k-nucleotide` | Hashtable update and k-nucleotide strings | fasta output |
| `reverse-complement` | Read DNA sequences, write their reverse-complement | fasta output |
| `binary-trees` | Allocate, traverse and deallocate many binary trees | 21 |
| `chameneos-redux` | Symmetrical thread rendezvous requests | 6M |
| `meteor-contest` | Search for solutions to shape packing puzzle | 2,098 |
| `thread-ring` | Switch from thread to thread passing one token | 50M |

Table 3.2: Languages sorted by paradigm

| Paradigm | Languages |
|---|---|
| Functional | Erlang, F#, Haskell, Lisp, Ocaml, Perl, Racket, Ruby, Rust; |
| Imperative | Ada, C, C++, F#, Fortran, Go, Ocaml, Pascal, Rust; |
| Object-Oriented | Ada, C++, C#, Chapel, Dart , F#, Java, JavaScript, Ocaml, Perl, PHP, Python, Racket, Rust, Smalltalk, Swift, TypeScript; |
| Scripting | Dart, Hack, JavaScript, JRuby, Lua, Perl, PHP, Python, Ruby, TypeScript; |

solutions for it are available. This criteria excluded `chameneos-redux`, `meteor-contest` and `thread-ring` from our study.

We then gathered the most efficient (i.e. fastest) version of the source code in each of the remaining 10 benchmark problems, for all the 27 considered programming languages.

The CLBG documentation also provides information about the specific compiler/runner version used for each language, as well as the compilation/execution options considered (for example, optimization flags at compile/run time). We strictly followed those instructions and installed the correct compiler versions, and also ensured that each solution was compiled/executed with the same options used in the CLBG. Once we had the correct compiler and benchmark solutions for each language, we tested each one individually to make sure that we could execute it with no errors and that the output was the expected one.

The next step was to gather the information about energy consumption, execution time and peak memory usage for each of the compilable and executable solutions in each language. It is to be noted that the CLBG already contains measured information on both the execution time and peak memory usage. We measured both not only to check the consistency of our results against the CLBG, but also since different hardware specifications would bring about different results. For measuring the energy consumption, we used Intel's Running Average Power Limit (RAPL) tool (Dimitrov et al., 2015), which is capable of providing accurate energy estimates at a very fine-grained level, as it has already been proven (Hähnel et al., 2012; Rotem et al., 2012). Also, the current version of RAPL allows it to be invoked from any program written in *C* and `Java` (through jRAPL (Liu et al., 2015)).

In order to properly compare the languages, we needed to collect the energy consumed by a single execution of a specific solution. In order to do this, we used the `system` function call in `C`, which executes the string values which are given as arguments; in our case, the command necessary to run a benchmark solution (for example, the `binary-trees` solution written in `Python` is executed by writing the command `/usr/bin/python binarytrees.py 21`).

The energy consumption of a solution will then be the energy consumed by the `system` call, which we measured using RAPL function calls. The overall process (i.e., the workflow of our energy measuring framework [2]) is described in Listing 3.1.

```
...
for (i = 0 ; i < N ; i++){
  time_before = getTime(...);
  //performs initial energy measurement
  rapl_before(...);

  //executes the program
  system(command);

  //computes the difference between
  //this measurement and the initial one
  rapl_after(...);
```

[2] The measuring framework and the complete set of results are publicly available at `https://sites.google.com/view/energy-efficiency-languages`

```
  time_elapsed = getTime(...) - time_before;
  ...
}
...
```

Listing 3.1: Overall process of the energy measuring framework.

In order to ensure that the overhead from our measuring framework, using the `system` function, is negligible or non-existing when compared to actually measuring with RAPL inside a program's source code, we design a simple experiment. It consisted of measuring the energy consumption inside of both a C and Java language solution, using RAPL and jRAPL respectively, and comparing the results to the measurements from our C language energy measuring framework. We found the resulting differences to be insignificant, and therefore negligible, thus we conclude that we could use this framework without having to worry about imprecisions in the energy measurements.

Also, we chose to measure the energy consumption and the execution time of a solution together, since the overhead will be the same for every measurement, and so this should not affect the obtained values.

The peak memory usage of a solution was gathered using the `time` tool, available in Unix-based systems. This tool runs a given program, and summarizes the system resources used by that program, which includes the peak of memory usage.

Each benchmark solution was executed and measured 10 times, in order to obtain 10 energy consumption and execution time samples. We did so to reduce the impact of cold starts and cache effects, and to be able to analyze the measurements' consistency and avoid outliers. We followed the same approach when gathering results for memory usage.

For some benchmark problems, we could not obtain any results for certain programming languages. In some cases, there was no source code available for the benchmark problem (i.e., no implementation was provided in a concrete language which reflects a language coverage below 100%).[3]

In other cases, the code was indeed provided but either the code itself was already buggy

[3] In these cases, we will include an `n.a.` indication when presenting their results.

or failing to compile or execute, as documented in CLBG, or, in spite of our best efforts, we could not execute it, e.g., due to missing libraries [3]. From now on, for each benchmark problem, we will refer as its execution coverage to the percentage of (best) solutions for it that we were actually able to successfully execute.

All studies were conducted on a desktop with the following specifications: Linux Ubuntu Server 16.10 operating system, kernel version 4.8.0-22-generic, with 16GB of RAM, a Haswell Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz.

### 3.2.3 Results

The results from our study are partially shown in this section, with the remainder shown both in the online appendix for this chapter [4], and in this document's Chapter 3 appendix (Appendix A). Tables 3.3-3.5 contains the measured data from different benchmark solutions. We only show the results for `binary-trees`, `fannkuch-redux`, and `fasta` within the chapter, which are the first 3 ordered alphabetically. Each row in a table represents one of the 27 programming languages which were measured.

The four rightmost columns, from left to right, represent the average values for the *Energy* consumed (Joules), *Time* of execution (milliseconds), *Ratio* between Energy and Time, and the amount of peak memory usage in *Mb*. The *Energy* value is the sum of CPU and DRAM energy consumption. Additionally, the *Ratio* can also be seen as the average Power, expressed in Kilowatts (kW). The rows are ordered according to the programming language's energy consumption, from lowest to highest. Finally, the online appendix contains the standard deviation and average values for our measured CPU, DRAM, and Time, allowing us to understand the variance.

The first column states the name of the programming languages, preceded by either a (c), (i), or (v) classifying them as either a compiled, interpreted, or virtual-machine language, respectively. In some cases, the programming language name will be followed with a $\uparrow_x/\downarrow_y$ and/or $\Uparrow_x/\Downarrow_y$ symbol. The first set of arrows indicates that the language would go up by x

[4] Chapter 3 Online Appendix:
`https://sites.google.com/view/energy-efficiency-languages/results`

Table 3.3: Results for binary-trees

| **binary-trees** | | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) C | 39.80 | 1125 | 0.035 | 131 |
| (c) C++ | 41.23 | 1129 | 0.037 | 132 |
| (c) Rust $\Downarrow_{2}$ | 49.07 | 1263 | 0.039 | 180 |
| (c) Fortran $\Uparrow_{1}$ | 69.82 | 2112 | 0.033 | 133 |
| (c) Ada $\Downarrow_{1}$ | 95.02 | 2822 | 0.034 | 197 |
| (c) Ocaml $\downharpoonleft_{1}$ $\Uparrow_{2}$ | 100.74 | 3525 | 0.029 | 148 |
| (v) Java $\upharpoonleft_{1}$ $\Downarrow_{16}$ | 111.84 | 3306 | 0.034 | 1120 |
| (v) Lisp $\downharpoonleft_{3}$ $\Downarrow_{3}$ | 149.55 | 10570 | 0.014 | 373 |
| (v) Racket $\downharpoonleft_{4}$ $\Downarrow_{6}$ | 155.81 | 11261 | 0.014 | 467 |
| (i) Hack $\upharpoonleft_{2}$ $\Downarrow_{9}$ | 156.71 | 4497 | 0.035 | 502 |
| (v) C# $\downharpoonleft_{1}$ $\Downarrow_{1}$ | 189.74 | 10797 | 0.018 | 427 |
| (v) F# $\downharpoonleft_{3}$ $\Downarrow_{1}$ | 207.13 | 15637 | 0.013 | 432 |
| (c) Pascal $\downharpoonleft_{3}$ $\Uparrow_{5}$ | 214.64 | 16079 | 0.013 | 256 |
| (c) Chapel $\upharpoonleft_{5}$ $\Uparrow_{4}$ | 237.29 | 7265 | 0.033 | 335 |
| (v) Erlang $\upharpoonleft_{5}$ $\Uparrow_{1}$ | 266.14 | 7327 | 0.036 | 433 |
| (c) Haskell $\upharpoonleft_{2}$ $\Downarrow_{2}$ | 270.15 | 11582 | 0.023 | 494 |
| (i) Dart $\downharpoonleft_{1}$ $\Uparrow_{1}$ | 290.27 | 17197 | 0.017 | 475 |
| (i) JavaScript $\downharpoonleft_{2}$ $\Downarrow_{4}$ | 312.14 | 21349 | 0.015 | 916 |
| (i) TypeScript $\downharpoonleft_{2}$ $\Downarrow_{2}$ | 315.10 | 21686 | 0.015 | 915 |
| (c) Go $\upharpoonleft_{3}$ $\Uparrow_{13}$ | 636.71 | 16292 | 0.039 | 228 |
| (i) Jruby $\upharpoonleft_{2}$ $\Downarrow_{3}$ | 720.53 | 19276 | 0.037 | 1671 |
| (i) Ruby $\Uparrow_{5}$ | 855.12 | 26634 | 0.032 | 482 |
| (i) PHP $\Uparrow_{3}$ | 1,397.51 | 42316 | 0.033 | 786 |
| (i) Python $\Uparrow_{15}$ | 1,793.46 | 45003 | 0.040 | 275 |
| (i) Lua $\downharpoonleft_{1}$ | 2,452.04 | 209217 | 0.012 | 1961 |
| (i) Perl $\upharpoonleft_{1}$ | 3,542.20 | 96097 | 0.037 | 2148 |
| (c) Swift | n.e. | | | |

positions ($\upharpoonleft_x$) or down by y positions ($\downharpoonleft_y$) if ordered by *execution time*. For example in Table 3.5, for the `fasta` benchmark, `Fortran` is the second most energy efficient language, but falls off 6 positions down if ordered by execution time. The second set of arrows states that the language would go up by x positions ($\Uparrow_x$) or down by y positions ($\Downarrow_y$) if ordered according to their *peak memory usage*. Looking at the same example benchmark, `Rust`, while the most energy efficient, would drop 9 positions if ordered by peak memory usage.

Table 3.6 shows the global results (on average) for *Energy*, *Time*, and *Mb* normalized to the most efficient language in that category. Since the `pidigits` benchmark solutions only contained less than half of the languages covered, we did not consider this one for the global results. The base values are as follows: *Energy* for `C` is 57.86J, *Time* for `C` is 2019.26ms, and *Mb* for `Pascal` is 65.96Mb. For instance, `Lisp`, on average, consumes 2.27x more energy (131.34J) than `C`, while taking 2.44x more time to execute (4926.99ms), and 1.92x more memory (126.64Mb) needed when compared to `Pascal`.

Table 3.4: Results for fannkuch-redux

| fannkuch-redux | | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) C $\Downarrow_{2}$ | 215.92 | 6076 | 0.036 | 2 |
| (c) C++ $\Uparrow_{1}$ | 219.89 | 6123 | 0.036 | 1 |
| (c) Rust $\Downarrow_{11}$ | 238.30 | 6628 | 0.036 | 16 |
| (c) Swift $\Downarrow_{5}$ | 243.81 | 6712 | 0.036 | 7 |
| (c) Ada $\Downarrow_{2}$ | 264.98 | 7351 | 0.036 | 4 |
| (c) Ocaml $\downarrow_{1}$ | 277.27 | 7895 | 0.035 | 3 |
| (c) Chapel $\uparrow_{1}$ $\Downarrow_{18}$ | 285.39 | 7853 | 0.036 | 53 |
| (v) Lisp $\downarrow_{3}$ $\Downarrow_{15}$ | 309.02 | 9154 | 0.034 | 43 |
| (v) Java $\uparrow_{1}$ $\Downarrow_{13}$ | 311.38 | 8241 | 0.038 | 35 |
| (c) Fortran $\Downarrow_{1}$ | 316.50 | 8665 | 0.037 | 12 |
| (c) Go $\uparrow_{2}$ $\Uparrow_{7}$ | 318.51 | 8487 | 0.038 | 2 |
| (c) Pascal $\Uparrow_{10}$ | 343.55 | 9807 | 0.035 | 2 |
| (v) F# $\downarrow_{1}$ $\Downarrow_{7}$ | 395.03 | 10950 | 0.036 | 34 |
| (v) C# $\uparrow_{1}$ $\Downarrow_{5}$ | 399.33 | 10840 | 0.037 | 29 |
| (i) JavaScript $\downarrow_{1}$ $\Downarrow_{2}$ | 413.90 | 33663 | 0.012 | 26 |
| (c) Haskell $\uparrow_{1}$ $\Uparrow_{8}$ | 433.68 | 14666 | 0.030 | 7 |
| (i) Dart $\Downarrow_{7}$ | 487.29 | 38678 | 0.013 | 46 |
| (v) Racket $\Uparrow_{3}$ | 1,941.53 | 43680 | 0.044 | 18 |
| (v) Erlang $\Uparrow_{3}$ | 4,148.38 | 101839 | 0.041 | 18 |
| (i) Hack $\Downarrow_{6}$ | 5,286.77 | 115490 | 0.046 | 119 |
| (i) PHP | 5,731.88 | 125975 | 0.046 | 34 |
| (i) TypeScript $\downarrow_{4}$ $\Uparrow_{4}$ | 6,898.48 | 516541 | 0.013 | 26 |
| (i) Jruby $\uparrow_{1}$ $\Downarrow_{4}$ | 7,819.03 | 219148 | 0.036 | 669 |
| (i) Lua $\downarrow_{3}$ $\Uparrow_{19}$ | 8,277.87 | 635023 | 0.013 | 2 |
| (i) Perl $\uparrow_{2}$ $\Uparrow_{12}$ | 11,133.49 | 249418 | 0.045 | 12 |
| (i) Python $\uparrow_{2}$ $\Uparrow_{14}$ | 12,784.09 | 279544 | 0.046 | 12 |
| (i) Ruby $\uparrow_{2}$ $\Uparrow_{17}$ | 14,064.98 | 315583 | 0.045 | 8 |

To better visualize and interpret the data, we also generated two different sets of graphical data for each of the benchmarks. The first set, Figures 3.1-3.3 contains the results of each language for a benchmark, consisting of three joint parts: a bar chart, a line chart, and a scatter plot. The bars represent the energy consumed by the languages, with the CPU energy consumption on the bottom half in blue dotted bars and DRAM energy consumption on the top half in orange solid bars, and the left y-axis representing the average Joules. The execution time is represented by the line chart, with the right y-axis representing average time in milliseconds. The joining of these two charts allow us to better understand the relationship between energy and time. Finally, a scatter plot on top of both represents the ratio between energy consumed and execution time. The ratio plot allows us to understand if the relationship between energy and time is consistent across languages. A variation in these values indicates that energy consumed is not directly proportional to time, but dependent on the language and/or benchmark solution.

Table 3.5: Results for fasta

| **fasta** | | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) Rust $\Downarrow_{9}$ | 26.15 | 931 | 0.028 | 16 |
| (c) Fortran $\downarrow_{6}$ | 27.62 | 1661 | 0.017 | 1 |
| (c) C $\uparrow_{1}$ $\Downarrow_{1}$ | 27.64 | 973 | 0.028 | 3 |
| (c) C++ $\uparrow_{1}$ $\Downarrow_{2}$ | 34.88 | 1164 | 0.030 | 4 |
| (v) Java $\uparrow_{1}$ $\Downarrow_{12}$ | 35.86 | 1249 | 0.029 | 41 |
| (c) Swift $\Downarrow_{9}$ | 37.06 | 1405 | 0.026 | 31 |
| (c) Go $\downarrow_{2}$ | 40.45 | 1838 | 0.022 | 4 |
| (c) Ada $\downarrow_{2}$ $\Uparrow_{3}$ | 40.45 | 2765 | 0.015 | 3 |
| (c) Ocaml $\downarrow_{2}$ $\Downarrow_{15}$ | 40.78 | 3171 | 0.013 | 201 |
| (c) Chapel $\uparrow_{5}$ $\Downarrow_{10}$ | 40.88 | 1379 | 0.030 | 53 |
| (v) C# $\uparrow_{4}$ $\Downarrow_{5}$ | 45.35 | 1549 | 0.029 | 35 |
| (i) Dart $\Downarrow_{6}$ | 63.61 | 4787 | 0.013 | 49 |
| (i) JavaScript $\Downarrow_{1}$ | 64.84 | 5098 | 0.013 | 30 |
| (c) Pascal $\downarrow_{1}$ $\Uparrow_{13}$ | 68.63 | 5478 | 0.013 | 0 |
| (i) TypeScript $\downarrow_{2}$ $\Downarrow_{10}$ | 82.72 | 6909 | 0.012 | 271 |
| (v) F# $\uparrow_{2}$ $\Uparrow_{3}$ | 93.11 | 5360 | 0.017 | 27 |
| (v) Racket $\downarrow_{1}$ $\Uparrow_{5}$ | 120.90 | 8255 | 0.015 | 21 |
| (c) Haskell $\uparrow_{2}$ $\Downarrow_{8}$ | 205.52 | 5728 | 0.036 | 446 |
| (v) Lisp $\Downarrow_{2}$ | 231.49 | 15763 | 0.015 | 75 |
| (i) Hack $\Downarrow_{3}$ | 237.70 | 17203 | 0.014 | 120 |
| (i) Lua $\Uparrow_{18}$ | 347.37 | 24617 | 0.014 | 3 |
| (i) PHP $\downarrow_{1}$ $\Uparrow_{13}$ | 430.73 | 29508 | 0.015 | 14 |
| (v) Erlang $\uparrow_{1}$ $\Uparrow_{12}$ | 477.81 | 27852 | 0.017 | 18 |
| (i) Ruby $\downarrow_{1}$ $\Uparrow_{2}$ | 852.30 | 61216 | 0.014 | 104 |
| (i) JRuby $\uparrow_{1}$ $\Downarrow_{2}$ | 912.93 | 49509 | 0.018 | 705 |
| (i) Python $\downarrow_{1}$ $\Uparrow_{18}$ | 1,061.41 | 74111 | 0.014 | 9 |
| (i) Perl $\uparrow_{1}$ $\Uparrow_{8}$ | 2,684.33 | 61463 | 0.044 | 53 |

The second set, Figures 3.4-3.6 consists of two parts: a bar chart, and a line chart. The blue bars represent the DRAM's energy consumption for each of the languages, with the left y-axis representing the average Joules. The orange line chart represents the peak memory usage for each language, with the right y-axis representing the average Mb. The joining of these two allows us to look at the relation between DRAM energy consumption and the peak memory usage for each language in each benchmark.

By turning to the CLBG, we were able to use a large set of software programming languages which solve various different programming problems with similar solutions. This allowed us to obtain a comparable, representative, and extensive set of programs, written in several of the most popular languages, along with the compilation/execution options, and compiler versions. With these joined together with our energy measurement framework, which uses the accurate Intel RAPL tool, we were able to measure, analyze, and compare the energy consumption, and in turn the energy efficiency, of software languages, thus answer-

Table 3.6: Normalized global results for Energy, Time, and Memory

| Total | | | | | |
| --- | --- | --- | --- | --- | --- |
| | Energy | | Time | | Mb |
| (c) C | 0.800 | (c) C | 0.800 | (c) Pascal | 0.800 |
| (c) Rust | 0.803 | (c) Rust | 0.804 | (c) Go | 0.805 |
| (c) C++ | 1.34 | (c) C++ | 1.56 | (c) C | 1.17 |
| (c) Ada | 1.70 | (c) Ada | 1.85 | (c) Fortran | 1.24 |
| (v) Java | 1.98 | (v) Java | 1.89 | (c) C++ | 1.34 |
| (c) Pascal | 2.14 | (c) Chapel | 2.14 | (c) Ada | 1.47 |
| (c) Chapel | 2.18 | (c) Go | 2.83 | (c) Rust | 1.54 |
| (v) Lisp | 2.27 | (c) Pascal | 3.02 | (v) Lisp | 1.92 |
| (c) Ocaml | 2.40 | (c) Ocaml | 3.09 | (c) Haskell | 2.45 |
| (c) Fortran | 2.52 | (v) C# | 3.14 | (i) PHP | 2.57 |
| (c) Swift | 2.79 | (v) Lisp | 3.40 | (c) Swift | 2.71 |
| (c) Haskell | 3.10 | (c) Haskell | 3.55 | (i) Python | 2.80 |
| (v) C# | 3.14 | (c) Swift | 4.20 | (c) Ocaml | 2.82 |
| (c) Go | 3.23 | (c) Fortran | 4.20 | (v) C# | 2.85 |
| (i) Dart | 3.83 | (v) F# | 6.30 | (i) Hack | 3.34 |
| (v) F# | 4.13 | (i) JavaScript | 6.52 | (v) Racket | 3.52 |
| (i) JavaScript | 4.45 | (i) Dart | 6.67 | (i) Ruby | 3.97 |
| (v) Racket | 7.91 | (v) Racket | 11.27 | (c) Chapel | 4.00 |
| (i) TypeScript | 21.50 | (i) Hack | 26.99 | (v) F# | 4.25 |
| (i) Hack | 24.02 | (i) PHP | 27.64 | (i) JavaScript | 4.59 |
| (i) PHP | 29.30 | (v) Erlang | 36.71 | (i) TypeScript | 4.69 |
| (v) Erlang | 42.23 | (i) Jruby | 43.44 | (v) Java | 6.01 |
| (i) Lua | 45.98 | (i) TypeScript | 46.20 | (i) Perl | 6.62 |
| (i) Jruby | 46.54 | (i) Ruby | 59.34 | (i) Lua | 6.72 |
| (i) Ruby | 69.91 | (i) Perl | 65.79 | (v) Erlang | 7.20 |
| (i) Python | 75.88 | (i) Python | 71.90 | (i) Dart | 8.64 |
| (i) Perl | 79.58 | (i) Lua | 82.91 | (i) Jruby | 19.84 |

ing **RQ1** as shown with our results. Additionally, we were also able to measure the execution time and peak memory usage which allowed us to analyze how these two relate with energy consumption. The analysis and discussion of our results is shown in the next section.

## 3.3 Analysis and Discussion

In this section we will present an analysis and discussion on the results of our study. While our main focus is on understanding the energy efficiency in languages, we will also try to understand how energy, time, and memory relate. Additionally, in this section we will try to answer the following three research questions, each with their own designated subsection.

- **RQ2**: *Is the faster language always the most energy efficient?* Properly understanding this will not only address if energy efficiency is purely a performance problem, but also allow developers to have a greater understanding of how energy and time relates in a

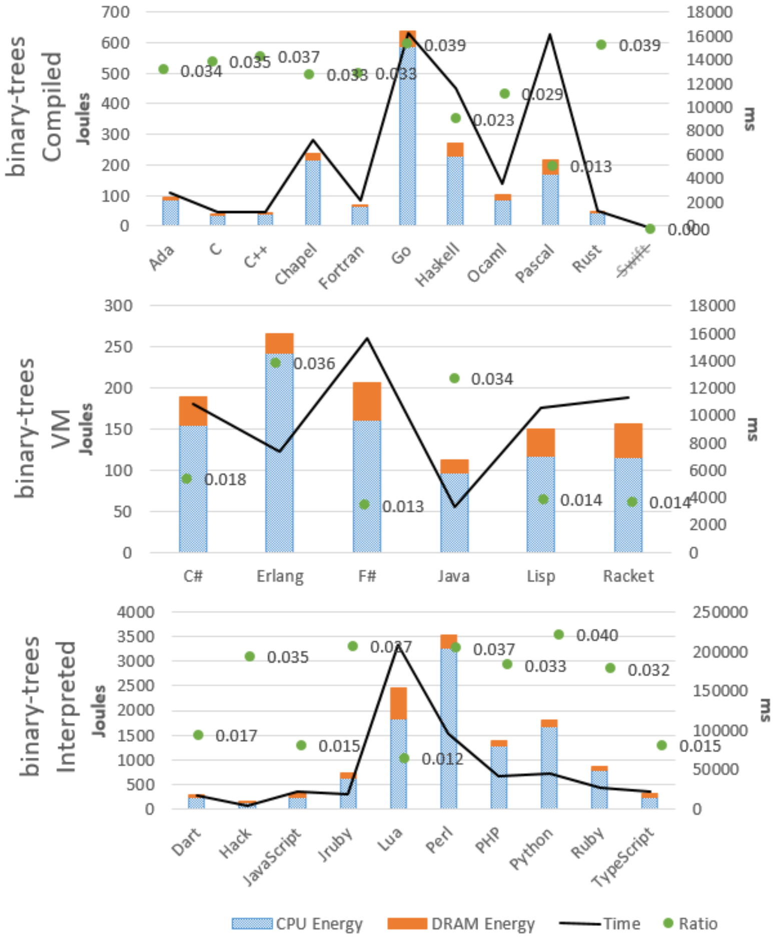


Figure 3.1: Energy and time graphical data for binary-trees

language, and between languages.

- **RQ3**: *How does memory usage relate to energy consumption?* Insight on how memory usage affects energy consumption will allow developers to better understand how to manage memory if their concern is energy consumption.

- **RQ4**: *Can we automatically decide what is the best programming language considering energy, time, and memory usage?* Often times developers are concerned with more than one (possibly limited) resource. For example, both energy and time, time and memory space, energy and memory space or all three. Analyzing these trade-offs will allow developers to know which programming languages are best in a given scenarios.

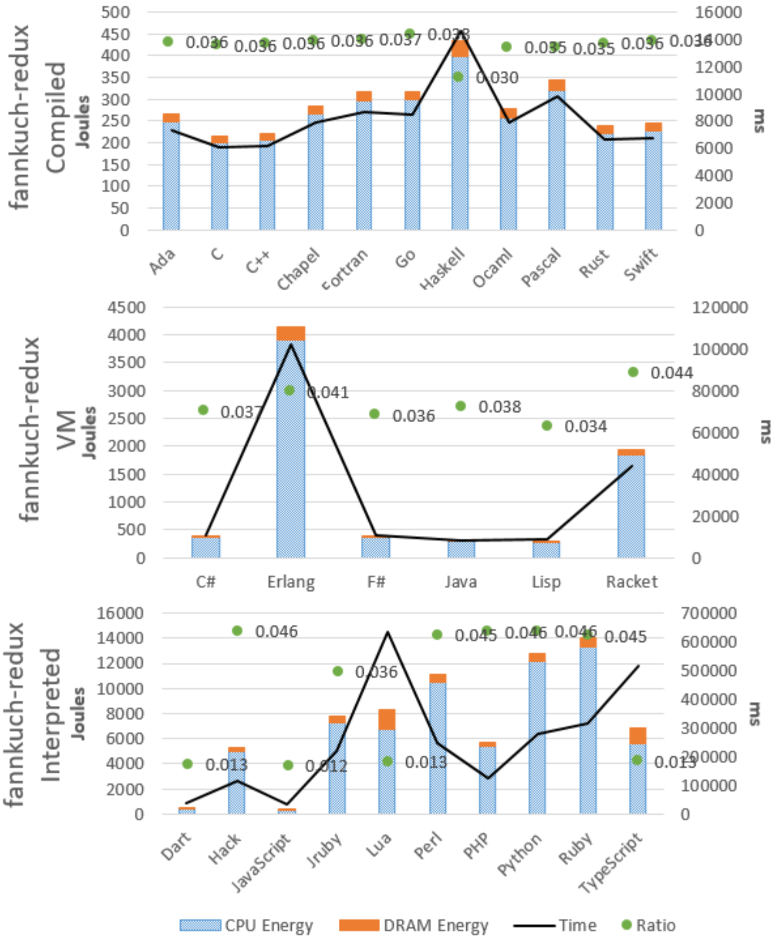


Figure 3.2: Energy and time graphical data for fannkuch-redux

### 3.3.1 Is Faster, Greener?

A very common misconception when analyzing energy consumption in software is that it will behave in the same way execution time does. In other words, reducing the execution time of a program would bring about the same amount of energy reduction. In fact, the `Energy` equation 3.1, indicates that reducing time implies a reduction in the energy consumed. However, the `Power` variable of the equation, which cannot be assumed as a constant, also has an impact on the energy. Therefore, conclusions regarding this issue diverge sometimes, where some works do support that energy and time are directly related (Yuki and Rajopadhye, 2014), and the opposite was also observed (Lima et al., 2016; Pinto et al., 2014b; Trefethen and Thiya-

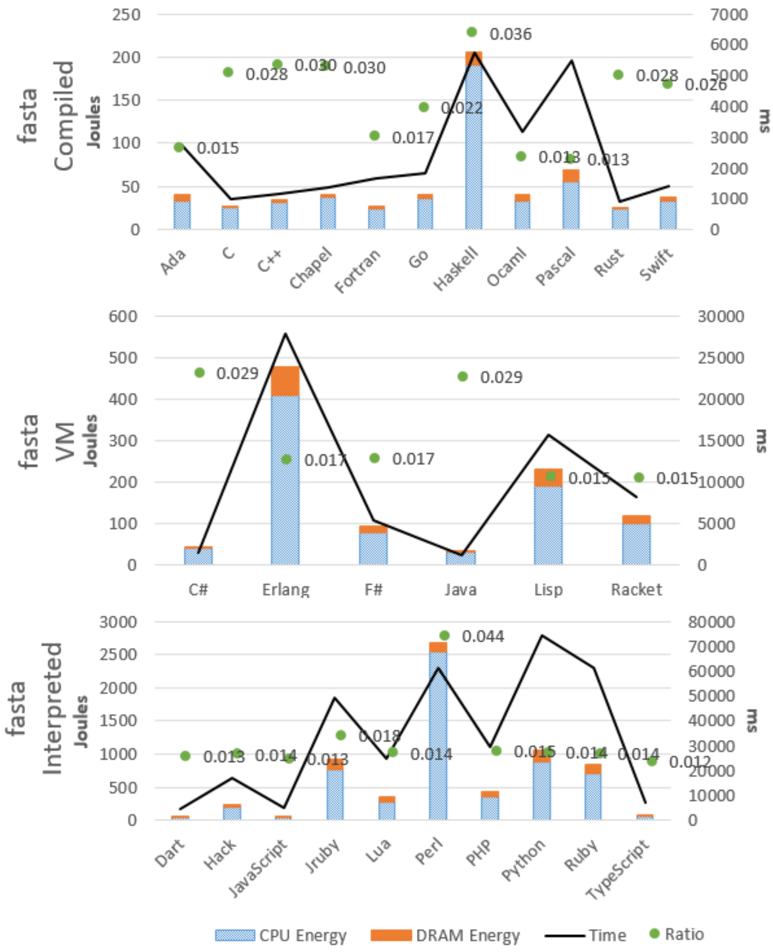


Figure 3.3: Energy and time graphical data for fasta

galingam, 2013a).

The data presented in the aforementioned tables and figures lets us draw an interesting set of observations regarding the efficiency of software languages when considering both energy consumption and execution time. Much like (Abdulsalam et al., 2015) and (Pereira et al., 2016), we observed different behaviors for energy consumption and execution time in different languages and tests.

By observing the data in Table 3.6, we can see that the C language is, overall, the fastest and most energy efficient. Nevertheless, in some specific benchmarks there are more efficient solutions (for example, in the `fasta` benchmark it is the third most energy efficient and

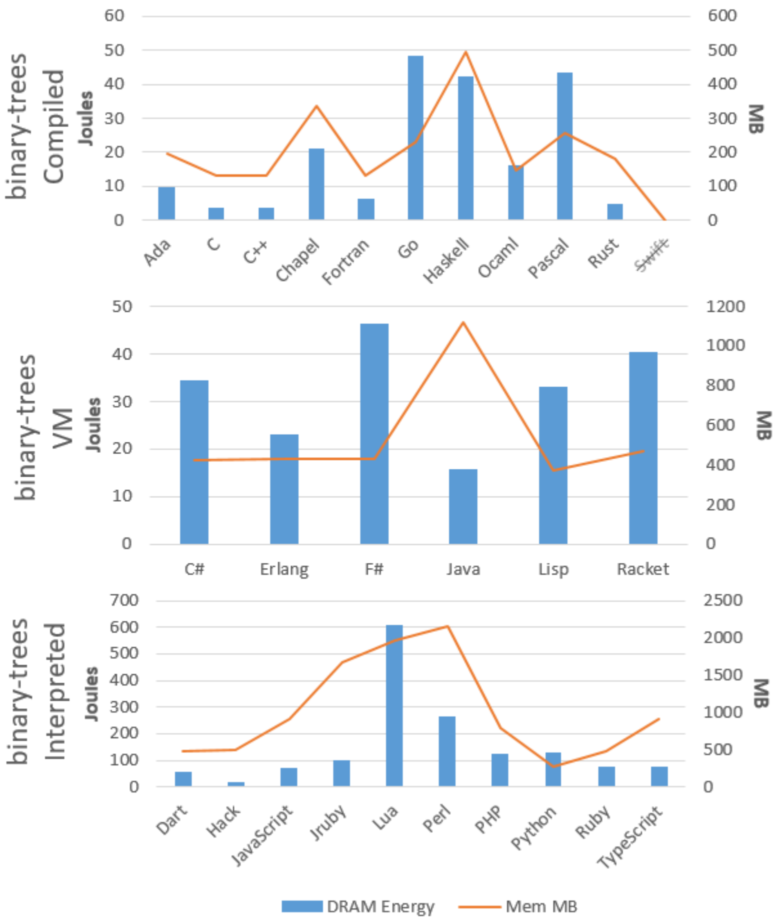


Figure 3.4: Energy and memory graphical data for binary-trees

second fastest).

Execution time behaves differently when compared to energy efficiency. The results for the 3 benchmarks presented in Tables 3.3-3.5 (and the remainder shown in the appendix under Appendix A.2 *Data tables*) show several scenarios where a certain language energy consumption rank differs from the execution time rank (as the arrows in the first column indicate). In the `fasta` benchmark, for example, the `Fortran` language is second most energy efficient, while dropping 6 positions when it comes to execution time. Moreover, by observing the *Ratio* values in Figures 3.1 to 3.3 (and the remainder in the appendices under under Appendix A.3 *Energy and Time Graphs*), we clearly see a substantial variation between languages. This means that the average power is not constant, which further strengthens the

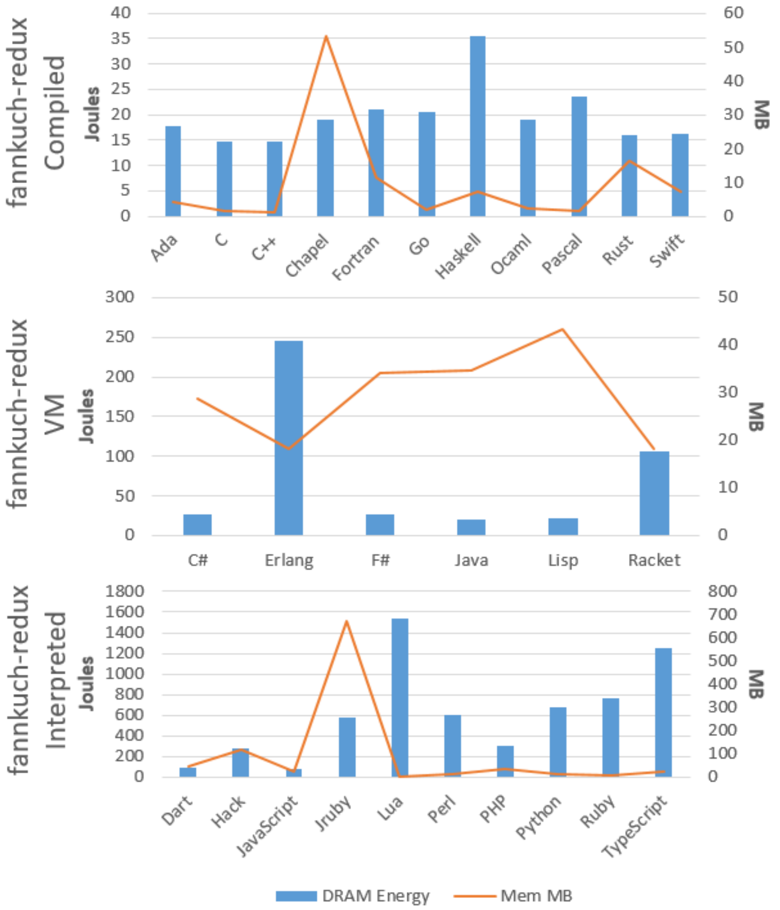


Figure 3.5: Energy and memory graphical data for fannkuch-redux

previous point. With this variation, we can have languages with very similar energy consumptions and completely different execution times, as is the case of languages `Pascal` and `Chapel` in the `binary trees` benchmark, which energy consumption differ roughly by 10% in favor of `Pascal`, while `Chapel` takes about 55% less time to execute.

Compiled languages tend to be, as expected, the fastest and most energy efficient ones. On average, compiled languages consumed 120J to execute the solutions, while for virtual machine and interpreted languages this value was 576J and 2365J, respectively. This tendency can also be observed for execution time, since compiled languages took 5103ms, virtual machine languages took 20623ms, and interpreted languages took 87614ms (on average). Grouped by the different paradigms, the imperative languages consumed and took on aver-

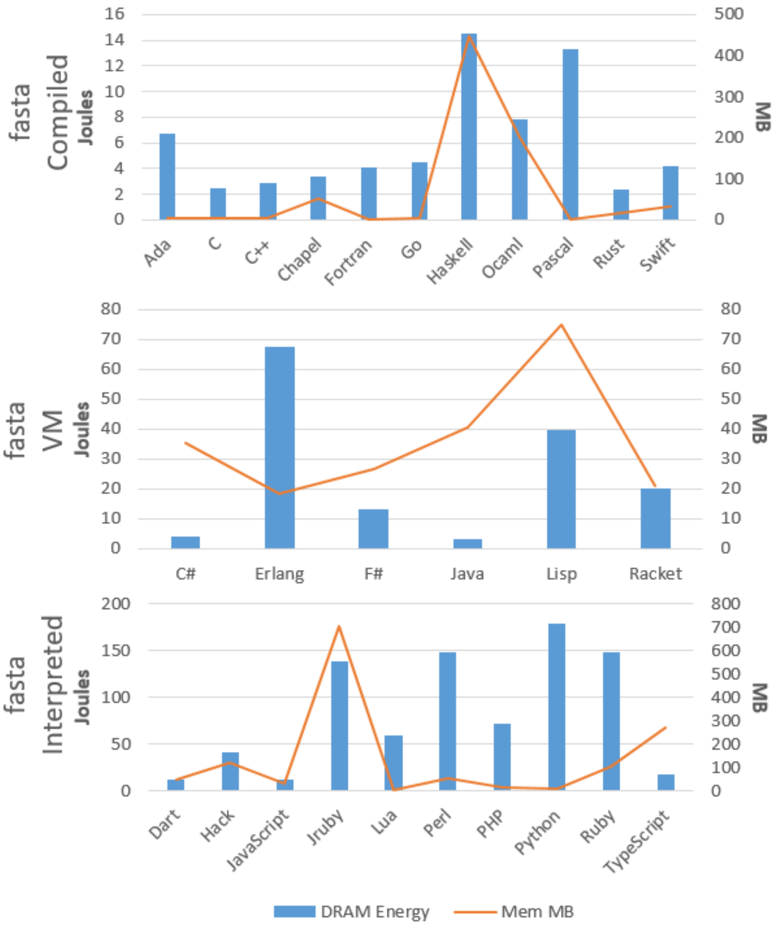


Figure 3.6: Energy and memory graphical data for fasta

age 125J and 5585ms, the object-oriented consumed 879J and spent 32965ms, the functional consumed 1367J and spent 42740ms and the scripting languages consumed 2320J and spent 88322ms.

Moreover, the top 5 languages that need less energy and time to execute the solutions are: `C` (57J, 2019ms), `Rust` (59J, 2103ms), `C++` (77J, 3155ms), `Ada` (98J, 3740ms), and `Java` (114J, 3821ms); of these, only `Java` is not compiled. As expected, the bottom 5 languages are all interpreted: `Perl` (4604J), `Python` (4390J), `Ruby` (4045J), `JRuby` (2693J), and `Lua` (2660Js) for energy; `Lua` (167416ms), `Python` (145178ms), `Perl` (132856ms), `Ruby` (119832ms), and `TypeScript` (93292ms) for time.

The CPU-based energy consumption always represents the majority of the energy consumed. On average, for the compiled languages, this value represents 88.94% of the energy consumed, being the remaining portion assigned to DRAM. This value is very similar for virtual machine (88.94%) and interpreted languages (87.98%). While, as explained in the last point, the overall average consumption for these 3 language types is very different, the ratio between CPU and DRAM based energy consumption seems to generally maintain the same proportion. This might indicate that optimizing a program to reduce the CPU-based energy consumption will also decrease the DRAM-based energy consumption. However, it is interesting to notice that this value varies more for interpreted languages (min of 81.57%, max of 92.90%) when compared to compiled (min of 85.27%, max of 91.75%) or virtual machine languages (min of 86.10%, max of 92.43%).

With these results, we can try to answer the question raised in **RQ2**: *Is the faster language always the most energy efficient?* By looking solely at the overall results, shown in Table 3.6, we can see that the top 5 most energy efficient languages keep their rank when they are sorted by execution time and with very small differences in both energy and time values. This does not come as a surprise, since in 9 out of 10 benchmark problems, the fastest and most energy efficient programming language was one of the top 3. Additionally, it is common knowledge that these top 3 language (`C`,`C++`, and `Rust`) are known to be heavily optimized and efficient for execution performance, as our data also shows. Thus, as time influences energy, we had hypothesized that these languages would also produce efficient energy consumptions as they have a large advantage in one of the variables influencing energy, even if they consumed more power on average.

Nevertheless, if we look at the remaining languages in Table 3.6, we can see that only 4 languages maintain the same energy and time rank (`OCaml`, `Haskel`, `Racket`, and `Python`), while the remainder are completely shuffled. Additionally, looking at individual benchmarks we see many cases where there is a different order for energy and time.

Moreover, the tables in Appendix A.2 *Data Tables* also allows us to understand that this question does not have a concrete and ultimate answer. Although the most energy efficient language in each benchmark is almost always the fastest one, the fact is that there is no language which is consistently better than the others. This allows us to conclude that the situation on which a language is going to be used is a core aspect to determine if that language is the most energy efficient option. For example, in the regex-redux benchmark, which manipulates strings using regular expressions, interpreted languages seem to be an energy efficient choice (TypeScript, JavaScript and PHP, all interpreted, are in the top 5), although they tend to be not very energy efficient in other scenarios. Thus, the answer for **RQ2** is: No, a faster language is **not always** the most energy efficient.

### 3.3.2 Memory Impact on Energy

How does memory usage affect the memory's energy consumption? There are two main possible scenarios which may influence this energy consumption: continuous memory usage and peak memory usage. With the data we have collected, we will try to answer the latter scenario.

The top 5 languages, also presented in Table 3.6, which needed the least amount of memory space (on average) to execute the solutions were: Pascal (66Mb), Go (69Mb), C (77Mb), Fortran (82Mb), and C++ (88Mb); these are all compiled languages. The bottom 5 languages were: JRuby (1309Mb), Dart (570Mb), Erlang (475Mb), Lua (444Mb), and Perl (437Mb); of these, only Erlang is not an interpreted language.

On average, the compiled languages needed 125Mb, the virtual machine languages needed 285Mb, and the interpreted needed 426Mb. If sorted by their programming paradigm, the imperative languages needed 116Mb, the object-oriented 249Mb, the functional 251Mb, and finally the scripting needed 421Mb.

Additionally, the top 5 languages which consumed the least amount of DRAM energy (average) were: C (5J), Rust (6J), C++ (8J), Ada (10J), and Java (11J); of these, only Java is not a compiled language. The bottom 5 languages were: Lua (430J), JRuby (383J), Python (356J), Perl (327J), and Ruby (295J); all are interpreted languages. On average, the compiled languages consumed 14J, the virtual machine languages consumed 52J, and the interpreted languages consumed 236J.

Looking at the visual data from Figures 3.4-3.6, and the right most figures under Appendix A.4 *Energy and Memory Graphs* in the appendix, one can quickly see that there does not seem to be a consistent correlation between the DRAM energy consumption and the peak memory usage. To verify this, we first tested both the DRAM energy consumption and peak memory usage for normality using the Shapiro-Wilk (Shapiro and Wilk, 1965) test. As the data is not normally distributed, we calculated the Spearman (Zwillinger and Kokoska, 1999) rank-order correlation coefficient. The result was a Spearman $\rho$ value equal to 0.2091, meaning it is between no linear relationship ($\rho = 0$) and a weak uphill positive relationship ($\rho = 0.3$).

While we did expect the possibility of little correlation between the DRAM's energy consumption and peak memory usage, we were surprised that the relationship is almost non-existent. Thus, answering the first part of **RQ3**, this indicates that the DRAM's energy consumption has very little to do with how much memory is saved at a given point, but possibly more of how it is used.

### Continuous Memory Usage

As there was no apparent relation between DRAM's energy consumption and peak memory usage, we decided to turn our attentions towards a different approach on memory usage: continuous memory usage. Thus, a new research question was asked, extending from our previous **RQ3**.

- **RQ3.5**: *How does total memory usage relate to energy consumption?* We looked at how peak memory usage has almost no statistical effect on the DRAM's energy consumption. Thus, approaching an alternative side of memory usage, in this case total memory usage over the program's execution, can help us better understand this relationship.

The experiment methodology was the same as the one performed for peak memory usage analysis. For each language, we executed every solution while keeping track of the total amount of memory used. We used the Python `memory_profiler` [5] library to obtain the values, and afterwards we calculated, for each language, the average of all solutions. Table 3.7 summarizes the results of the experiment, by showing the relation between DRAM energy consumption and total memory usage.

[5] Python memory profiler page: `https://pypi.org/project/memory_profiler/`.

Table 3.7: Results for DRAM Energy Consumption and Total Memory

| | DRAM Joules | Peak MB | Total MB |
|---|---|---|---|
| (c) C | 5.28 | 77 | 626 |
| (c) Rust | 5.70 | 102 | 1087 |
| (c) C++ | 8.54 | 88 | 2274 |
| (c) Ada | 10.00 | 97 | 3020 |
| (c) Pascal | 15.24 | 66 | 3046 |
| (v) Erlang | 205.36 | 475 | 5457 |
| (c) Go | 15.49 | 69 | 5797 |
| (v) Lisp | 23.84 | 127 | 7544 |
| (c) Haskell | 22.40 | 162 | 8126 |
| (c) Chapel | 12.37 | 264 | 10513 |
| (c) Fortran | 24.16 | 82 | 10715 |
| (v) Java | 12.89 | 397 | 13935 |
| (v) C# | 18.62 | 188 | 14351 |
| (c) Swift | 25.72 | 179 | 23102 |
| (v) F# | 35.28 | 280 | 30218 |
| (i) Dart | 36.24 | 570 | 33891 |
| (c) OCaml | 19.62 | 186 | 36839 |
| (v) Racket | 63.29 | 232 | 38921 |
| (i) TypeScript | 272.30 | 309 | 52967 |
| (i) JavaScript | 42.70 | 303 | 88831 |
| (i) Python | 358.75 | 185 | 116265 |
| (i) PHP | 155.13 | 169 | 188136 |
| (i) Hack | 133.88 | 221 | 194589 |
| (i) Ruby | 353.00 | 262 | 203864 |
| (i) Perl | 326.82 | 437 | 255738 |
| (i) Lua | 487.50 | 444 | 690087 |
| (i) JRuby | 383.85 | 1309 | 890144 |

The average values presented in the table, and most importantly the order in which the languages appear, gives as a clear first impression that the DRAM's energy consumption relates differently with peak memory usage and continuous memory usage. In the previous section, we saw that the top 5 languages with lowest peak memory usage were `Pascal`, `Go`, `C`, `Fortran`, and `C++`. For continuous memory usage, the top 5 less consuming languages are `C` (626 Mb), `Rust` (1,087 Mb), `C++` (2,274 Mb), `Ada` (3,020 Mb), and `Pascal` (3,046 Mb). In fact, almost every other language switches places from one ranking to another.

In order to test if there is a correlation between DRAM energy consumption and continuous memory usage, we repeated the statistical test performed for peak memory usage. Once again, the Shapiro-Wilk test revealed the values were not normally distributed, thus we calculated the Spearman correlation coefficient, which resulted in a $\rho$ value of 0.744, indicating a strong positive relationship. Thus, answering **RQ3.5**, we now know that there is a strong uphill relationship between total memory usage and DRAM energy consumption. The most memory used over a program's lifecyle, the more DRAM energy consumption is spent.

There seems to be in fact a clear relation between the DRAM energy and total memory used, where a lower memory usage value leads to less energy consumed. Since the opposite was observed for peak memory usage (i.e., almost no relation with DRAM energy), these results seem to indicate that, it might be more energy efficient to store high amounts of memory at once and releasing it right afterwards than continuous memory usage throughout the execution.

### 3.3.3 Energy vs. Time vs. Memory

There are many situations where a software engineer has to choose a particular software language to implement his algorithm according to functional or non functional requirements. For instance, if he is developing software for wearables, it is important to choose a language and apply energy-aware techniques to help save battery. Another example is the implementation of tasks that run in background. In this case, execution time may not be a main concern, and they may take longer than the ones related to the user interaction.

With the fourth research question **RQ4**, we try to understand if it is possible to automatically decide what is the best programming language when considering energy consumption, execution time, and peak memory usage needed by their programs, globally and individually. In other words, if there is a "best" programming languages for all three characteristics, or if not, which are the best in each given scenario.

To this end, we present in Table 3.8 a comparison of three language characteristics: energy consumption, execution time, and peak memory usage. In order to compare the languages using more than one characteristic at a time we use a multi-objective optimization algorithm to sort these languages, known as Pareto optimization (Deb et al., 2005, 2002). It is necessary to use such an algorithm because in some cases it may happen that no solution simultaneously optimizes all objectives. For our example, energy, time, and peak memory are the optimization objectives. In these cases, a dominant solution does not exist, but each solution is a set, in our case, of software languages. Here, the solution is called the Pareto optimal.

We used this technique, and in particular the software available at (Woodruff and Her-

Table 3.8: Pareto optimal sets for different combination of objectives.

| **Time & Memory** | **Energy & Time** |
|---|---|
| C • Pascal • Go | C |
| Rust • C++ • Fortran | Rust |
| Ada | C++ |
| Java • Chapel • Lisp • Ocaml | Ada |
| Haskell • C# | Java |
| Swift • PHP | Pascal • Chapel |
| F# • Racket • Hack • Python | Lisp • Ocaml • Go |
| JavaScript • Ruby | Fortran • Haskell • C# |
| Dart • TypeScript • Erlang | Swift |
| JRuby • Perl | Dart • F# |
| Lua | JavaScript |
| | Racket |
| | TypeScript • Hack |
| | PHP |
| | Erlang |
| | Lua • JRuby |
| | Ruby |
| | Perl • Python |
| **Energy & Memory** | **Energy & Time & Memory** |
| C • Pascal | C • Pascal • Go |
| Rust • C++ • Fortran • Go | Rust • C++ • Fortran |
| Ada | Ada |
| Java • Chapel • Lisp | Java • Chapel • Lisp • Ocaml |
| OCaml • Swift • Haskell | Swift • Haskell • C# |
| C# • PHP | Dart • F# • Racket • Hack • PHP |
| Dart • F# • Racket • Hack • Python | JavaScript • Ruby • Python |
| JavaScript • Ruby | TypeScript • Erlang |
| TypeScript | Lua • JRuby • Perl |
| Erlang • Lua • Perl | |
| JRuby | |

man, 2013), to calculate different rankings for the analyzed software languages. In Table 3.8 we present four multi-objective rankings: time & peak memory, energy & time, energy & peak memory, and energy & time, & peak memory. For each ranking, each line represents a Pareto optimal set, that is, a set containing the languages that are equivalent to each other for the underlying objectives. In other words, each line is a single rank or position. A single software language in a position signifies that the language was clearly the best for the analyzed characteristics. Multiple languages in a line imply that a tie occurred, as they are essentially similar; yet ultimately, the languages lean slightly towards one of the objectives over the other as a slight trade-off.

The most common performance characteristics of software languages used to evaluate and choose them are execution time and peak memory usage. If we consider these two characteristics in our evaluation, `C`, `Pascal`, and `Go` are equivalent. However, if we consider energy and time, `C` is the best solution since it is dominant in both single objectives. If we prefer energy and peak memory, `C` and `Pascal` constitute the Pareto optimal set. Finally, analyzing all three characteristics, this scenario is very similar as for time and peak memory.

It is interesting to see that, when considering energy and time, the sets are usually reduced to one element. This means, that it is possible to actually decide which is the best language. This happens possibly because there is a mathematical relation between energy and time and thus they are usually tight together, thus being common that a language is dominant in both objectives at the same time. However, there are cases where this is not true. For instance, for `Pascal` and `Chapel` it is not possible to decide which one is the best as `Pascal` is better in energy and peak memory use, but worse in execution time. In these situations the developer needs to intervene and decide which is the most important aspect to be able to decide for one language.

It is also interesting to note that, when considering peak memory use, languages such as `Pascal` tend to go up in the ranking. Although this is natural, it is a difficult analysis to perform without information such as the one we present.

Given the information presented in Table 3.8 we can try to answer **RQ4: Can we automatically decide what is the best software language considering energy, time, and peak memory usage?** If the developer is only concerned with execution time and energy consumption, then yes, it is almost always possible to choose the best language. Unfortunately, if peak memory is also a concern, it is no longer possible to automatically decide for a single language. In all the other rankings most positions are composed by a set of Pareto optimal languages, that is, languages which are equivalent given the underlying characteristics. In these cases, the developer will need to make a decision and take into consideration which are the most important characteristics in each particular scenario, while also considering any fuctional/non-functional requirements necessary for the development of the application. Still, the information we provide in this study is quite important to help group languages by equivalence when considering the different objectives. For the best of our knowledge, this is the first time such work is presented. Note that we provide the information of each individual characteristic in Table 3.6 so the developer can actually understand each particular set (we do not show such information in Table 3.8 to avoid cluttering the section with to many tables with numbers).

## 3.4 Threats to Validity

The goal of our study was to both measure and understand the energetic behavior of several programming languages, allowing us to bring about a greater insight on how certain languages compare to each other mainly in terms of energy consumption, but also performance and memory. We present in this subsection some threats to the validity of our study, divided into four categories (Cook and Campbell, 1979), namely: conclusion validity, internal validity, construct validity, and external validity.

**Conclusion Validity** From our experiment it is clear that different programming paradigms and even languages within the same paradigm have a completely different impact on energy consumption, time, and memory. We also see interesting cases where the most energy efficient is not the fastest, and believe these results are useful for programmers. For a better comparison, we not only measured CPU energy consumption but also DRAM energy consumption. This allowed us to further understand the relationship between DRAM energy consumption and peak memory usage, while also understanding the behavior languages have in relation the energy usage derived from the CPU and DRAM. Additionally, the way we grouped the languages is how we felt is the most natural to compare languages (by programming paradigm, and how the language is executed). Thus, this was the chosen way to present the data. Nevertheless, all the data is available and any future comparison groups such as “.NET languages” or “JVM languages” can be very easily analyzed.

**Internal Validity** This category concerns itself with what factors may interfere with the results of our study. When measuring the energy consumption of the various different programming languages, other factors alongside the different implementations and actual languages themselves may contribute to variations, i.e. specific versions of an interpreter or virtual machine. To avoid this, we executed every language and benchmark solution equally. In each, we measured the energy consumption (CPU and DRAM), execution time, and peak memory 10 times, removed the furthest outliers, and calculated the median, mean, standard deviation, min, and max values. This allowed us to minimize the particular states of the tested machine, including uncontrollable system processes and software. However, the measured results are quite consistent, and thus reliable. In addition, the used energy measurement tool has also been proven to be very accurate.

**Construct Validity** We analyzed 27 different programming languages, each with roughly 10 solutions to the proposed problems, totaling out to almost 270 different cases. These solutions were developed by experts in each of the programming languages, with the main goal of "winning" by producing the best solution for performance time. While the different languages contain different implementations, they were written under the same rules, all produced the same exact output, and were implemented to be the fastest and most efficient as possible. Having these different yet efficient solutions for the same scenarios allows us to compare the different programming languages in a quite just manner as they were all placed against the same problem. Albeit certain paradigms or languages could have an advantage for certain problems, and others may be implemented in a not so traditional sense. Nevertheless, there is no basis to suspect that these projects are best or worst than any other kind we could have used.

**External Validity** We concern ourselves with the generalization of the results. The obtained solutions were the best performing ones at the time we set up the study. As the CLBG is an ongoing "competition", we expect that more advanced and more efficient solutions will substitute the ones we obtained as time goes on, and even the languages' compilers might evolve. Thus this, along with measurements in different systems, might produce slightly different resulting values if replicated. Nevertheless, unless there is a huge leap within the language, the comparisons might not greatly differ. The actual approach and methodology we used also favors easy replications. This can be attributed to the CLBG containing most of the important information needed to run the experiments, these being: the source code, compiler version, and compilation/execution options. Thus we believe these results can be further generalized, and other researchers can replicate our methodology for future work.

## 3.5 Conclusions

In this chapter, we first present an analysis and comparison of the energy efficiency of 27 well-known software languages from the popular software repository *The Computer Language Benchmarks Game*. We are able to show which were the most energy efficient software languages, execution types, and paradigms across 10 different benchmark problems.

Through also measuring the execution time and peak memory usage, we were able to relate both to energy, as to understand not only how memory usage affects energy consumption, but also how time and energy relate. This allowed us to understand if a faster language is always the most energy efficient. As we saw, this is not always the case.

As often times developers have limited resources and may be concerned with more than one efficiency characteristic we calculated which were the best/worst languages according to a combination of the previous three characteristics: Energy & Time, Energy & Peak Memory, Time & Peak Memory, and Energy & Time & Peak Memory.

Finally, the work in this chapter has been further extended, where we additionally analyzed the *Rosetta Code* chrestomathy repository and compare the results from this and *The Computer Language Benchmarks Game*, and has been submitted and is currently under review. Our work helps contribute another stepping stone in bringing more information to developers to allow them to become more energy-aware when programming.

# Chapter 4

# Spectrum-based Energy Leak Localization

*This chapter proposes a technique to detect energy inefficient fragments in the source code of a software system. Test cases are executed to obtain energy consumption measurements, and a statistical method, based on spectrum-based fault localization, is introduced to relate energy consumption to the source code. The result of our technique is an energy ranking of source code fragments pointing developers to possible energy leaks in their code. This technique was implemented in the SPELL toolkit.*

*In order to evaluate our technique, we conducted an empirical study where we asked participants to optimize the energy efficiency of a software system using the SPELL toolkit, while also having two other groups using no tool assistance and a profiler, respectively. We showed statistical evidence that developers using our technique and tool were able to improve the energy efficiency by 43% on average, and even out performing a runtime profiler when used for energy optimization.*

## 4.1 Introduction

In the previous chapter, we looked at the varying energy efficiency of different programming languages. With all the possible approaches to solve the same problem (different algorithms, design patterns, data structures, etc), even the most energy efficient language, such as `C`, can be written in an energy (or runtime) inefficient way.

To detect inefficiency at runtime, many programming languages offer advanced profilers which locate source code fragments which are possibly responsible for such inefficiencies. In the same line of reasoning, while IDEs have traditionally incorporated powerful advanced type and modular systems, testing and debugging frameworks, and other tools to improve software developers productivity and effectiveness, there is no concrete evidence that this trend has included techniques to optimize or even analyze source code energy consumption (Pinto et al., 2014a; Pang et al., 2016).

Indeed, if we compare energy-aware software engineering, or Energyware Engineering, with the long lasting series of engineering techniques that aim at helping software developers quickly construct correct programs with optimal runtime we see an obvious deficit. While the latter includes compiler constructions such as partial and/or runtime compilation, advanced garbage collectors or parallel execution, the former is still clearly more modest in terms of achievements (Lago, 2015).

In this chapter, we present a technique to locate energy inefficiencies within a program's source code, such as how a profiler would locate performance inefficiencies in a program's source code. Focusing on **TRQ2**, this chapter defines a technique, named SPELL - SPectrum-based Energy Leak Localization, which has been implemented in a tool, to determine *red* (energy inefficient) areas in software. The idea of this approach has been previously proposed in (Pereira et al., 2017a). In this work, we consider an *energy leak* synonymous to an energy inefficiency. In this context, a parallel is made between the detection of anomalies in the energy consumption of software during program execution, and the detection of faults in the execution of a program. Having this parallelism established, we adapted fault detection techniques, often used to investigate software bugs in program executions, to detect energetic faults in programs.

The software system to be analyzed is executed with a set of test cases, and components of such system (for example, packages, functions, loops, etc.) are instrumented to estimate/measure the energy consumption at runtime. Inefficient energy consumption, the so-called energy leaks, are interpreted in SPELL as program faults, and we adapt Spectrum-based Fault Localization (SFL) techniques (Abreu et al., 2009, 2007) to relate energy consumption to the system's source code. Our analysis associates different percentage of responsibility for the energy consumed to the different components of the underlying system. Thus, the result of our analysis is a ranking of components sorted by their likelihood of being responsible for energy leaks, essentially pinpointing and prioritizing the developer's attention on the most critical *red* spots in the analyzed system. Thus, giving more useful information to have better support in making decisions of what parts of the system need to be optimized, ultimately helping place a new stepping stone for energy-aware programming.

Our proposed technique is language independent, allowing the analysis of programs written in any programming language. Additionally, it is also context independent, allowing it to be applied to detect *red* areas on various levels of code. This means we could use it to detect the inefficiencies at different granularity levels, be that packages, classes, methods, functions, lines of code, etc. Even more so, the technique allows the use of different hardware component's energy values (CPU, DRAM, HDD, GPU, etc.) to compute the energy spent by a program, and may return the analysis of one specific factor (energy, time, or frequency of usage), or a global analysis considering all three factors.

Supported by our tool, our technique was able to identify potential energy leaks in the source code of concrete Java projects. Based on this identification, a set of expert Java programmers were then asked to improve the (energy) efficiency of those projects. The analysis of their success in doing so provided statistical evidence that the programs they ended up altering indeed consume less energy that the ones they were originally given, with an improvement, for different projects, between 15% and 74%.

Complementary, we compared the energy efficiency of the programs obtained as explained above against programs obtained from the original ones but by programmers working without the knowledge of any energy leak. From such comparison, we found statistical evidence that the difference is significant, in favor of the former: their performance is between 14% and 38% better.

A recurrent debate when optimizing energy consumption in software is whether a performance optimization is always an energy consumption optimization. Indeed, the `Energy` equation (EQ 3.1) does indicate that reducing time would imply a reduction in energy. However, the `Power` variable of the equation, not to be assumed as a constant, also has an impact alongside `Time`. Therefore, conclusions regarding this issue tend to diverge, where some works do support that optimizing for energy is optimizing for performance (Yuki and Rajopadhye, 2014), while many others have studied contexts where the opposite was observed (Pinto et al., 2014b; Trefethen and Thiyagalingam, 2013b; Couto et al., 2017b; Lima et al., 2016; Lorenzo et al., 2015; Nobre et al., 2018; Kambadur and Kim, 2014; Li et al., 2013; Abdulsalam et al., 2015; Pereira et al., 2016, 2017b). This suggests that only looking at performance might not be enough for optimizing energy.

Consequently, performance profilers are also not enough as they focus on indicating where one should optimize in order to improve execution time. Therefore, an *energy profiler* is needed if one wishes to optimize for energy consumption. Indeed we will show this is the case in Section 4.4.

In order to shed light and contribute to this debate with a particular focus on our context, we have complementary analyzed and compared our tool with an off the shelf profiler. This means that the experts were asked to improve the efficiency of the projects we considered with the guidance of SPELL and with the guidance of a runtime profiler. Our analysis provided statistical evidence that experts with access to located energy leaks were able to better optimize the energy consumption of those projects than when using a profiler, with improvements between 2% and 72%.

The contributions of this chapter are three-fold:

- A language independent technique to locate energy inefficient components in the source code of software systems. This technique is also independent of the approach used to measure (via external devices (Ferreira et al., 2013; Hähnel et al., 2012; Rotem et al., 2012)) or estimate (via predictive models (Liqat et al., 2013; Noureddine et al., 2015)) energy consumption (Section 4.2).

- An implementation of our technique as a Java-based tool that is able to (automatically) instrument the energy consumption of Java source code fragments, relying on the RAPL power estimation consumption model provided by Intel (Dimitrov et al., 2015; Hähnel et al., 2012), and to locate energy leaks in Java source code (Section 4.3).

- An evaluation of our technique and tool by detecting energy leaks in an empirical study. Programmers following SPELL recommendations were able to optimize programs to have energetic gains of 43% on average (Section 4.4).

This chapter concludes with final comments in Section 4.6.

## 4.2 Spectrum-based Energy Leak Localization

In this chapter, we present a language independent technique, termed SPELL – or Spectrum-based Energy Leak Localization – that localizes *red* areas in source code.

### 4.2.1 Spectrum-based Fault Localization

Our technique is based on spectrum-based fault localization (Abreu et al., 2009, 2007; Perez, 2018), a statistical analysis technique to detect faults in a program based on its implementation (source code).

In particular, SFL uses a hit spectrum (set of flags which reflect if a certain component is used or not in a particular run of the software) (Abreu et al., 2009; Passos et al., 2015) to build a matrix $A$ of dimension $n \times m$, where $m$ columns represent the different components (e.g. methods, classes) of a program during $n$ independent test executions. A component can be anything being analyzed, be this a program, a package, a class, a method, or even a line of code. An entry $a_{i,j}$ in $A$ of value 0 means that component $j$ was not executed in test execution $i$, and an entry of value 1 means that it was. Complementing the hit spectrum, SFL also uses a vector $e$, with $n$ elements, each of which indicates whether each of the $n$ tests succeeded or not.

Equation 4.1 illustrates the generic format of $A$ and $e$, and Equation 4.2 presents a concrete (simulated) example of the application of SFL with 3 test cases executed on a program with 4 components. The first line of the matrix $A$ in the example, e.g., reads as: in the execution of the first test case, components $c_1$, $c_2$ and $c_4$ were executed and component $c_3$ was not. The first element of $e$, i.e., the value 0, indicates that the execution of the first test case met its expected output (or in other words, that it did not fail).

$$\begin{array}{ccc} & m\text{ components} & \text{error detection} \\ n\text{ spectra} & \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{bmatrix} \end{array} \tag{4.1}$$

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \tag{4.2}$$

Using $(A, e)$, SFL tries to find which components are the most likely to be faulty by calculating: $n_{11}(j)$: the number of failed runs (indicated by the second 1 subscript) where component $j$ was involved (indicated by the first 1 subscript); $n_{10}(j)$: the number of successful runs in which component $j$ was involved; and $n_{01}(j)$: the number of failed runs where component $j$ was not involved. This produces a $3 \times m$ matrix $N$, where $m$ is the number of components in the program, and whose first/second/third line holds, for each component $j \in \{1, ..., m\}$, $n_{11}(j)$, $n_{10}(j)$ and $n_{01}(j)$, respectively.

Equation 4.3 shows the generic formulation of $N$ and Equation 4.4 shows its instance for the illustration in Equation 4.2. Finally, SFL applies the Ochiai coefficient of similarity (Equation 4.5) to each component $j \in [1..m]$ to indicate which component has the highest probability of being faulty. This produces the matrix $S$ given in Equation 4.6.

$$\begin{array}{c} m \text{ components} \\ \begin{bmatrix} n_{11}(1) & n_{11}(2) & \cdots & n_{11}(m) \\ n_{10}(1) & n_{10}(2) & \cdots & n_{10}(m) \\ n_{01}(1) & n_{01}(2) & \cdots & n_{01}(m) \end{bmatrix} \end{array} \tag{4.3}$$

$$\begin{bmatrix} 2 & 0 & 2 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 2 & 0 & 1 \end{bmatrix} \tag{4.4}$$

$$S_j = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}} \tag{4.5}$$

$$\begin{array}{cc} & 4 \text{ components} \\ S & \begin{bmatrix} 0.82 & 0.0 & 1.0 & 0.5 \end{bmatrix} \end{array} \tag{4.6}$$

Analyzing the elements of matrix $S$, we finally conclude that component 3 is the most likely to be faulty. The rationale for this is that such component was involved in all the test executions that failed and was not involved in the test execution that succeeded.

In our proposed technique, that we introduce in the following subsections, we also rely on the spectrum of a program, which allows us to discriminate the usage of each component, and in what cases it was used, further extracting more information of the components being analyzed.

### 4.2.2 Static Model Formalization

Similarly to SFL, the technique that we propose, SPELL, relies on an input matrix $A$, with dimension $n \times m$, where the $n$ lines also correspond to the number of test executions, and the $m$ columns to the number of components.[1] It is very important to note that by *test* we mean test scenarios which replicate a real-world usage of the application, i.e., system tests. The quality of the tested scenarios is also important because only with tests which stress the

[1] For a complete example please refer to Section 4.2.4.

components with different inputs replicating real-world scenarios, can one extract reliable information.

Differently to SFL, however, elements of $A$ actually hold triples. Each such element $\lambda_{i,j}$ is defined as follows:

$$\begin{cases} (0,0,0), \text{if component } j \text{ was not executed in test } i; \\ (E_{i,j}, N_{i,j}, T_{i,j}), \text{otherwise.} \end{cases}$$

The execution data of each component is therefore segmented in 3 categories: $E$ for energy consumption, $N$ for the number of executions and $T$ for the execution time.

In the energy consumption category, $E$, values of the energy consumed by different hardware components may be present, for example: CPU ($E_{\text{CPU}}$), DRAM ($E_{\text{DRAM}}$), fans ($E_{\text{fans}}$), HDD ($E_{\text{disk}}$), GPU ($E_{\text{GPU}}$), etc. At least one hardware component must be present.

The energy consumption values are expressed in the energy unit Joules (J), and the execution time is represented in milliseconds (ms). Finally, $N$ holds the number of executions (cardinality).

### 4.2.3 Energy Leak Localization

Now that we have our spectrum model, we can begin extracting useful information and localizing the energy leaks.

While in SFL there is an error vector to reason about the validity of the output obtained by a test, the SPELL analysis does not receive an error vector. This is because there is still no known understanding to signal what can be seen as an excess of energy consumption. Therefore, an error vector needs to be calculated, and we define two different perspectives to calculate error vectors and similarities. These perspectives, that we describe next, are called *Component Category Similarity* and *Global Similarity*. An interesting consideration to draw here is that use of the error vector cannot result in a binary decision (pass or fail) for a test execution; the criterion has to use continuous values to represent the *greenness* of a test.

***Component Category Similarity.*** The construction of this oracle was based on the regulation of greenhouse gas emissions for countries. After assessing how much is the total emission of gases in the different years, and depending on what each country contributed to these total emissions, each country is assigned a percentage of responsibility. We try to establish an analogy, where the $n$ years are the different tests, and the $m$ countries are the different components with the total for each category (energy, cardinality, and execution time), with the goal of assigning responsibilities to each component comparing with the total value.

To construct the error vector, we sum up all the values of all $m$ components for each test $i \in \{1, ..., n\}$, shown in Equation 4.7:

$$e_i = (\sum_{j=1}^{m} E_{i,j}, \sum_{j=1}^{m} N_{i,j}, \sum_{j=1}^{m} T_{i,j}) \tag{4.7}$$

As this is applied for all tests, we obtain Equation 4.8, a vector of triples called $e$:[2]

$$e = [e_1 \; e_2 \; \dots \; e_n]^T \tag{4.8}$$

With $(A, e)$ at hand, we now have an oracle model, and can begin localizing the energy leaks. Continuing our analogy of gas emissions, we need to relate the (3-category) data of each component with the total data. This is achieved *comparing* each component in $A$ with $e$. The main goal is to obtain a simple structure containing the similarity between each column $j \in \{1, ..., m\}$ in $A$ (which refers to the resources spent by component $j$) and vector $e$ (the total amount of resources that were spent). This similarity can be interpreted as how much component $j$ is responsible for each execution information of the total vector.

Assuming that $A(j)$ projects column $j$ from matrix $A$, the similarity between component $j$ and $e$ is defined as $\phi_j$, where:

$$\phi_j = (\alpha_1(A(j), e), \alpha_2(A(j), e), \alpha_3(A(j), e)) \tag{4.9}$$

Finally, assuming that for $x \in \{1, 2, 3\}$[3], $A(j, x)$ and $e(x)$ project the $x$-th element from all the triples of $A(j)$ and $e$, respectively, we define:

[2] We use superscript $T$ as the transpose of a matrix.

[3] Here, indexes 1, 2 and 3 represent E, N and T respectively.

$$\alpha_x(A(j),e) = \frac{\sum_{i=1}^{n} A(j,x)_i}{\sum_{i=1}^{n} e(x)_i} \tag{4.10}$$

To calculate the Ochiai coefficient similarity, we need to now be able to distinguish between a passed and a failed test. As previously stated, we cannot binarily define excess energy consumption. Thus, for this formula, we inspired ourselves in the Jaccard similarity coefficient (Real and Vargas, 1996). This coefficient is well-known and widely used to calculate the similarity coefficient between two vectors and has been used for a long period of time in the biology domain (Dombek et al., 2000; Rousseau, 1998). Using this definition, we calculate the similarity coefficient for each of the component's constituents $E$, $N$ and $T$.

Applying this similarity function to all components $j \in \{1,...,m\}$ will result in a row vector which represents, for each component and each test execution, their influence in the overall context for a given perspective ($E$, $N$ or $T$). The higher the similarity (the closer it is to 1) the more responsible it is in that category.

***Global Similarity.*** Using the similarity of each component category, we can have a parametrized analysis. However, it is also useful to have a value encoding the global similarity, allowing a numerical and global comparison between the different components.

The energy category $E$ of a software component $j$ can contain information on different hardware components such as *CPU*, *DRAM*, *GPU*, *fans*, and *disk*. These hardware components have different power consumption patterns that are known in advance. So, this information should be standardized according to the spontaneity of those hardware components.

Let us assume the following scenario:

- For a concrete test suite, software components 1 and 2 showed the same total energy consumption;

- However, they rely differently on hardware components A and B, wherein A on average consumes more power than B;

- The energy of component 1 only accounts for the use of component A;

- The energy of component 2 only accounts for the use of component B;

In spite of having the same consumption value, software components 1 and 2 should have their global similarity value influenced in different ways. As hardware component A has a higher average power consumption, component 1 is likely to contribute more to energy consumption than component 2 in scenarios that are not captured by the test suite in use.

A multiplicative factor can be defined for each hardware component and applied to allow standardization. Table 4.1 details the average power consumption for each component[4].

Table 4.1: Average W consumption for hardware components

| **Component name** | **Average power consumption (W)** |
|---|---|
| CPU | 102.5 |
| DRAM | 3.75 |
| Fans | 3.3 |
| Hard Drive | 7.5 |
| GPU | 187.5 |

Observing, e.g., that $CPU$ is responsible for 34% of the total power consumption on average, for each test $i \in \{1, ..., n\}$ and component $j \in \{1, ..., m\}$ we propose the formula:

$$\begin{aligned} EF_{i,j} = 0.34 \times E_{\mathrm{CPU}_{i,j}} + 0.01 \times E_{\mathrm{DRAM}_{i,j}} + 0.01 \times E_{\mathrm{fans}_{i,j}} + \\ 0.02 \times E_{\mathrm{disk}_{i,j}} + 0.62 \times E_{\mathrm{GPU}_{i,j}} \end{aligned} \tag{4.11}$$

Note this formula can be rewritten to account for any other combination of hardware parts (e.g., include a screen of a smartphone).

Now we can calculate the global value for each component:

$$\mathrm{global}_c(j) = [\mathrm{g}_c(1, j)\ \mathrm{g}_c(2, j)\ \dots\ \mathrm{g}_c(n, j)]^T \tag{4.12}$$

where

$$\mathrm{g}_c(i, j) = EF_{i,j} \times N_{i,j} \times T_{i,j} \tag{4.13}$$

This global value takes into consideration not only the energetic consumption of a com-

[4] `http://www.buildcomputers.net/power-consumption-of-pc-components.html`

ponent, but the cardinality and execution time all as one value. This allows us to have a better understanding of what are the most important components to look at and try to optimize. For example, a component A may consume twice the amount of energy of component B, but component B is used five times as often which might make it a good candidate to prioritize the attention on. This would give a weight to component B as it would seem to be a core part of the analyzed program.

Once we have the global values for each component, we can proceed to calculate the global error vector as:

$$\text{global}_e = [\text{g}_e(1)\ \text{g}_e(2)\ \ldots\ \text{g}_e(n)]^T \tag{4.14}$$

where

$$\text{g}_e(i) = \sum_{j=1}^{m} \text{g}_c(i,j) \tag{4.15}$$

Finally, we apply the similarity function $\alpha$ to each component $j$ to obtain the global similarity with the error vector, defined as $\psi$,

$$\psi(j) = \alpha(\text{global}_c(j), \text{global}_e) \tag{4.16}$$

where

$$\alpha(c,e) = \frac{\sum_{i=1}^{n} c_i}{\sum_{i=1}^{n} e_i} \tag{4.17}$$

Once again, the higher the similarity value, and closer it is to 1, the more responsible it is. We can rank the components by this global similarity and have initial indicators of where in the program we should prioritize our attention on, and which are the most important components to optimize.

### 4.2.4 An Example

To understand how the SPELL analysis works and see how it handles the execution data, we present in the following a concrete example.

Imagine a program written in any programming language, which has four different components (say, methods in Java), and is stressed with a test suite of five different inputs. This program has its energy consumption, usage frequency, and execution time identified for each component in each of the tests. Therefore, we can use the information of this program's execution and start the analysis.

Table 4.2: SPELL matrix built for the example program

| | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $e$ | $global_e$ |
|---|---|---|---|---|---|---|
| $t_1$ | $\begin{pmatrix}\{37,6\}\\1\\75\end{pmatrix}$ | $\begin{pmatrix}\{61,11\}\\2\\102\end{pmatrix}$ | $\begin{pmatrix}0\\0\\0\end{pmatrix}$ | $\begin{pmatrix}\{42,18\}\\1\\34\end{pmatrix}$ | $\begin{pmatrix}\{140,35\}\\4\\211\end{pmatrix}$ | 5693.04 |
| $t_2$ | $\begin{pmatrix}\{38,7\}\\3\\77\end{pmatrix}$ | $\begin{pmatrix}\{50,7\}\\1\\103\end{pmatrix}$ | $\begin{pmatrix}\{34,5\}\\2\\42\end{pmatrix}$ | $\begin{pmatrix}\{44,21\}\\1\\37\end{pmatrix}$ | $\begin{pmatrix}\{166,40\}\\7\\259\end{pmatrix}$ | 6295.43 |
| $t_3$ | $\begin{pmatrix}\{36,6\}\\1\\73\end{pmatrix}$ | $\begin{pmatrix}\{58,10\}\\1\\102\end{pmatrix}$ | $\begin{pmatrix}\{35,5\}\\1\\43\end{pmatrix}$ | $\begin{pmatrix}0\\0\\0\end{pmatrix}$ | $\begin{pmatrix}\{129,21\}\\3\\218\end{pmatrix}$ | 3433.39 |
| $t_4$ | $\begin{pmatrix}\{37,7\}\\3\\74\end{pmatrix}$ | $\begin{pmatrix}\{66,18\}\\2\\105\end{pmatrix}$ | $\begin{pmatrix}0\\0\\0\end{pmatrix}$ | $\begin{pmatrix}\{61,20\}\\2\\43\end{pmatrix}$ | $\begin{pmatrix}\{164,45\}\\7\\222\end{pmatrix}$ | 9359.34 |
| $t_5$ | $\begin{pmatrix}\{39,9\}\\2\\75\end{pmatrix}$ | $\begin{pmatrix}\{54,9\}\\3\\100\end{pmatrix}$ | $\begin{pmatrix}\{51,15\}\\4\\60\end{pmatrix}$ | $\begin{pmatrix}\{65,20\}\\2\\60\end{pmatrix}$ | $\begin{pmatrix}\{209,53\}\\11\\295\end{pmatrix}$ | 14411.1 |
| $\phi$ | $\begin{pmatrix}\{0.2314,0.1804\}\\0.3125\\0.3104\end{pmatrix}$ | $\begin{pmatrix}\{0.3577,0.2835\}\\0.2813\\0.4249\end{pmatrix}$ | $\begin{pmatrix}\{0.1485,0.1289\}\\0.2188\\0.1203\end{pmatrix}$ | $\begin{pmatrix}\{0.2624,0.4072\}\\0.1875\\0.1444\end{pmatrix}$ | | |
| $\psi$ | 0.2464 | 0.4674 | 0.1451 | 0.1411 | | |

We can see the entire model of the SPELL analysis defined in Table 4.2, but let us construct it step by step. The input data can be seen in the top left $5 * 4$ matrix shown in Table 4.2 where each component and each test has a triple of three categories.

This triple contains the CPU energy consumption value, the number of times that software component was used, and the execution time:

$$\begin{pmatrix} E_{CPU} \\ N \\ T \end{pmatrix}$$

In this case, the only hardware component shown is the CPU for the sake of simplicity of presenting our technique, but it still straightforwardly applies if energy consumption information of more hardware components is available.

***Similarity by Component's Category.*** Having these inputs defined in SPELL, we will first calculate the software component similarities. We begin by building the error ($e$ vector). To do so, for each test, we sum all the values of each category of the component data. This is shown on the right hand side of our example matrix under the $e$ column. Next, we calculate each of the component's category similarity. For example, for the energy category of component $c_1$ we will have the following:

$$\alpha_1(A(c_1), e) = \frac{37+38+36+37+39}{140+166+129+164+209} = 0.2314$$

This would be applied for the other two categories, and for each of the other components, producing the results seen in the similarity by component's category row $\phi$ in Table 4.2.

***Global Similarity.*** For the global similarity, we begin by calculating the global values of each component, and afterwards our new total global value vector. We obtain values $global_c(1)$ and $global_e$:

$$\text{global}_c(1) = [948.0\ 2991.45\ 894.25\ 2799.42\ 1992.00]^T$$

$$\text{global}_e = [5693.04\ 6295.43\ 3433.39\ 9359.34\ 14411.1]^T$$

Finally, we use the coefficient similarity $\psi(1)$, to obtain the global similarity value for component $c_1$ of 0.246642. Applying this to each component, we obtain the results under the global similarity $\psi$.

***Analysis.*** Having all the needed information to analyze this program we begin extracting useful information. Reading the global similarity values ($\psi$), we can see which component has the highest probability of having an energy leak with the order of $c_2$ (with similarity of 0.4678), $c_1$ (with 0.2466), $c_3$ (with 0.1450), and finally $c_4$ (with 0.1406). This indicates to the developer that he should first consider looking into component $c_2$ to try to improve the energy consumption of this program.

An advantage of this technique, which highlights the complementary perspectives of the two types of similarities that we consider, is that it can tell, besides having a global view of the component, indicators of why the component is faulty. For example, $c_2$ is given the highest global similarity value. If we now look into its category similarity values ($\phi$), we can see that this component has the highest energy and time execution similarities (0.3577 and 0.4249, respectively), and the second highest cardinality similarity (0.2813).

## 4.3 SPELL Toolkit

As previously stated, this technique is language independent, where the only required input is a matrix representing the tests, components, and categories. As a proof of concept we have implemented SPELL toolkit in Java and for Java systems.

To use SPELL in detecting energy leaks in software applications we have developed two language dependent support tools in our SPELL toolkit. The first one consists of an energy monitoring instrumentation tool: it automatically instruments the source code of each method in a class with calls to the API of a Java energy estimation framework during the beginning and end of each method (including before any nested returns), and writes the execution trace and energy consumption in a file at the end of the execution.

This tool uses Intel's Runtime Average Power Limit (RAPL) (Dimitrov et al., 2015), and jRAPL (Liu et al., 2015). This allowed us to record precise energy consumption measurements from the CPU, since RAPL is a very reliable tool (as shown in (Hähnel et al., 2012) and (Rotem et al., 2012)). Note the technique is not limited to only using RAPL to measure energy, but any energy measurement framework or tool can be used allowing the analysis of other languages, or even other domains such as Android applications when using Trepn (Hoque et al., 2015) or Monsoon (Monsoon, 2018) for example.

The second tool uses the output execution trace of our instrumentation tool as input to construct the SPELL matrix. This automatically looks at the method calls, and aggregates the energy consumption, execution time, and frequency of methods into our matrix representation.

As our technique is language independent, one may easily develop front-end tools for other languages to measure the consumption and generate the SPELL matrix to run the analysis. Our core tool, and its two supporting tools are open source[5]. The toolkit contains more information on how to run the tools, and the representation of the input and output data of each.

## 4.4 Empirical Evaluation

One of our goals is to help provide programmers ways to become more energy-aware. Additionally, our SPELL technique is to be used by developers to help them detect energy leaks (or energy inefficiencies) on a source code level. Thus, we designed an empirical study to understand and answer the following research questions:

- **RQ1** *Can the energy leaks identified by SPELL help developers improve the overall energy efficiency of their programs?* Answering this question allows us to understand if in fact SPELL can detect areas in the source code where there is a probability of an energy hotspot occurring. If SPELL were to consistently point to areas in the code, where in turn the developer would go ahead and alter, and the energy efficiency improves, we can assume it is indeed identifying energy leaks. If it were to indicate areas where by the developer's changes actually brought about a deterioration in the energy consumption, then SPELL is not able to identify energy leaks.

- **RQ2** *Are the programs improved by developers assisted by SPELL significantly more energy efficient than the programs improved by developers without tool-assistance?* If a developer using SPELL is not significantly producing more energy efficient programs,

[5] GitHub: `https://github.com/greensoftwarelab/SPELL`

then it would mean there is no need to use such a tool as a developer's own knowledge is enough for such a task.

- **RQ3** *Are the programs improved by developers assisted by SPELL significantly more energy efficient than the programs improved by developers with an off the shelf profiler?* This question is very important, as one might assume that SPELL is nothing more than another profiler, or that using an off the shelf profiler is enough to improve the energy efficiency of a program. Additionally, answering this question will allow us to understand if looking at a program's execution performance, and optimizing based off that information is enough to optimize for energy.

The following sections will describe in detail the design, execution, results and discussion of our empirical study.

### 4.4.1 Experimental Setup

In order to evaluate our technique with programmers, and to answer the previous research questions, we have designed and conducted an empirical study.

Participants in this study were selected from a candidate group that replied to an invitation that we publicized among our departments and two software houses. The selection process consisted of a self assessment step, in which to be eligible, candidates had to consider themselves experienced Java programmers. Ultimately, 15 programmers were selected: 12 male and 3 female; all with computer science background and/or professional experience: 6 postdoc researchers, 6 PhD students, and 3 professional programmers.

For this study, we asked programmers to try to optimize the energy consumption of a program in three different scenarios: a control group, with our SPELL technique, and with a profiler.

The participants were then arranged into groups of threes (one for each scenario) according to their professional status. Essentially, the outcome was 5 different groups of 3: 2 groups of postdoc, 2 groups of PhD students, and 1 group of professional programmers.

In order to support the study, we initially considered 63 Java projects from an object-oriented course for computer science students, where students were asked to build a journalism support platform, where users (Collaborators, Journalists, Readers, and Editors) can write chronicles and reports, give likes and comments, and perform other tasks.

We filtered these projects to obtain the ones which passed a set of system tests designed by the course instructors, and all 16 unique operations and functional requirements were implemented (posting chronicles/reports, registering users, writing comments, viewing top commented, etc). By doing so, we ended up with 42 comparable and differently implemented projects[6].

Due to allowing certain operations such as *Listing Comments*, and to provide an initial "warm-up", for each of the 42 projects we populated the system with an initial set-up with: 3000 Chronicles, 3000 Reports, 7655 Likes, 8586 Comments, 60 Collaborators, 60 Journalists, 406 Readers, and 15 Editors.

To execute the projects, we defined 7 test scenarios (i.e., 7 scenarios replicating real program usage), simulating 7 days of interaction with the platform. Each test scenario was made up of a random number (varying between the hundreds and the thousands) of the 16 unique operations. While each test scenario contained each of the 16 unique operations, the randomness allowed certain *days* to have more of a certain type of operation than others. For example Tests 5 and 6 contain more write operations, while the others contain more read and lookup operations.

For selecting which projects would actually be explored in our study, we have resorted to SPELL itself. Indeed, we have used the test scenarios described previously to calculate the global similarity value for each of the 42 software projects (each component was defined as one project, so 42 components were analyzed in total). Project 1 ($P_1$) obtained a global similarities of 0.4259, $P_{47}$ of 0.4093, $P_{49}$ of 0.1439, $P_6$ of 0.0042, $P_{59}$ of 0.0042, $P_{36}$ of 0.0029, $P_{17}$ of 0.0015, etc.

The reason for using SPELL here is that a higher global similarity represents a more probable scenario where an energy leak may be occurring as it is more responsible for the overall

[6] `http://www.di.uminho.pt/~jas/Research/spellStudies.rar`

consumption, and means developers should focus their attention on that specific component as it is the most energetically problematic one.

This gives us a ranking of the most problematic projects according to SPELL. However, still we do not know where to look at to try to optimize. Thus, applying SPELL to each program but considering components as methods would allow us to obtain a ranking of methods that are the most/least responsible for energy consumption. So, we ran the SPELL analysis on the 5 worst ranking projects, so that 1 project is considered by each of our participant groups, to localize where energy leaks are present on a method level.

The global similarity for each of the projects' methods where $\psi > 0.08$ or to show at least 2 methods per project is shown in Table 4.3. The first column indicates the project, while the second column states the problematic Class.method according to SPELL, and the third column states the global similarity value. The higher it is, the more responsible it is for the global inefficiency, and where a problem is most probable to be found.

Table 4.3: SPELL and Profiler ranking of methods from Projects $P_1$, $P_{47}$, $P_{49}$, $P_6$, and $P_{59}$

| Proj. | Method (SPELL) | $\psi$ | % | Method (Profiler) |
|---|---|---|---|---|
| $P_1$ | voteInReport | 0.97 | 95.3 | voteInReport |
| | getUserLoggedInType | 0.02 | 2.7 | listArticlesByTheme |
| $P_{47}$ | listAllChronicles | 0.57 | 51.1 | addComment |
| | listAllReports | 0.15 | 15.8 | listAllChronicles |
| | chronicleExist | 0.12 | 7.8 | chronicleExist |
| $P_{49}$ | Like | 0.27 | 29.3 | ListTheme |
| | ListComments | 0.19 | 27.5 | ListTopic |
| | AddComment | 0.10 | 6.5 | ListComments |
| | ListTopic | 0.08 | 5.5 | Like |
| $P_6$ | printNoticiaTopicoTema | 0.40 | 32.8 | listChronicles |
| | printCronicaTopicoTema | 0.20 | 24.9 | listReports |
| | isLogged | 0.15 | 13.2 | topChronicles |
| $P_{59}$ | getArticle | 0.94 | 81.9 | getArticle |
| | vote | 0.04 | 12.4 | getComments |

As a profiling tool, we turned to the NetBeans (8.2) Profiler[7], a Java profiler integrated into the NetBeans IDE. By using the profiling methods mode, and more specifically the *Hot spots*

[7] https://profiler.netbeans.org/

tool[8], we were able to see what methods the tool was uncovering as performance bottlenecks. Presented in Table 4.3 are the methods pointed by the profiler, under the *Method (Profiler)* column, and under the *%* column is the total time (CPU) of the method as stated by the *Hot spots* tools. Just as with SPELL, the higher the value, the more problematic the method is.

To further characterize the projects that we used, we show in Table 4.4 concrete metrics about them. Each line represents the metrics for a single project, with the last 3 being the minimum, average, and maximum values. Columns 2–4 are the number of classes, methods, and lines of code (LOC), respectively. Column 5 represents the max cyclomatic complexity present in that project from a single method. Finally, column 6 represents the average cyclomatic complexity for that class, excluding methods with a complexity of 0 or 1.

Table 4.4: Software Metrics for Projects $P_1$, $P_{47}$, $P_{49}$, $P_6$, and $P_{59}$

| | **Classes** | **Methods** | **LOC** | **Max Comp.** | **Avg Comp.** |
|---|---|---|---|---|---|
| $P_1$ | 38 | 181 | 1037 | 26 | 5.05 |
| $P_{47}$ | 32 | 155 | 923 | 25 | 2.38 |
| $P_{49}$ | 27 | 131 | 811 | 17 | 3.37 |
| $P_6$ | 15 | 122 | 691 | 37 | 5.04 |
| $P_{59}$ | 32 | 151 | 905 | 11 | 3.45 |
| **Min** | 15 | 122 | 691 | 11 | 2.38 |
| **Avg** | 28.8 | 148 | 873.4 | 23.2 | 3.86 |
| **Max** | 38 | 181 | 1037 | 37 | 5.05 |

In order to analyze the energy consumption of all projects, we have instrumented their code using the SPELL toolkit. The instrumentation code is realized with calls to RAPL, which allows us to measure and monitor the energy that is being consumed.

Our measurements were made on a system with the following specifications: Linux 3.13.0-53-generic operating system, with 8GB of RAM, and a Sandy Bridge Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz. In the architecture of our machine, RAPL is only able to provide information regarding the energy consumption of the CPU. Each test was executed 30 times (Hogg and Tanis, 1977), and we extracted the cardinality and average values for both the time and CPU energy consumption (of the specific test and not the initial population as to only analyze the tests).

[8] `https://profiler.netbeans.org/docs/help/5.5/snap_cpu.html`

### 4.4.2 Energy Optimization with Developers

In the previous subsection, we applied SPELL within a program to localize what methods are possibly causing energy leaks, and used a profiler to identify possible performance bottlenecks. But is the SPELL information reliable and useful for developers when trying to optimize their program to become *greener*? And how does SPELL compare to a pure performance analysis?

To answer this, we asked our 5 groups of participants to analyze one of the 5 projects and, to the best of their knowledge, optimize its energetic performance. Each group was randomly assigned one of the 5 projects. They were also given the project's description and input examples to familiarize themselves with the software requirements and structure, and allowed them to navigate the program looking at whatever they felt they needed to understand. We asked them to dedicate approximately 30 minutes to first understand the project. Each participant was given a series of test cases and their expected outputs. This allowed them to verify if they changed the business logic when refactoring and optimizing the project.

Finally, we randomly chose one of the participants in each group to have access to information produced by our SPELL technique for the given project, and one to have access to information produced by the NetBeans profiler. Both were asked to closely follow the recommendations of the tools. Thus, for each group/project, one participant used SPELL, one used a profiler, and one used no tool (control-group). The only imposed restriction was to try to dedicate no more than 2 hours to optimize the project.

We instructed them to take note of the time they began and, when they were satisfied with their work and felt they did indeed made an impact to the performance, to take note of the end time. They were also asked to describe what changes they made (or, if due to time restrictions, what changes they would make), and if they (non control-group participants) found it beneficial to have the data produced by the tools when optimizing for energy, or if they (control-group participants) would have found it impactful.

Afterwards, we collected all the refactored programs (3 different variations for each), made sure everything produced the expected output, and measured the energy consumption and execution time from these refactorings.

### 4.4.3 Results

Table 4.5 presents the results for Projects $P_1$, $P_{47}$, $P_{49}$ $P_6$, and $P_{59}$, respectively. Each row under *Test* represents the data for one of the 7 tests scenarios, with the final row being the totals and global values. The first block of 2 columns represents the data for the original project, showing Joules (J) and execution time in milliseconds (ms). The second, third, and fourth block (with 4 columns each) represent the measured energy, execution time, and energetic gain percentage (relative to the original project) for the control group, SPELL group, and profiler group, respectively. The time taken to optimize is shown in parentheses above each block next to the group name. A graphical representation of the global percentage of gains for each project can be seen in Figure 4.1, where the blue dotted bars represents the energy improvement (Joules) and the orange bars represent the execution time improvement (ms).

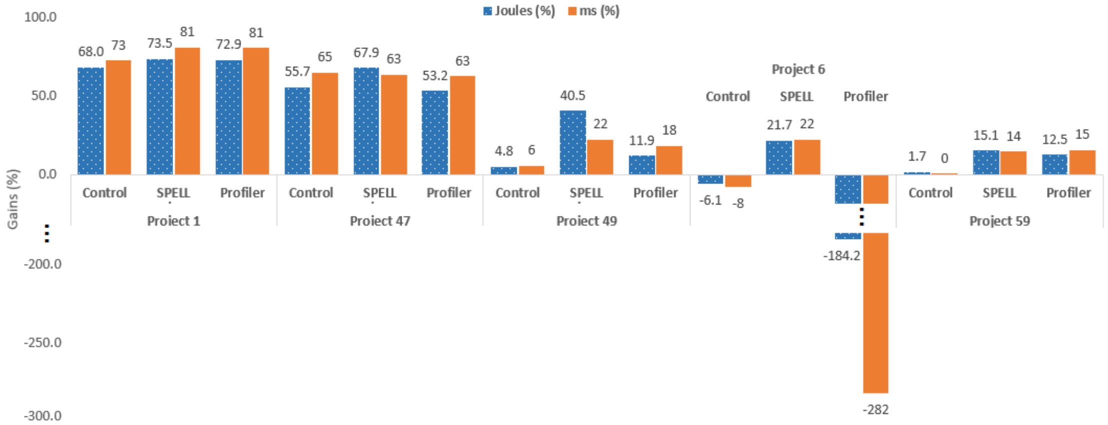


Figure 4.1: Global percentage of gains for all projects

The energy metrics (Abdulsalam et al., 2015) shown in Table 4.6, and the GPS-UP software energy efficiency quadrant graphs shown in Figure 4.2, represent the ratios between the original and optimized projects. *Greenup* (GU) is the ratio of the total energy consumption, *Speedup* (SU) is the ratio of the execution time, and *Powerup* (PU) is the average power consumption ratio. The higher the *Greenup* and *Speedup*, the more energy and performance efficient the optimization is. *Powerup* represents the power effects of the optimization, where less than 1 implies average power savings, while a greater than 1 implies more power consumed. Category (*Cat*) represents where the optimization falls under, with 1–4 being green

Table 4.5: Study results from all projects

| | | Original | | Control - (2h05) | | | | SPELL - (1h13) | | | | Profiler - (1h33) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Gain (%) | | | | Gain (%) | | | | Gain (%) | |
| | **Test** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** |
| $P_1$ | 1 | 93.1 | 8621 | 17.8 | 1289 | 80.9 | 85 | 13.9 | 913 | 85.0 | 89 | 13.8 | 888 | 85.2 | 90 |
| | 2 | 20.3 | 1645 | 11.3 | 1796 | 44.2 | -9 | 8.8 | 537 | 56.4 | 67 | 9.3 | 567 | 54.0 | 66 |
| | 3 | 87.4 | 7982 | 15.7 | 1146 | 82.0 | 86 | 13.8 | 869 | 84.2 | 89 | 13.8 | 869 | 84.2 | 89 |
| | 4 | 32.0 | 2666 | 15.0 | 1005 | 53.0 | 62 | 13.4 | 859 | 58.0 | 68 | 14.0 | 905 | 56.2 | 66 |
| | 5 | 58.5 | 5322 | 14.9 | 985 | 74.5 | 81 | 11.8 | 784 | 79.8 | 85 | 11.9 | 785 | 79.6 | 85 |
| | 6 | 17.9 | 1343 | 15.0 | 753 | 16.1 | 44 | 11.7 | 725 | 34.6 | 46 | 12.1 | 753 | 32.2 | 44 |
| | 7 | 14.0 | 928 | 13.6 | 850 | 2.9 | 8 | 11.9 | 725 | 14.8 | 22 | 12.6 | 780 | 10.1 | 16 |
| | **Total** | 323.1 | 28507 | 103.3 | 7824 | 68.0 | 73 | 85.5 | 5413 | 73.5 | 81 | 87.6 | 5547 | 72.9 | 81 |

| | | Original | | Control - (2h02) | | | | SPELL - (1h16) | | | | Profiler - (0h44) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Gain (%) | | | | Gain (%) | | | | Gain (%) | |
| | **Test** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** |
| $P_{47}$ | 1 | 51.4 | 4487 | 19.3 | 1216 | 62.3 | 73 | 13.3 | 1160 | 74.1 | 74 | 21.5 | 1417 | 58.2 | 68 |
| | 2 | 18.2 | 1235 | 10.3 | 641 | 43.2 | 48 | 7.6 | 798 | 58.4 | 35 | 10.1 | 643 | 44.2 | 48 |
| | 3 | 36.7 | 2972 | 14.4 | 899 | 60.8 | 70 | 11.1 | 1018 | 69.8 | 65 | 16.1 | 1022 | 56.2 | 66 |
| | 4 | 44.3 | 3683 | 18.1 | 1197 | 59.2 | 68 | 11.2 | 1024 | 74.7 | 72 | 19.1 | 1268 | 56.8 | 66 |
| | 5 | 39.3 | 3323 | 18.3 | 1266 | 53.5 | 62 | 14.1 | 1267 | 64.1 | 61 | 18.8 | 1270 | 52.3 | 62 |
| | 6 | 26.9 | 2024 | 15.6 | 991 | 42.1 | 51 | 13.0 | 1166 | 51.7 | 42 | 16.0 | 1008 | 40.4 | 50 |
| | 7 | 30.0 | 2311 | 13.3 | 836 | 55.5 | 64 | 8.9 | 882 | 70.3 | 61 | 13.9 | 881 | 53.5 | 62 |
| | **Total** | 246.7 | 20034 | 109.3 | 7045 | 55.7 | 65 | 79.1 | 7316 | 67.9 | 63 | 115.5 | 7510 | 53.2 | 63 |

| | | Original | | Control - (1h49) | | | | SPELL - (0h47) | | | | Profiler - (0h36) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Gain (%) | | | | Gain (%) | | | | Gain (%) | |
| | **Test** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** |
| $P_{49}$ | 1 | 40.2 | 2508 | 39.8 | 2458 | 1.1 | 2 | 23.4 | 2085 | 41.9 | 17 | 33.5 | 2056 | 16.8 | 18 |
| | 2 | 19.9 | 1392 | 17.1 | 1210 | 14.2 | 13 | 9.0 | 972 | 54.8 | 30 | 15.9 | 943 | 20.0 | 32 |
| | 3 | 34.1 | 2094 | 33.4 | 2042 | 2.1 | 2 | 15.8 | 1539 | 53.7 | 27 | 29.8 | 1728 | 12.6 | 17 |
| | 4 | 36.0 | 2202 | 36.0 | 2200 | 0.1 | 0 | 17.0 | 1701 | 52.7 | 23 | 31.7 | 1865 | 12.0 | 15 |
| | 5 | 24.5 | 1572 | 22.0 | 1380 | 10.1 | 12 | 20.2 | 1280 | 17.5 | 19 | 22.9 | 1341 | 6.4 | 15 |
| | 6 | 19.9 | 1240 | 19.1 | 1182 | 4.0 | 5 | 17.1 | 1092 | 13.9 | 12 | 19.1 | 1063 | 3.9 | 14 |
| | 7 | 29.3 | 1813 | 26.8 | 1644 | 8.5 | 9 | 18.8 | 1289 | 36.0 | 29 | 26.8 | 1501 | 8.7 | 17 |
| | **Total** | 203.9 | 12821 | 194.1 | 12115 | 4.8 | 6 | 121.3 | 9958 | 40.5 | 22 | 179.7 | 10497 | 11.9 | 18 |

| | | Original | | Control - (2h13) | | | | SPELL - (1h22) | | | | Profiler - (2h00) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Gain (%) | | | | Gain (%) | | | | Gain (%) | |
| | **Test** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** |
| $P_6$ | 1 | 18.5 | 1351 | 19.0 | 1460 | -2.6 | -8 | 12.9 | 966 | 30.2 | 28 | 79.7 | 7781 | -330.9 | -476 |
| | 2 | 9.4 | 600 | 10.3 | 668 | -9.8 | -11 | 7.8 | 487 | 17.0 | 19 | 13.9 | 1072 | -47.9 | -79 |
| | 3 | 13.0 | 878 | 14.2 | 969 | -9.8 | -10 | 9.8 | 663 | 24.2 | 24 | 33.7 | 3010 | -160.6 | -243 |
| | 4 | 21.2 | 1519 | 21.7 | 1571 | -2.1 | -3 | 17.1 | 1215 | 19.5 | 20 | 72.4 | 6953 | -240.9 | -358 |
| | 5 | 13.1 | 939 | 14.8 | 1061 | -13.0 | -13 | 10.7 | 732 | 18.2 | 22 | 18.1 | 1453 | -37.8 | -55 |
| | 6 | 12.0 | 804 | 13.2 | 902 | -10.3 | -12 | 10.3 | 673 | 14.3 | 16 | 14.0 | 1010 | -16.3 | -26 |
| | 7 | 18.7 | 1254 | 19.1 | 1306 | -2.2 | -4 | 14.3 | 986 | 23.6 | 21 | 69.4 | 6759 | -270.5 | -439 |
| | **Total** | 106.0 | 7345 | 112.4 | 7937 | -6.1 | -8 | 83.0 | 5723 | 21.7 | 22 | 301.2 | 28038 | -184.2 | -282 |

| | | Original | | Control - (1h58) | | | | SPELL - (1h04) | | | | Profiler - (1h21) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | Gain (%) | | | | Gain (%) | | | | Gain (%) | |
| | **Test** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** | **J** | **ms** |
| $P_{59}$ | 1 | 13.2 | 989 | 13.3 | 992 | -0.6 | 0 | 11.0 | 803 | 16.5 | 19 | 11.3 | 797 | 14.9 | 19 |
| | 2 | 8.0 | 453 | 7.1 | 436 | 11.5 | 4 | 5.5 | 391 | 31.1 | 14 | 6.6 | 395 | 18.0 | 13 |
| | 3 | 10.2 | 722 | 10.4 | 730 | -1.8 | -1 | 8.5 | 643 | 16.6 | 11 | 9.4 | 630 | 8.3 | 13 |
| | 4 | 10.5 | 763 | 10.4 | 758 | 1.1 | 1 | 9.8 | 679 | 6.8 | 11 | 9.5 | 648 | 10.0 | 15 |
| | 5 | 11.5 | 840 | 11.5 | 846 | -0.5 | -1 | 9.8 | 681 | 14.9 | 19 | 10.0 | 687 | 12.5 | 18 |
| | 6 | 10.0 | 633 | 9.3 | 611 | 7.0 | 3 | 8.6 | 548 | 14.1 | 13 | 8.4 | 539 | 16.5 | 15 |
| | 7 | 7.8 | 529 | 8.0 | 535 | -2.5 | -1 | 7.3 | 472 | 7.0 | 11 | 7.3 | 472 | 7.0 | 11 |
| | **Total** | 71.3 | 4929 | 70.1 | 4908 | 1.7 | 0 | 60.5 | 4215 | 15.1 | 14 | 62.4 | 4168 | 12.5 | 15 |

categories (a 1 represents better performance and energy efficiency, while a 3 represents better performance at the cost of energy efficiency), and 5–8 being red categories. A fully detailed description of these metrics, categories, and how to interpret these values can be found in the original work (Abdulsalam et al., 2015).

Table 4.6: GPS-UP Software Energy Efficiency metrics

| | | **Control** | | | | **SPELL** | | | | **Profiler** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Test** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** |
| $P_1$ | 1 | 5.24 | 6.69 | 1.28 | 3 | 6.68 | 9.44 | 1.41 | 3 | 6.76 | 9.71 | 1.43 | 3 |
| | 2 | 1.79 | 0.92 | 0.51 | 4 | 2.29 | 3.06 | 1.34 | 3 | 2.18 | 2.90 | 1.33 | 3 |
| | 3 | 5.57 | 6.97 | 1.25 | 3 | 6.32 | 9.18 | 1.45 | 3 | 6.32 | 9.18 | 1.45 | 3 |
| | 4 | 2.13 | 2.65 | 1.25 | 3 | 2.38 | 3.10 | 1.30 | 3 | 2.28 | 2.95 | 1.29 | 3 |
| | 5 | 3.92 | 5.40 | 1.38 | 3 | 4.96 | 6.79 | 1.37 | 3 | 4.91 | 6.78 | 1.38 | 3 |
| | 6 | 1.19 | 1.78 | 1.50 | 3 | 1.53 | 1.85 | 1.21 | 3 | 1.47 | 1.78 | 1.21 | 3 |
| | 7 | 1.03 | 1.09 | 1.06 | 3 | 1.17 | 1.28 | 1.09 | 3 | 1.11 | 1.19 | 1.07 | 3 |
| | **Total** | 3.13 | 3.64 | 1.16 | 3 | 3.78 | 5.27 | 1.39 | 3 | 3.69 | 5.14 | 1.39 | 3 |

| | | **Control** | | | | **SPELL** | | | | **Profiler** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Test** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** |
| $P_{47}$ | 1 | 2.66 | 3.69 | 1.39 | 3 | 3.87 | 3.87 | 1.00 | 1 | 2.39 | 3.17 | 1.32 | 3 |
| | 2 | 1.76 | 1.92 | 1.09 | 3 | 2.40 | 1.55 | 0.64 | 1 | 1.79 | 1.92 | 1.07 | 3 |
| | 3 | 2.55 | 3.30 | 1.30 | 3 | 3.31 | 2.92 | 0.88 | 1 | 2.28 | 2.91 | 1.27 | 3 |
| | 4 | 2.45 | 3.08 | 1.26 | 3 | 3.96 | 3.60 | 0.91 | 1 | 2.31 | 2.90 | 1.26 | 3 |
| | 5 | 2.15 | 2.63 | 1.22 | 3 | 2.79 | 2.62 | 0.94 | 1 | 2.10 | 2.62 | 1.25 | 3 |
| | 6 | 1.73 | 2.04 | 1.18 | 3 | 2.07 | 1.74 | 0.84 | 1 | 1.68 | 2.01 | 1.20 | 3 |
| | 7 | 2.25 | 2.77 | 1.23 | 3 | 3.37 | 2.62 | 0.78 | 1 | 2.15 | 2.62 | 1.22 | 3 |
| | **Total** | 2.26 | 2.84 | 1.26 | 3 | 3.12 | 2.74 | 0.88 | 1 | 2.14 | 2.67 | 1.25 | 3 |

| | | **Control** | | | | **SPELL** | | | | **Profiler** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Test** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** |
| $P_{49}$ | 1 | 1.01 | 1.02 | 1.01 | 3 | 1.72 | 1.20 | 0.70 | 1 | 1.20 | 1.22 | 1.01 | 3 |
| | 2 | 1.17 | 1.15 | 0.99 | 1 | 2.21 | 1.43 | 0.65 | 1 | 1.25 | 1.48 | 1.18 | 3 |
| | 3 | 1.02 | 1.03 | 1.00 | 3 | 2.16 | 1.36 | 0.63 | 1 | 1.14 | 1.21 | 1.06 | 3 |
| | 4 | 1.00 | 1.00 | 1.00 | 3 | 2.11 | 1.29 | 0.61 | 1 | 1.14 | 1.18 | 1.04 | 3 |
| | 5 | 1.11 | 1.14 | 1.02 | 3 | 1.21 | 1.23 | 1.01 | 3 | 1.07 | 1.17 | 1.10 | 3 |
| | 6 | 1.04 | 1.05 | 1.01 | 3 | 1.16 | 1.14 | 0.98 | 1 | 1.04 | 1.17 | 1.12 | 3 |
| | 7 | 1.09 | 1.10 | 1.01 | 3 | 1.56 | 1.41 | 0.90 | 1 | 1.10 | 1.21 | 1.10 | 3 |
| | **Total** | 1.05 | 1.06 | 1.01 | 3 | 1.68 | 1.29 | 0.77 | 1 | 1.13 | 1.22 | 1.08 | 3 |

| | | **Control** | | | | **SPELL** | | | | **Profiler** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Test** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** |
| $P_6$ | 1 | 0.97 | 0.93 | 0.95 | 6 | 1.43 | 1.40 | 0.98 | 1 | 0.23 | 0.17 | 0.75 | 6 |
| | 2 | 0.91 | 0.90 | 0.99 | 6 | 1.21 | 1.23 | 1.02 | 3 | 0.68 | 0.56 | 0.83 | 6 |
| | 3 | 0.91 | 0.91 | 0.99 | 6 | 1.32 | 1.32 | 1.00 | 3 | 0.38 | 0.29 | 0.76 | 6 |
| | 4 | 0.98 | 0.97 | 0.99 | 6 | 1.24 | 1.25 | 1.01 | 3 | 0.29 | 0.22 | 0.74 | 6 |
| | 5 | 0.89 | 0.88 | 1.00 | 6 | 1.22 | 1.28 | 1.05 | 3 | 0.73 | 0.65 | 0.89 | 6 |
| | 6 | 0.91 | 0.89 | 0.98 | 6 | 1.17 | 1.20 | 1.02 | 3 | 0.86 | 0.80 | 0.93 | 6 |
| | 7 | 0.98 | 0.96 | 0.98 | 6 | 1.31 | 1.27 | 0.97 | 1 | 0.27 | 0.19 | 0.69 | 6 |
| | **Total** | 0.94 | 0.93 | 0.98 | 6 | 1.28 | 1.28 | 1.01 | 3 | 0.35 | 0.26 | 0.74 | 6 |

| | | **Control** | | | | **SPELL** | | | | **Profiler** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Test** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** | **GU** | **SU** | **PU** | **CAT** |
| $P_{59}$ | 1 | 0.99 | 1.00 | 1.00 | 8 | 1.20 | 1.23 | 1.03 | 3 | 1.18 | 1.24 | 1.06 | 3 |
| | 2 | 1.13 | 1.04 | 0.92 | 1 | 1.45 | 1.16 | 0.80 | 1 | 1.22 | 1.15 | 0.94 | 1 |
| | 3 | 0.98 | 0.99 | 1.01 | 8 | 1.20 | 1.12 | 0.94 | 1 | 1.09 | 1.15 | 1.05 | 3 |
| | 4 | 1.01 | 1.01 | 1.00 | 1 | 1.07 | 1.12 | 1.05 | 3 | 1.11 | 1.18 | 1.06 | 3 |
| | 5 | 0.99 | 0.99 | 1.00 | 6 | 1.17 | 1.23 | 1.05 | 3 | 1.14 | 1.22 | 1.07 | 3 |
| | 6 | 1.08 | 1.04 | 0.96 | 1 | 1.16 | 1.16 | 0.99 | 1 | 1.20 | 1.17 | 0.98 | 1 |
| | 7 | 0.98 | 0.99 | 1.01 | 8 | 1.08 | 1.12 | 1.04 | 3 | 1.07 | 1.12 | 1.04 | 3 |
| | **Total** | 1.02 | 1.00 | 0.99 | 1 | 1.18 | 1.17 | 0.99 | 1 | 1.14 | 1.18 | 1.03 | 3 |

### 4.4.4 Discussion

To validate improvements and changes in energy consumption, we tested the following hypothesis:

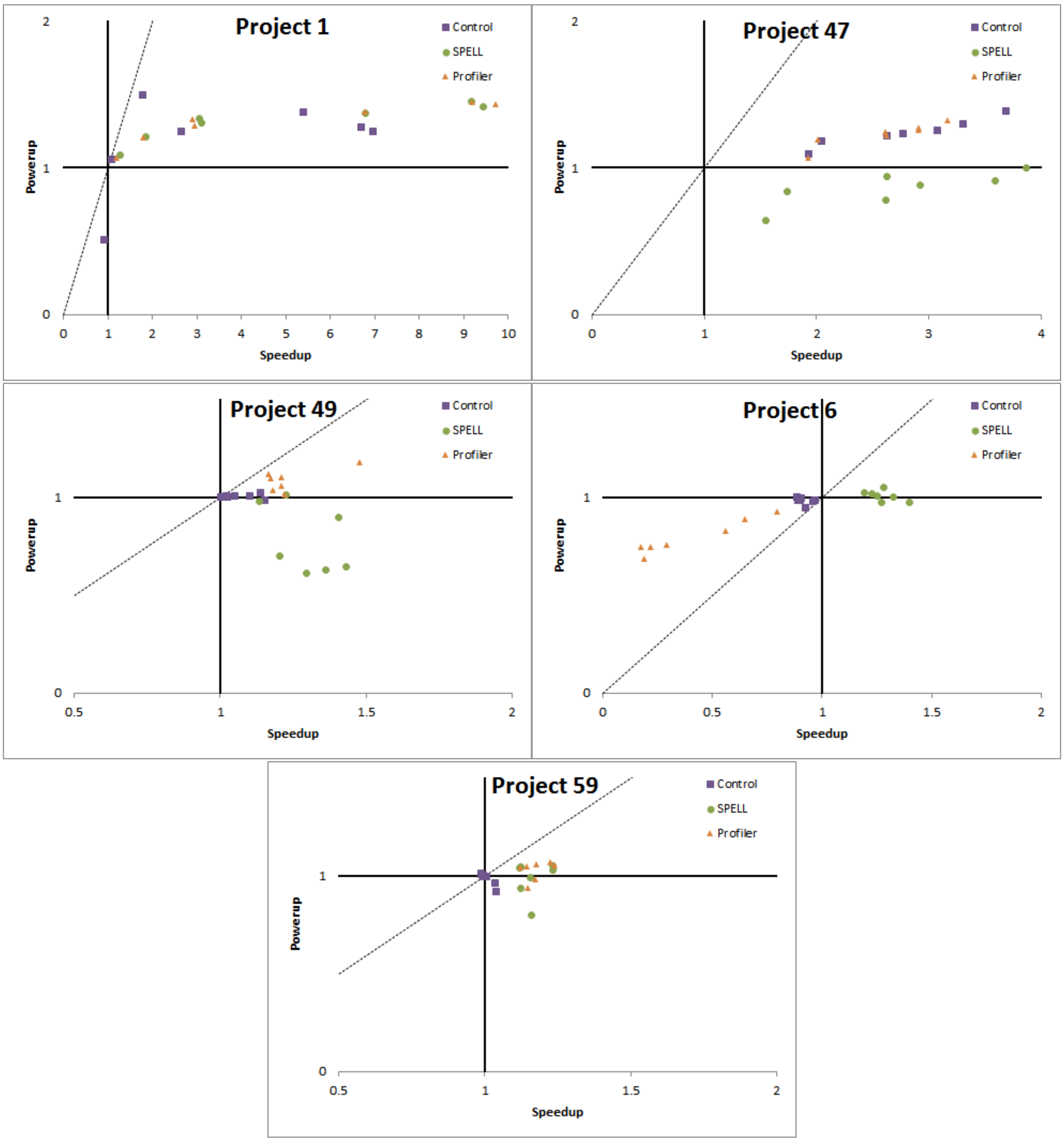


Figure 4.2: GPS-UP Software Energy Efficiency Quadrant Graph

$$H_0 : P(A > B) = 0.5$$

$$H_1 : P(A > B) \neq 0.5$$

where $P(A > B)$ represents, when we randomly draw from both A and B, that the probability of a draw from A is larger than the one from B is 50% in the case of our null hypothesis, and different than 50% in our alternative hypothesis. To understand if there is an overall significant relevance between the (A,B) distributions, and not only per test scenario or per project, the data from all 30 measured samples, 7 tests, and 5 projects were grouped per distribution (Original, Control, SPELL, and Profiler). The distributions were defined in the following (A, B) pairs: (Original, Control), (Original, SPELL), (Original, Profiler), (Control, SPELL), and (Profiler, SPELL). We consider the samples as independent, non-normal distributed, and ran the Wilcoxon signed-rank test with a two-tail P value with $\alpha$=0.01. The improvements were indeed very significant, producing significant relevance in all 5 cases, with p-values < 0.0001.

To calculate a nonparametric effect size, Field (Field, 2009) suggests using Rosenthal's formula (Rosenthal, 1991; Rosenthal et al., 1994) to compute a correlation, and compare the correlation values against Cohen's (Cohen, 1988) suggested thresholds of 0.1, 0.3, and 0.5 for small, medium, and large magnitudes respectively. Thus we obtained the values of: 0.4 (medium), 0.6 (large), 0.3 (medium), 0.6 (large), and 0.5 (large) for the respective 5 (A,B) pairs. Thus, we can see that SPELL outperforms the profiler when compared to both the original versions, where SPELL achieved a large effect size and the profiler a medium effects size, and to each other with also a large effect size.

Returning to our research questions, we have shown that there is both significant relevance and a large effect size when using our SPELL technique to improve the energy efficiency of programs, with an average energy gain of 44% (**RQ1**). While both the control-group (no tool assistance) and the profiler group did also produce significant relevance with their energy optimizations when compared to the original versions, SPELL outperformed both. Whereas the control group achieved a medium effect size, SPELL achieved a large effect size and when comparing SPELL to the control group, the former once again achieved a large effect size (**RQ2**). Finally, the same applies to the profiler group where it achieved a medium effect size when comparing the optimizations to the original version (versus the large effect size of SPELL), and again SPELL achieved a large effect size when comparing to the profiler group (**RQ3**).

***Observations*** From this study, we can see several interesting observations. In the case of Project $P_1$ and $P_{59}$, the rankings from both using SPELL and the profiler pointed to the same principal method (as shown in Table 4.3). Both were given a very high responsibility percentage (by SPELL) and high CPU time (by the profiler). This meant that if this method was optimized, a great impact in the performance would occur as this was, without a doubt, a very problematic method due to a bottleneck. Consequently for these two projects, the participants achieved very similar energy optimizations as one would expect. The slight difference can be attributed to what methods SPELL and the profiler pointed to afterwards, with the SPELL recommendations producing slightly better results.

We can also see how programmers with access to the SPELL recommendations were more efficient spending between 38%–57% less time, compared to the control-group, to detect and correct the problem, while also producing more efficient programs in both cases of energy and execution time. While those with the profile recommendations did also spend less time, they did not achieve results as good as those with the SPELL recommendation as we have seen. The participants also felt that having the ranking of responsibility percentage was very useful in identifying the *energy leaks* in the code, while the participants without this information expressed how they did not know where to start looking, or if certain parts were in fact problematic. All this is actually what we expected (for both SPELL and the profiler) as there is a substantial impact on having tools for energy-aware programming, as also suggested by (Pinto et al., 2014a; Pang et al., 2016).

Another interesting case is in Project $P_6$, where the results indicate a clear efficiency loss (both time and energy) for the case study using the profiler information. By comparing the original and transformed versions of the code, we discovered that the programmer responsible for this study decided to optimize the code by improving the efficiency of all listings and lookups on data structures, hence worsening insertions. The fact is that the feature tests that we provided contained more insertions than listings or lookups, leading to a decrease in the refactored version's performance. To understand if this outlier skewed our previous statistical analysis, we re-ran the analysis without considering Project $P_6$. The results maintained the same, with the only difference being the profiler obtained a slightly larger effect size when compared to the original projects. Thus, this does not change the conclusions of the study.

Looking at both the energy metrics and GPS-UP quadrant graphs, we can see how the optimizations following SPELL recommendations achieved on average lower *Powerup* values (implying average power savings). They also fell under *(Cat)egory* 1 more often (representing better performance and energy efficiency), while the optimizations performed with the profiler recommendations fell under *(Cat)egory* 3 (better performance at the cost of energy efficiency).

As the study only focused on giving participants one "round" or iteration of both the SPELL and profiler analysis, the participants using SPELL and the profiler tended to be "satisfied" with their optimizations much quicker, with time to spare in their maximum 2 hours scenario. In a real-world scenario, they would then run through another analysis, looking for new (if any) *energy leaks* and continue to further optimize if possible.

Finally, none of the participants had any knowledge of what techniques or optimizations could be done to specifically reduce energy consumption before going into the study. Nevertheless, with the knowledge on basic performance issues, algorithms, program complexity, and generally aiming for standard execution time optimization, they were able to achieve good results.

### 4.4.5 Looking back with DRAM

Initially, while SPELL allows various sources of energy measurements, our study only considered energy measurements from the CPU. This was due to the limitation in hardware which was available at the time. Since then, a new desktop system has been obtained which allows the energy consumption from both CPU and DRAM to be measured. As to both understand what impact the DRAM energy consumption would have had on our study, and also to validate the consistency of SPELL's analysis across different systems, we re-executed the initial stages of the study to calculate the global similarity.

The steps, and methodology are identical. The measurements were made on a new system, allowing RAPL measurements of the DRAM, with the following specifications: Linux Ubuntu Server 16.10 operating system, kernel version 4.8.0-22-generic, with 16GB of RAM, a Haswell Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz.

The global similarity results are shown in Table 4.7. The left-hand side are the results from

Table 4.7: SPELL with and without DRAM ranking of methods from Projects $P_1$, $P_{47}$, $P_{49}$, $P_6$, and $P_{59}$

| Proj. | Method (SPELL) | $\psi$ | $\psi$ | Method (SPELL w/ DRAM) |
|---|---|---|---|---|
| $P_1$ | voteInReport | 0.97 | 0.99 | voteInReport |
| | getUserLoggedInType | 0.02 | 0.00 | getUserLoggedInType |
| $P_{47}$ | listAllChronicles | 0.57 | 0.52 | listAllChronicles |
| | listAllReports | 0.15 | 0.16 | listAllReports |
| | chronicleExist | 0.12 | 0.1130 | Like |
| | Like | 0.078 | 0.1125 | chronicleExist |
| $P_{49}$ | Like | 0.27 | 0.35 | Like |
| | ListComments | 0.19 | 0.13 | ListComments |
| | AddComment | 0.10 | 0.09 | AddComment |
| | ListTopic | 0.08 | 0.07 | ListTopic |
| $P_6$ | printNoticiaTopicoTema | 0.40 | 0.40 | printNoticiaTopicoTema |
| | printCronicaTopicoTema | 0.20 | 0.22 | printCronicaTopicoTema |
| | isLogged | 0.15 | 0.17 | isLogged |
| $P_{59}$ | getArticle | 0.94 | 0.93 | getArticle |
| | vote | 0.04 | 0.05 | vote |

the original analysis (also shown in Table 4.3), and the right-hand side are the results from the SPELL analysis including DRAM energy consumption. In almost all cases, the rankings between both analyses maintained the same, with slight differences in the global similarity ($\psi$) value. The single exception can be observed in Project $P_{47}$. Here, the `Like` method came in fourth with a $\psi$ value of 0.078, while it came in third with a $\psi$ value of 0.1130 just slightly surpassing the `chronicleExist` method when DRAM energy consumption was also analyzed.

The initial hypothesis was that the results from the first analysis would not suffer any major changes in this case, as DRAM does not tend to have a high impact in overall energy consumption, as shown in Chapter 3 and other research (Melfe et al., 2018). Even so, this post-analysis shows how having more available information on the energy consumption of different hardware components (for example, DRAM) can bring about a deeper analysis, and such as in the case of Project $P_{47}$, can reveal more information on the problematic spots within one's application. The more components considered, the more accurate of an analysis can be performed by SPELL.

## 4.5 Threats to Validity

We present now some threats to validity of our study, divided in four categories as defined in (Cook and Campbell, 1979).

***Conclusion Validity*** From our experiment it is clear that we can effectively find energy *hot spots* in source code, both on a project level, and on a method level. Moreover, through the empirical study we have shown that these results are useful for programmers. Nevertheless, by energy consumption we only considered energy consumption that can be related to CPU usage due to our machine limitations. While we have shown that energy and performance are sometimes related in non-predictable ways, the impacts of other hardware components on energy consumption deserve further elaboration. Thus, we intend to explore this in the future by running a similar study on a machine with a more recent architecture.

***Internal Validity*** In this case we are concerned with other factors that may interfere with our experiment results. The energy consumption measurements we have for the different projects could have other factors than not just the source code itself. To avoid this we ran all the tests in the same way. For every test we added a “warm-up”, and we ran every test 30 times, taking the average values for these runs so we could minimize particular states of the machine used and its other software. Also, the results from participants may have been influenced by other factors other than the SPELL and profiler recommendations we gave them. However, the results achieved through the five projects are quite consistent.

***Construct Validity*** The purpose of our study was to evaluate our SPELL technique alongside programmers, to both properly understand the benefits of our technique with programmers, and to validate the efficiency of our technique in detecting energy leaks. Thus, we constructed an empirical study based off the suggestions of Ko et al. (Ko et al., 2015). For example, for the task duration, they suggest that the tasks should not be so easy as to have almost every participant complete them before the time expires (leading to *ceiling effects* (Rosenthal and Rosnow, 1984)), nor making it so difficult that no one can complete them in the allotted time no matter which tool is used (leading to *floor effects* (Rosenthal and Rosnow, 1984)). Both of these cases would make it difficult to statistically discriminate and show the differences between tools.

Due to this, and in addition to another suggestion that such studies should not be more than 2 hours long (Ko et al., 2015), we decided to use the academic Java projects we presented. This allowed us to have projects which were neither too difficult nor too easy to both understand and optimize within our established time limit. Using larger real-world applications would introduce a risk of participants not completing or understanding (possible *floor effects*) due to the complexity and possible lack of domain documentation. Nevertheless, there is no basis to suspect that these projects are better or worse than any other kind we could have used.

***External Validity*** In this case we are concerned about the generalization of the results. The used source code has no particular characteristics that could influence our findings. Its only particularity is that it is written in Java, and maybe different results could be found for other PLs. However, our technique is independent of the language and thus we do not anticipate that. Thus, we believe that these results can be further generalized for other programs.

## 4.6 Conclusions

This chapter introduced SPELL — a spectrum-based energy leak localization technique to identify inefficient energy consumption in the source code of software systems. This technique uses a statistical method to associate different percentage of responsibility for the energy consumed to the different source code components of a software system, thus pinpointing the developer's attention on the most critical *red* spots in his code. Such software components may not only be source code fragments, but also a set of equivalent software systems from which we need to select the greenest one.

The chapter also presented the implementation of this technique as a language independent tool to locate energy leaks in a program's source code. A front-end for the Java language was constructed to monitor energy consumption at runtime, which uses the developed SPELL tool to locate leaks in Java.

To evaluate both our technique and tool, we executed an empirical study where we asked five groups of three developers to optimize the energy efficiency of a software system. One developer had no tool assistance, while the other two used our SPELL technique and a profiler, respectively. We showed that developers using our technique were able to improve the energy efficiency of their programs by 43% on average, while also showing statistical evidence that the difference between a profiler and our technique is significant, in favor of the former: the performance is between 2% and 72% better.

Thus, we have also shown that optimizing for energy efficiency is not directly the same as optimizing for performance. We also showed that using our technique, the performed optimizations achieved on average a lower *Powerup* (implying average power savings, with better performance and energy efficiency), while optimizations following a profiler's recommendations achieved better performance at the cost of energy efficiency.

# Chapter 5

# Energy Efficiency Across Java Collections

*This chapter presents a detailed study of the energy consumption of the different Java Collection Framework (JFC) implementations. For each method of an implementation in this framework, we present its energy consumption when handling different amounts of data. Knowing the greenest methods for each implementation, we present an energy optimization approach for Java programs: based on calls to JFC methods in the source code of a program, we select the greenest implementation. We present preliminary results of optimizing a set of Java programs where we obtained* 6.2% *energy savings.*

*Finally, we present jStanley, a tool which automatically finds collections in Java programs which can be replaced by others with a positive impact on the energy consumption. In seconds, developers obtain information about energy-eager collection usage. jStanley will further suggest alternative collections to improve the code, making it use less time, energy, or a combination of both. A second preliminary evaluation using jStanley shows energy gains between 2% - 17%, and a reduction in execution time between 2% - 13%.*

## 5.1 Introduction

In addition to the chosen programming language influencing a program's energy consumption, other factors also play part, such as the used algorithms or libraries. Programming languages often times include extensive and wildly used libraries, some of which are very popular and almost core to the system, such as Java's Collections Framework (JCF).

To properly and fully understand the energy impact of Java collections and their methods, this chapter focuses on answering **TRQ3**. For this, we conduct a detailed study in terms of energy consumption of the widely used JCF library [1]. We consider three different groups of data structures, namely Sets, Lists, and Maps, and for each of these groups, we study the energy consumption of each of its different implementations and methods. We exercise and monitor the energy consumed by each of the API methods when handling low, medium and big data sets.

A first result of our study is a quantification of the energy spent by each method of each implementation, for each of the data structures we consider. This energy-awareness can not only be used to steer software developers in writing greener software, but also in optimizing legacy code. In fact, we have used/validated this quantification by semi-automatically optimizing the energy consumption of a set of similar software systems.

As a second result, we statically compute which implementations and methods are used in the source code of such projects, and then look up the energy consumption data to find which equivalent implementation has the lowest energy consumption for those specific methods. Finally, we manually transform the source code to use the "greenest" implementation. Our preliminary results show that energy consumption decreased in all the optimized software systems that we tested, with an average energy saving of 6.2%.

With our work we are answering the following research questions:

- **RQ1** - *Can we define an energy consumption quantification of Java data structures and their methods?* This research question focuses on answering if one is actually able to quantify the energy consumption of data structures and their methods. Being able

[1] `docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html`

to precisely measure the energy consumption on such a small level (for example a Map.put() method) is many times restricted to how well the measurement tool can reach, or in other words the granularity of the energy measurement. To perform such a task, not only do we need a good measuring tool, but also a benchmarking system ready to tackle such a problem.

- **RQ2** - *Can we use such quantification to decrease the energy consumption of software systems?* After obtaining a quantification of the energy consumption, understanding how to actually use that information to improve the energy consumption of a software program can be very tricky. There are many differences between each data structure and the energy spent by each method (even if they are similar), due to the nature of the actual data structure. Such structural differences, and all the various possible scenarios one may implement a program can affect which is the best alternative data structure depending on what to be executed.

Finally, we detail *jStanley*, a static analysis tool which suggests a more energy efficient (and/or performance efficient) Java collection, by statically detecting collections used in a Java project, and which methods are used for each collection. Using this information, it not only suggests a better alternative, but can automatically change the code with the new collections if the programmer chooses so. However, it is not possible to always guarantee the change to best collection is a refactoring as some collections for instance change the order of the elements.

This chapter is organized as follows: Section 5.2 contains our analysis of the energy consumption of the different Java Collection Framework implementations. In Section 5.3 we describe our methodology to optimize Java programs and its application to five Java programs. Section 5.4 presents the jStanely tool based on the research in the prior sections. Section 5.5 presents the validity threats for our analysis. Finally, we present our conclusions in Section 5.6.

## 5.2 Ranking of Java Implementation's Methods

One of our goals is to compare the energy consumption of different Java implementations of the same abstract data structures. To do this, we designed an experiment that simulates different kinds of uses of such structures. In this section we present the design, execution, and results of that simulation.

### 5.2.1 Design

Our experiment design is inspired by the one used in (Manotas et al., 2014), since our analysis also considers a simple scenario of storing, retrieving, and deleting String values in the various collections.

**JCF Data structures** The most classical way to separate Java data structures is into groups which implement the interfaces Set[2], List[3], or Map[4], respectively. This separation indeed makes sense as each interface has its own distinct properties and purposes (for example, there is no ordering notion under Sets).

In our study, a few implementations were not evaluated as they are quite particular in their usage and could not be populated with strings. In particular, *JobStateReasons* (Set) only accepts *JobStateReason* objects, *IdentityHashMap* (Map) accepts strings but compares its elements with the identity function, and not with the *equals* method.

Given these considerations, we evaluated the following implementations:

**Sets** *ConcurrentSkipListSet, CopyOnWriteArraySet, HashSet, LinkedHashSet, TreeSet*

**Lists** *ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector*

**Maps** *ConcurrentHashMap, ConcurrentSkipListMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, Properties, SimpleBindings, TreeMap, UIDefaults, WeakHashMap*

[2] docs.oracle.com/javase/7/docs/api/java/util/Set.html

[3] docs.oracle.com/javase/7/docs/api/java/util/List.html

[4] docs.oracle.com/javase/7/docs/api/java/util/Map.html

**Methods** To choose the methods to measure for each abstraction, we looked at the generic API list for the corresponding interface.

From this list, we chose the methods which performed insertion, removal, or searching operations on the data structures, along with a method to iterate and consult all the values in the structure. In some methods (e.g. containsAll or addAll), a second data structure is needed.

**Sets** *add, addAll, clear, contains, containsAll, iterateAll, iterator, remove, removeAll, retainAll, toArray*

**Lists** *add, addAll, add (at an index), addAll (at an index), clear, contains, containsAll, get, indexOf, iterator, lastIndexOf, listIterator, listIterator (at an index), remove, removeAll, remove (at an index), retainAll, set, sublist, and toArray*

**Maps** *clear, containsKey, containsValue, entrySet, get, iterateAll, keySet, put, putAll, remove, and values*

**Benchmark** To evaluate the different implementations on each of the described methods, we started by creating and populating objects with different sizes for each implementation. [5]

We considered initial objects with 25.000, 250.000, and 1.000.000 elements, providing our analysis with multiple orders of magnitude of measurement. This will allow us to better understand how the energy consumption scales in regards to population size.

When a second data structure is required, we have adopted for it a size[6] of 10% the *popsize* of the tested structure, containing half existing values from the tested structure and half new values, all shuffled.

Table 5.1 briefly summarizes how each method is tested for the Set collection, with the tests for the other collections being similar.

### 5.2.2 Execution

To analyze the energy consumption, we first implemented our data structure analysis design as an energy benchmark framework. This is one of our contributions, and can be found at `github.com/greensoftwarelab/Collections-Energy-Benchmark`. This implementation

[5] We will refer to the population size of an object as *popsize*.

[6] We will refer to the size of each such structure as *secondaryCol*.

Table 5.1: Test description of Set methods

| Method | Description of the test for the method |
|---|---|
| add | add popsize/10 elements. half existing, half new |
| addAll | addAll of secondaryCol 5 times |
| clear | clear 5 times |
| contains | contains popsize/10 elements. half existing, half new |
| containsAll | containsAll of secondaryCol 5 times |
| iterateAll | iterate and consult popsize values |
| iterator | iterator popsize times |
| remove | remove popsize/10 elements. half existing, half new |
| removeAll | removeAll of secondaryCol 5 times |
| retainAll | retainAll of secondaryCol 5 times |
| toArray | toArray 5 times |

is based on a publicly available micro-benchmark[7] which evaluates the runtime performance of different implementations of the Collections API, and has been used in a previous study to obtain energy measurements (Manotas et al., 2014).

To allow us to record precise energy consumption measurements from the CPU, we used Intel's Runtime Average Power Limit (RAPL) (David et al., 2010). RAPL is an interface which allows access to energy and power readings via a model-specific register. Its precision and reliability has been extensively studied (Hähnel et al., 2012; Rotem et al., 2012). More specifically, we used jRAPL (Liu et al., 2015) which is a framework for profiling Java programs using RAPL. Using these tools permitted us to obtain energy measurements on a method level, allowing us a fine grained measurement.

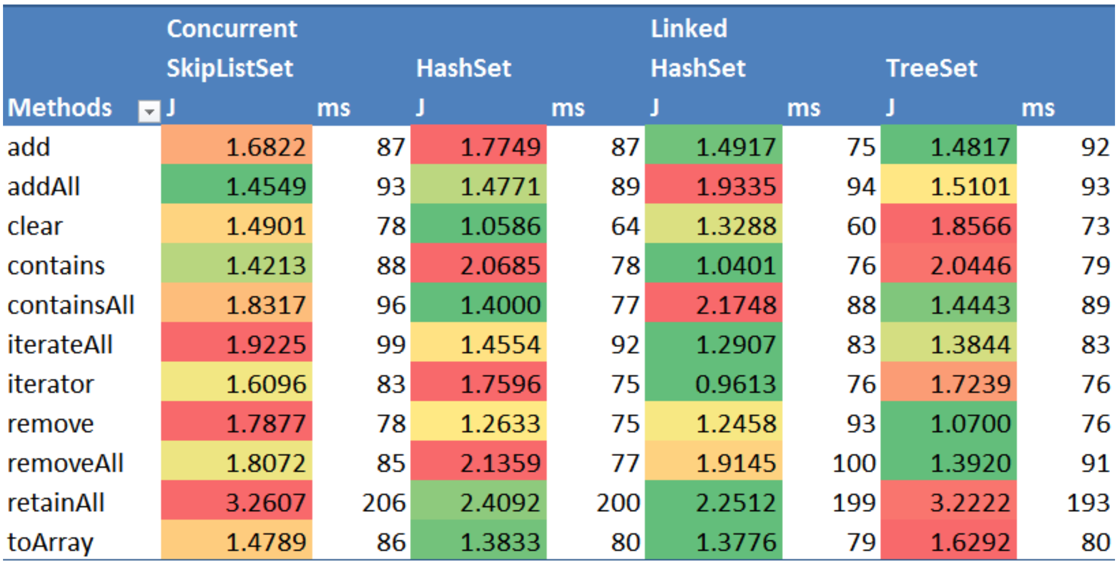
We ran this study on a server with the following specifications: Linux 3.13.0-74-generic operating system, 8GB of RAM, and Intel(R) Core(TM) i3-3240 CPU @ 3.40GHz. This system has no other software installed or running other than necessary to run this study, and the operating system daemons. Both the Java compiler and interpreter were versions 1.8.0_66.

Prior to executing a test, we ran an initial "warm-up" where we instantiated, populated (with the designated *popsize*), and performed simple actions on the data structures. Each test was executed 10 times, and the average values for both the time and energetic consumption were extracted (of the specific test, and not the initial "warm-up" as to only measure the tested methods) after removing the lowest and highest 20% as to limit outliers.

[7] dzone.com/articles/java-collection-performance

### 5.2.3 Results

This section presents the results we gathered from the experiment. We highly recommend and assume the images are being viewed in color. Figures 5.1, 5.2, and 5.3 represent the data for our analyzed Sets, Lists, and Maps respectively, for a population size of 25k elements. Each row in the tables represents the measured methods, and for each analyzed implementation, we have two columns representing the consumption in Joules (*J*) and execution time in milliseconds(*ms*). Each row has a color highlight (under the *J* columns) varying between a Red to Yellow to Green. The most energetically inefficient implementation for that row's method (the one with the highest consumed Joules) is highlighted Red. The implementation with the lowest consumed Joules (most energetically efficient) is highlighted Green. The rest are highlighted depending on their consumption values when compared to the most inefficient and efficient implementation, and colored accordingly in the color scale.

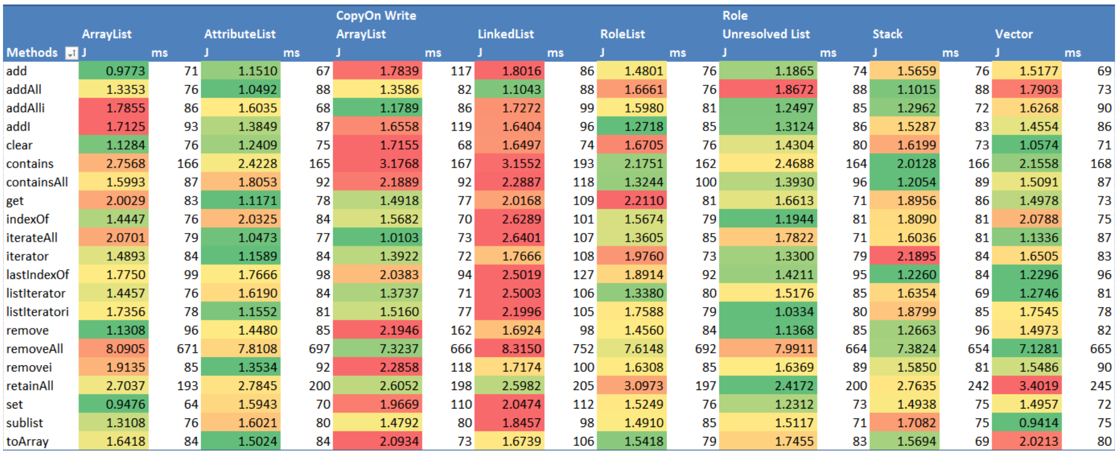


| | Concurrent SkipListSet | | HashSet | | Linked HashSet | | TreeSet | |
|---|---|---|---|---|---|---|---|---|
| Methods | J | ms | J | ms | J | ms | J | ms |
| add | 1.6822 | 87 | 1.7749 | 87 | 1.4917 | 75 | 1.4817 | 92 |
| addAll | 1.4549 | 93 | 1.4771 | 89 | 1.9335 | 94 | 1.5101 | 93 |
| clear | 1.4901 | 78 | 1.0586 | 64 | 1.3288 | 60 | 1.8566 | 73 |
| contains | 1.4213 | 88 | 2.0685 | 78 | 1.0401 | 76 | 2.0446 | 79 |
| containsAll | 1.8317 | 96 | 1.4000 | 77 | 2.1748 | 88 | 1.4443 | 89 |
| iterateAll | 1.9225 | 99 | 1.4554 | 92 | 1.2907 | 83 | 1.3844 | 83 |
| iterator | 1.6096 | 83 | 1.7596 | 75 | 0.9613 | 76 | 1.7239 | 76 |
| remove | 1.7877 | 78 | 1.2633 | 75 | 1.2458 | 93 | 1.0700 | 76 |
| removeAll | 1.8072 | 85 | 2.1359 | 77 | 1.9145 | 100 | 1.3920 | 91 |
| retainAll | 3.2607 | 206 | 2.4092 | 200 | 2.2512 | 199 | 3.2222 | 193 |
| toArray | 1.4789 | 86 | 1.3833 | 80 | 1.3776 | 79 | 1.6292 | 80 |

Figure 5.1: Set results for population of 25k

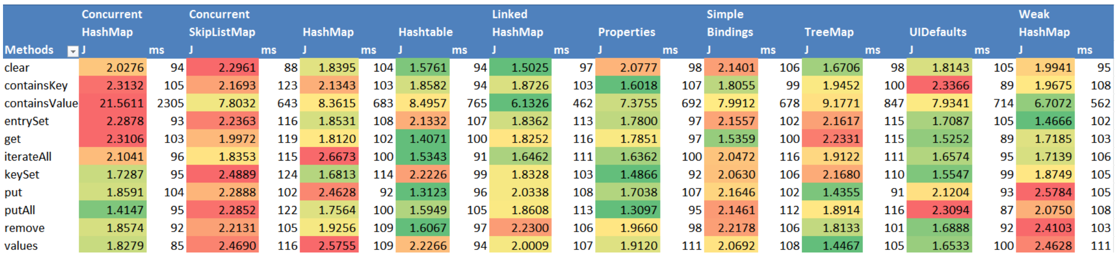
Figures 5.4 is a graphical representation of the data for our analyzed Sets. The Y-Axis represents the consumption in Joules, and the X-Axis represents the various measured methods. Each column represents a specific analyzed implementation.

The *CopyOnWriteArraySet* implementation was discarded during the experiment execu-



| | ArrayList | | AttributeList | | CopyOn Write ArrayList | | LinkedList | | RoleList | | Role Unresolved List | | Stack | | Vector | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Methods | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms |
| add | 0.9773 | 71 | 1.1510 | 67 | 1.7839 | 117 | 1.8016 | 86 | 1.4801 | 76 | 1.1865 | 74 | 1.5659 | 76 | 1.5177 | 69 |
| addAll | 1.3353 | 76 | 1.0492 | 88 | 1.3586 | 82 | 1.1043 | 88 | 1.6661 | 76 | 1.8672 | 88 | 1.1015 | 88 | 1.7903 | 73 |
| addAlli | 1.7855 | 86 | 1.6035 | 68 | 1.1789 | 86 | 1.7272 | 99 | 1.5980 | 81 | 1.2497 | 85 | 1.2962 | 72 | 1.6268 | 90 |
| addI | 1.7125 | 93 | 1.3849 | 87 | 1.6558 | 119 | 1.6404 | 96 | 1.2718 | 85 | 1.3124 | 86 | 1.5287 | 83 | 1.4554 | 86 |
| clear | 1.1284 | 76 | 1.2409 | 75 | 1.7155 | 68 | 1.6497 | 74 | 1.6705 | 76 | 1.4304 | 80 | 1.6199 | 73 | 1.0574 | 71 |
| contains | 2.7568 | 166 | 2.4228 | 165 | 3.1768 | 167 | 3.1552 | 193 | 2.1751 | 162 | 2.4688 | 164 | 2.0128 | 166 | 2.1558 | 168 |
| containsAll | 1.5993 | 87 | 1.8053 | 92 | 2.1889 | 92 | 2.2887 | 118 | 1.3244 | 100 | 1.3930 | 96 | 1.2054 | 89 | 1.5091 | 87 |
| get | 2.0029 | 83 | 1.1171 | 78 | 1.4918 | 77 | 2.0168 | 109 | 2.2110 | 81 | 1.6613 | 71 | 1.8956 | 86 | 1.4978 | 73 |
| indexOf | 1.4447 | 76 | 2.0325 | 84 | 1.5682 | 70 | 2.6289 | 101 | 1.5674 | 79 | 1.1944 | 81 | 1.8090 | 81 | 2.0788 | 75 |
| iterateAll | 2.0701 | 79 | 1.0473 | 77 | 1.0103 | 73 | 2.6401 | 107 | 1.3605 | 85 | 1.7822 | 71 | 1.6036 | 81 | 1.1336 | 87 |
| iterator | 1.4893 | 84 | 1.1589 | 84 | 1.3922 | 72 | 1.7666 | 108 | 1.9760 | 73 | 1.3300 | 79 | 2.1895 | 84 | 1.6505 | 83 |
| lastIndexOf | 1.7750 | 99 | 1.7666 | 98 | 2.0383 | 94 | 2.5019 | 127 | 1.8914 | 92 | 1.4211 | 95 | 1.2260 | 84 | 1.2296 | 96 |
| listIterator | 1.4457 | 76 | 1.6190 | 84 | 1.3737 | 71 | 2.5003 | 106 | 1.3380 | 80 | 1.5176 | 85 | 1.6354 | 69 | 1.2746 | 81 |
| listIteratori | 1.7356 | 78 | 1.1552 | 81 | 1.5160 | 77 | 2.1996 | 105 | 1.7588 | 79 | 1.0334 | 80 | 1.8799 | 85 | 1.7545 | 78 |
| remove | 1.1308 | 96 | 1.4480 | 85 | 2.1946 | 162 | 1.6924 | 98 | 1.4560 | 84 | 1.1368 | 85 | 1.2663 | 96 | 1.4973 | 82 |
| removeAll | 8.0905 | 671 | 7.8108 | 697 | 7.3237 | 666 | 8.3150 | 752 | 7.6148 | 692 | 7.9911 | 664 | 7.3824 | 654 | 7.1281 | 665 |
| removei | 1.9135 | 85 | 1.3534 | 92 | 2.2858 | 118 | 1.7174 | 100 | 1.6308 | 85 | 1.6369 | 89 | 1.5850 | 81 | 1.5486 | 90 |
| retainAll | 2.7037 | 193 | 2.7845 | 200 | 2.6052 | 198 | 2.5982 | 205 | 3.0973 | 197 | 2.4172 | 200 | 2.7635 | 242 | 3.4019 | 245 |
| set | 0.9476 | 64 | 1.5943 | 70 | 1.9669 | 110 | 2.0474 | 112 | 1.5249 | 76 | 1.2312 | 73 | 1.4938 | 75 | 1.4957 | 72 |
| sublist | 1.3108 | 76 | 1.6021 | 80 | 1.4792 | 80 | 1.8457 | 98 | 1.4910 | 85 | 1.5117 | 71 | 1.7082 | 75 | 0.9414 | 75 |
| toArray | 1.6418 | 84 | 1.5024 | 84 | 2.0934 | 73 | 1.6739 | 106 | 1.5418 | 79 | 1.7455 | 83 | 1.5694 | 69 | 2.0213 | 80 |

Figure 5.2: List results for population of 25k



| | Concurrent HashMap | | Concurrent SkipListMap | | HashMap | | Hashtable | | Linked HashMap | | Properties | | Simple Bindings | | TreeMap | | UIDefaults | | Weak HashMap | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Methods | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms |
| clear | 2.0276 | 94 | 2.2961 | 88 | 1.8395 | 104 | 1.5761 | 94 | 1.5025 | 97 | 2.0777 | 98 | 2.1401 | 106 | 1.6706 | 98 | 1.8143 | 105 | 1.9941 | 95 |
| containsKey | 2.3132 | 105 | 2.1693 | 123 | 2.1343 | 103 | 1.8582 | 94 | 1.8726 | 103 | 1.6018 | 107 | 1.8055 | 99 | 1.9452 | 100 | 2.3366 | 89 | 1.9675 | 108 |
| containsValue | 21.5611 | 2305 | 7.8032 | 643 | 8.3615 | 683 | 8.4957 | 765 | 6.1326 | 462 | 7.3755 | 692 | 7.9912 | 678 | 9.1771 | 847 | 7.9341 | 714 | 6.7072 | 562 |
| entrySet | 2.2878 | 93 | 2.2363 | 116 | 1.8531 | 108 | 2.1332 | 107 | 1.8362 | 113 | 1.7800 | 97 | 2.1557 | 102 | 2.1617 | 115 | 1.7087 | 105 | 1.4666 | 102 |
| get | 2.3106 | 103 | 1.9972 | 119 | 1.8120 | 102 | 1.4071 | 100 | 1.8252 | 116 | 1.7851 | 97 | 1.5359 | 100 | 2.2331 | 115 | 1.5252 | 89 | 1.7185 | 103 |
| iterateAll | 2.1041 | 96 | 1.8353 | 115 | 2.6673 | 100 | 1.5343 | 91 | 1.6462 | 111 | 1.6362 | 100 | 2.0472 | 116 | 1.9122 | 111 | 1.6574 | 95 | 1.7139 | 106 |
| keySet | 1.7287 | 95 | 2.4889 | 124 | 1.6813 | 114 | 2.2226 | 99 | 1.8328 | 103 | 1.4866 | 92 | 2.0630 | 106 | 2.1680 | 110 | 1.5547 | 99 | 1.8749 | 105 |
| put | 1.8591 | 104 | 2.2888 | 102 | 2.4628 | 92 | 1.3123 | 96 | 2.0338 | 108 | 1.7038 | 107 | 2.1646 | 102 | 1.4355 | 91 | 2.1204 | 93 | 2.5784 | 105 |
| putAll | 1.4147 | 95 | 2.2852 | 122 | 1.7564 | 100 | 1.5949 | 105 | 1.8608 | 113 | 1.3097 | 95 | 2.1461 | 112 | 1.8914 | 116 | 2.3094 | 87 | 2.0750 | 108 |
| remove | 1.8574 | 92 | 2.2131 | 105 | 1.9256 | 109 | 1.6067 | 97 | 2.2300 | 106 | 1.9660 | 98 | 2.2178 | 106 | 1.8133 | 101 | 1.6888 | 92 | 2.4103 | 103 |
| values | 1.8279 | 85 | 2.4690 | 116 | 2.5755 | 109 | 2.2266 | 94 | 2.0009 | 107 | 1.9120 | 111 | 2.0692 | 108 | 1.4467 | 105 | 1.6533 | 100 | 2.4628 | 111 |

Figure 5.3: Map results for population of 25k

tion as the tests did not finish in a reasonable amount of time. For the full representation of the data/graphs of the other two population sizes and omitted data, please consult the online appendix[8], the online interactive data tables[9] or this document's Chapter 5 appendix (Appendix B). From our data, we can draw interesting observations:

- Looking at the Set results for population of 25k data (shown in Fig 5.1) we can see that *LinkedHashSet* includes most of the energetically efficient methods. Nevertheless, one can easily notice that it is also the most inefficient with the *addAll* and *containsAll* methods.

[8] Online Appendix for Chapter 5:
`http://greenlab.di.uminho.pt/wp-content/uploads/2016/06/appendixGreens.pdf`

[9] Interactive Data Tables:
`http://greenlab.di.uminho.pt/collections/`

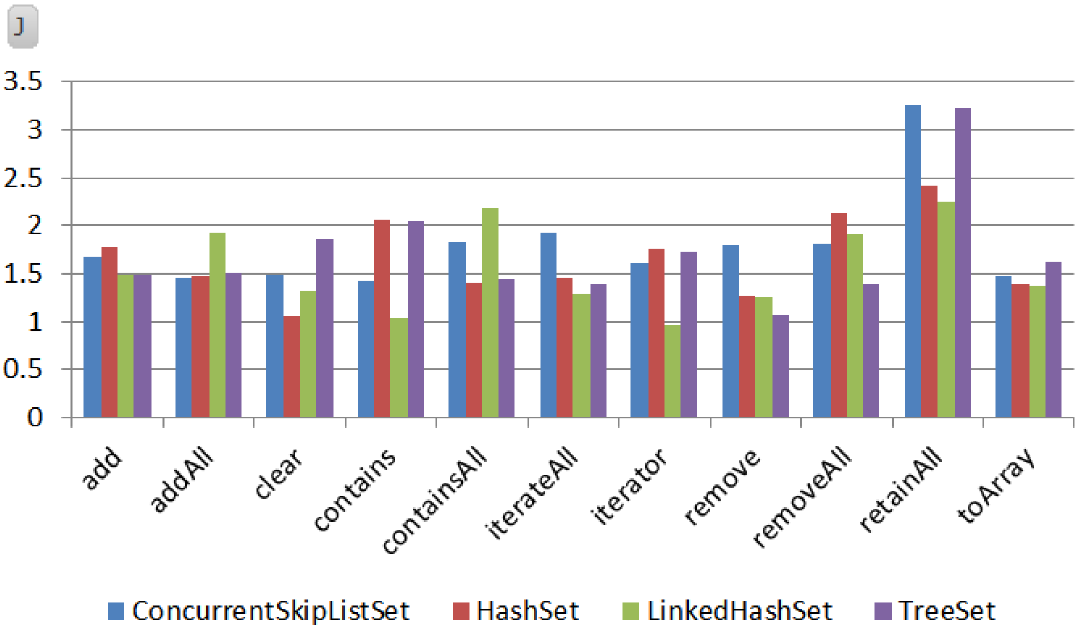


Figure 5.4: Set results graph for population of 25k

- Figure 5.2 presents the List results for population of 25k. Both *RoleUnresolvedList* and *AttributeList* contain the most efficient methods. Interesting to point out that both of these extend *ArrayList*, which contains less efficient methods, and very different consumption values in comparison with these two. We can also clearly see that *LinkedList* is by far the most inefficient List implementation.

- In Figure 5.3, we can see that *Hashtable*, *LinkedHashMap*, and *Properties* contain the most efficient methods, and with no red methods. Interesting to note is that while the Properties data structure is generally used to store project configuration data/settings, it produced very good results for our scenario of storing string values.

- The concurrent data structure implementations (*ConcurrentSkipListSet, CopyOnWriteArrayList, ConcurrentHashMap, ConcurrentSkipListMap,* and the removed *CopyOnWriteArraySet*) perform very poorly. As such, these should probably be avoided if a requirement is a low consuming application.

- One can see cases where a decrease in execution time translates into a decrease in the energy consumed as suggested by (Yuki and Rajopadhye, 2014). For instance in Figure 5.3, when comparing *Hashtable* and *TreeSet* for the get method, we see that *Hashtable* has both a lower execution time and energy consumption. As observed by (Trefethen and Thiyagalingam, 2013b; Pinto et al., 2014b), cases where an increase in execution time brings about a decrease in the energy consumed can also be seen, for example in Figure 5.3 when comparing *HashMap* and *Hashtable* for the keySet method. As such, we cannot draw any conclusion of the correlation between execution time and energy being consumed.

- Different conclusions can be drawn for the 250k and 1m population sizes (which can be seen in our appendices). This also shows that the energy consumption of different data structure implementations scale differently in regards to size. What may be the most efficient implementation for one population size, may not be the best for another.

## 5.3 Optimizing Energy Consumption of Java Programs

The results presented in the previous section may allow software developers to develop more energy efficient software. In this section we present a methodology to optimize, at compilation time, existing Java programs. This methodology consists of the following steps:

1. *Computing which implementation/methods are used in the programs*
2. *Looking up the appropriate energy tables for the used implementation/methods*
3. *Choosing the most efficient implementation based on total energy*

In the next subsection, we describe in detail how we applied this approach and how it was used to optimize a set of equivalent Java programs.

### 5.3.1 Data Acquisition

First, we obtained several Java projects from an object-oriented course for undergraduate computer science students. For this course, students were asked to build a journalism support platform, where users (Collaborators, Journalists, Readers, and Editors) can write articles (chronicles and reports), and give likes and comments. Along with these different platform implementations, we obtained seven test cases which simulated using the system (registering, logging in, writing articles, commenting, etc.). The size of users, articles, and comments varied between approximately 2.000 and 10.000 each for each different test case and each entity. These projects had an average of 36 classes, 104 methods, and 2.000 lines of code.

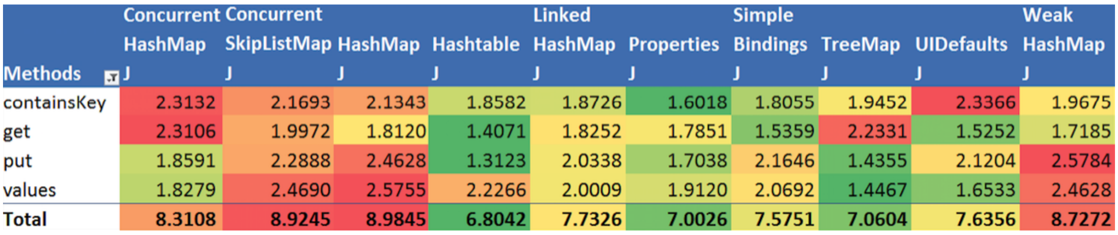
Next we discuss the optimization of five of those projects, where we semi-automatically detected the use of any JCF implementation (both efficient and inefficient implementations), and which were the used methods for each implementation.

### 5.3.2 Choosing an energy efficient alternative

To try to optimize these projects based on the data structures and their used methods, we looked at our data for the 25k population. We chose this one, as it is the one which is closest to the population used in the test cases (which was between 2.000 and 10.000 for each different entity).

For each detected data structure implementation, we selected the used methods, and chose our optimized data structure based on the implementation which consumed the least amount of energy for this specific case.

Figure 5.5 shows the data used to make our decision for the Maps of Project 1, where *Hashtable* was used in place of *TreeMap* (as *Hashtable* was the most efficient implementation in this scenario with 6.8J). Table 5.2 details the 5 Projects, their originally used data structure implementations, new implementation, and used methods for the implementations.



| Methods | Concurrent HashMap J | Concurrent SkipListMap J | HashMap J | Hashtable J | Linked HashMap J | Properties J | Simple Bindings J | TreeMap J | UIDefaults J | Weak HashMap J |
|---|---|---|---|---|---|---|---|---|---|---|
| containsKey | 2.3132 | 2.1693 | 2.1343 | 1.8582 | 1.8726 | 1.6018 | 1.8055 | 1.9452 | 2.3366 | 1.9675 |
| get | 2.3106 | 1.9972 | 1.8120 | 1.4071 | 1.8252 | 1.7851 | 1.5359 | 2.2331 | 1.5252 | 1.7185 |
| put | 1.8591 | 2.2888 | 2.4628 | 1.3123 | 2.0338 | 1.7038 | 2.1646 | 1.4355 | 2.1204 | 2.5784 |
| values | 1.8279 | 2.4690 | 2.5755 | 2.2266 | 2.0009 | 1.9120 | 2.0692 | 1.4467 | 1.6533 | 2.4628 |
| **Total** | **8.3108** | **8.9245** | **8.9845** | **6.8042** | **7.7326** | **7.0026** | **7.5751** | **7.0604** | **7.6356** | **8.7272** |

Figure 5.5: Choosing optimized Map for Project 1

Table 5.2: Original and optimized data structures, and used methods for each project

| | **Data Structures** | | |
|---|---|---|---|
| **Projects** | **Original** | **Optimized** | **Methods** |
| 1 | TreeMap<br>LinkedList | Hashtable<br>ArrayList | {containsKey, get, put, values}<br>{add, listIterator} |
| 2 | HashMap | Hashtable | {containsKey, get, put, values} |
| 3 | LinkedList | ArrayList | {add, addAll, iterator, listIterator, remove} |
| 4 | LinkedList<br>HashMap | AttributeList<br>Hashtable | {add (at an index), iterator}<br>{containsKey, get, put} |
| 5 | HashMap<br>TreeSet | Hashtable<br>LinkedHastSet | {containsKey, get, put}<br>{add, iterator} |

### 5.3.3 Pre-energy measurement setup

Now that we have chosen our energy efficient alternative, we need to change the projects to reflect this. The source code was manually altered to use the chosen implementations. Finally, we verified that the program maintained the original consistency and state by verifying if the outputs and operations produced by these two versions did not change.

### 5.3.4 Energy measurements

To measure the original, and optimized projects, we followed the same methodology detailed in Section: 5.2.2 *Execution*. We executed the seven test cases in the same server, and using jRAPL obtained the energy consumption measurements. Each test was also executed 10 times, and the average values (after removing the 20% highest and lowest values) were calculated.

### 5.3.5 Results

Table 5.3 presents, for each project, the energy consumption in Joules (J), and execution time in milliseconds (ms) for both the original and optimized implementations. The last column shows the improvement gained after having performed the optimized implementations for both the consumption and execution time.

As we can see, all five programs improve their energy efficiency. Our optimization improves their energy consumption between 4.37% and 11.05%, with an average of 6.2%.

Table 5.3: Results of pre and post optimization

| **Projects** | **Data Structures**<br>**Original** | | **Optimized** | | **Improvement** | |
|---|---|---|---|---|---|---|
| | J | ms | J | ms | J | ms |
| 1 | 23.744583 | 1549 | 22.7071302 | 1523 | 4.37% | 1.68% |
| 2 | 24.6787895 | 1823 | 23.525123 | 1741 | 4.67% | 4.50% |
| 3 | 25.0243507 | 1720 | 22.259355 | 1508 | 11.05% | 12.33% |
| 4 | 17.1994425 | 1258 | 16.2014997 | 1217 | 5.80% | 3.26% |
| 5 | 19.314512 | 1372 | 18.3067573 | 1245 | 5.22% | 9.26% |

## 5.4 jStanely

*jStanley*[10] is a static analyzer developed as an Eclipse plug-in since this is the most used IDE for Java. The tool we propose is capable of statically detecting the usage of energy inefficient collections and suggest better alternatives. It can do the same, but considering the execution time or both energy and time at the same time. To this end, it uses information on the energy consumption and execution time for Java collections shown in the prior section. This information can be provided to the tool through a set of CSV files, one per type of collection (map, list and set). Each file must contain information about the energy and time usage for each method of the collection.

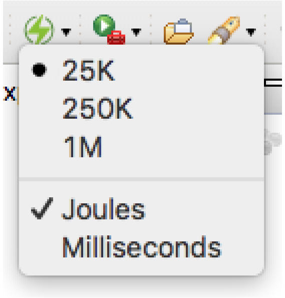
*jStanley* constructs the abstract syntax tree (AST) of the program being analyzed, traversing it to compute the number of method calls of a given collection variable. As a result, the tool knows how many method calls for each variable of a collection the program has. For instance, for a given program, the tool can tell variable `a` of type `ArrayList` has 3 calls to the method `add` and 9 to `get`, and variable `b` of type `HashMap` has 20 calls to `put`, throughout the whole program.

Our tool provides a drop down menu within Eclipse, as shown in Figure 5.6, allowing programmers to select their preferences. This menu displays options to choose if they wish to focus on energy, execution time, or both, by choosing Joules, Milliseconds, or both, respectively. Additionally, as we have seen in our previous study, different population sizes bring about different energy profiles for collections. Thus, there is also an option to allow the pro-

[10] *jStanley*, and other resources for this section, can be found at: `https://greensoftwarelab.github.io/jStanley`

grammer to choose which of the three options are closest to what they believe would best represent the program.

Figure 5.6: *jStanley* Eclipse menu



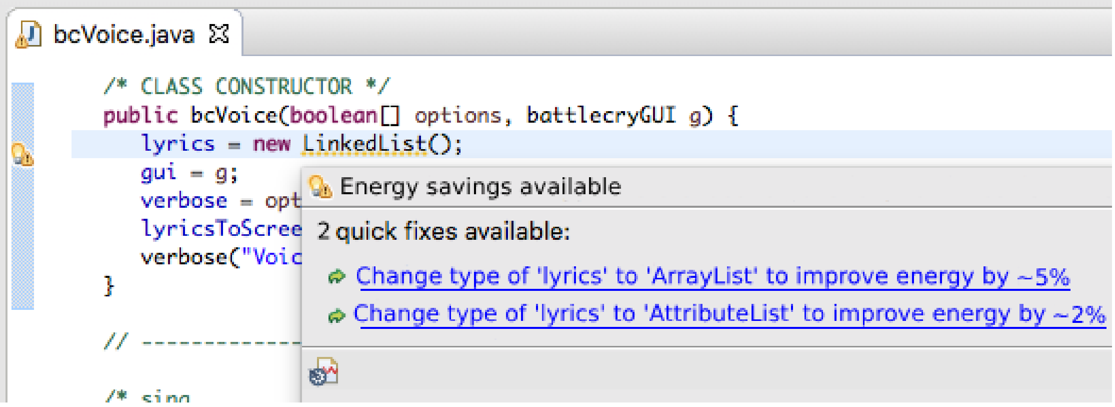
Before analyzing a program, the tool loads the data tables (from CSV files) corresponding to the settings chosen by the programmer. This information is first normalized by sorting each method from each collection and attributing a value of 1 to the lowest. Afterwards, each other value is divided by the lowest, obtaining a value greater than 1 indirectly encoding the percentage of how worse that collection's method is to the lowest one. This is done for both energy and time values, allowing the tool to combine these values to obtain an overall ranking if both options are chosen to optimize.

The calculated suggestions can be shown visually through source code flagging, as shown in Figure 5.7. Here a small flag appears next to an identified collection which may be changed to a more optimized one, and shows the programmer the best two alternatives. If the programmer wishes, they may select the option to change the collections to the suggested one. It is to note that pure refactoring properties may not be guaranteed when changing collections, for example natural sorting in `Tree` collections, or the non-acceptance of `null` values in `HashTable` collections.

### 5.4.1 Implementation

Four tasks divide the tool's analysis. The first task, **Source Code Analysis**, detects existing collections within the program, and all the invocations on these collections. Using the constructed AST, and the *ASTVisitor* offered by the Eclipse JDT API, we can easily visit each vari-

Figure 5.7: Suggestion flagging and quick-fix



```
/* CLASS CONSTRUCTOR */
public bcVoice(boolean[] options, battlecryGUI g) {
   lyrics = new LinkedList();
   gui = g;
   verbose = opt
   lyricsToScree
   verbose("Voic
}

// ------------

/* sing
```

Energy savings available

2 quick fixes available:

- Change type of 'lyrics' to 'ArrayList' to improve energy by ~5%
- Change type of 'lyrics' to 'AttributeList' to improve energy by ~2%

able and method invocation. Using the *ASTVisitor*, we collect all the AST nodes which are *FieldDeclarations*, *VariableDeclarationStatements*, and *Assignments*. These nodes allow us to determine the data type of a variable and focus on those which are collections.

Afterwards, we analyze all the *MethodDeclaration* nodes. Often times, collections are passed to methods as parameters. Thus, we also collect all these method references along with the type of collections passed through, allowing us to match declarations and method invocations.

Finally, all *MethodInvocation* nodes are analyzed to determine which are JCF API methods which may be invoked during the program's execution. In these cases, we divide them in two types: direct and indirect invocations. Direct invocations would be for example a line of code with `students.add();`. An indirect invocation represents methods declared throughout the program, in which a direct invocation may occur. For example, method `m1` contains a line with `students.add(); students.add();` (two direct invocations), thus `m1` would be an indirect invocation. The result of this analysis is a list of all the existing collections within the program, all direct invocations of each collection (and their source code location), all indirect invocations, and the collections which are passed as parameters of these indirect invocations.

In the second task, **Resolve Invocations**, using the information we now have on all the direct and indirect invocations, we calculate the amount of times each method may be used.

For example, `LinkedList.add();` is within method `m1`. So we know the `add()` method on `LinkedList` will be executed once. But method `m1` is invoked twice within another method, so we may assume the `add()` method may be invoked at least twice, and not only once. Additionally, during an indirect invocation where a collection is passed as a parameter, a match is also made with our existing list of collections. This way, every possible path a collection may go through and all possible method invocations upon the collection are traced.

Afterwards in the **Calculate Cost** task, the cost of each collection, considering all the existing invocations and data tables, is calculated. *jStanley* calculates a matrix with all the invocation costs of each collection and method by multiplying the total number of invocations of each method by its normalized value.

Finally, in **Calculate Suggestions**, the sum of all the method costs are calculated for each collection (representing either the total cost of energy, time, or both based on selection).

### 5.4.2 Evaluation

To evaluate our tool regarding the energy savings it promotes, we selected 7 Java projects which use JCF collections and have either a test suite or simulated example of the program's execution. We obtained these projects from SourceForge (Media, 2018), a repository for open-source applications, and from the SF110 corpus of classes (Fraser and Arcuri, 2014), a statistically representative sample of 110 Java projects.

The selected projects and some of their metrics are listed in Table 5.4. The projects vary from games to time libraries. The size and complexity also varies between 3.000 and 70.000 lines of code (LOC) or between 76 and 5.000 branches in the flow graph control.

We ran *jStanley*, according to a 25k population size (since we are using only unit tests which tend to not stress collections as much), on each project and obtained the list of suggested energy optimizations, which were automatically applied. For each new project version, we re-ran the test suite, obtaining exactly the same results as the original ones. This means that, considering the available tests, the changes acted as refactorings. The amount of time spent by our tool to analyze, and the number of suggested changes are found in Table 5.5 under the *Analysis (ms)* and *#Changes* column.

Table 5.4: Metrics of the evaluated evaluated. These metrics were calculated using OpenClover (Atlassian, 2018)

| Project | Branches | Statements | Methods | Classes | Files | Packages | LOC |
|---|---|---|---|---|---|---|---|
| Barbecue (for bar codes) | 536 | 2,536 | 369 | 59 | 59 | 13 | 8,838 |
| Battlecry (game) | 534 | 1,800 | 125 | 12 | 11 | 1 | 3,343 |
| Jodatime (time library) | 5,162 | 13,318 | 3,909 | 242 | 166 | 7 | 70,872 |
| Lagoon (web site maintenance) | 1,746 | 4,211 | 646 | 96 | 81 | 10 | 16,922 |
| Templateit (template file generator) | 408 | 1,067 | 177 | 22 | 19 | 3 | 3,317 |
| Twfbplayer (game) | 684 | 3,307 | 777 | 135 | 104 | 12 | 14,682 |
| Xisemele (XML library) | 76 | 522 | 250 | 57 | 56 | 3 | 5,770 |

Table 5.5: Evaluation data for the projects

| | Test Suite | | Analysis | | Improvement | | |
|---|---|---|---|---|---|---|---|
| Project | #Tests | %Coverage | Analysis (ms) | #Changes | %PKG (J) | %CPU (J) | %ms |
| Barbecue | 152 | 62 | 2735 | 14 | 5.10 | 5.81 | 1.70 |
| Battlecry | 1* | 69.4 | 514 | 4 | 16.79 | 11.49 | 12.76 |
| Jodatime | 4221 | 88.5 | 10490 | 5 | 7.21 | 7.29 | 7.75 |
| Lagoon | 18 | 4 | 1513 | 7 | 1.55 | 1.77 | 2.05 |
| Templateit | 3 | 14 | 1019 | 14 | 6.07 | 6.05 | 3.14 |
| Twfbplayer | 57 | 91 | 3437 | 51 | 6.04 | 6.30 | 4.36 |
| Xisemele | 167 | 20 | 588 | 1 | 4.25 | 4.38 | 3.18 |

* Instead of unit tests, this project has a simulated execution example

To measure the energy consumed by each project, before and after the changes, we used Intel's Runtime Average Power Limit (RAPL) (Dimitrov et al., 2015). RAPL is an interface provided by (modern) Intel processors to allow the access to energy and power readings. RAPL is capable of providing very fine-grained level measurements as it has already proven (Hähnel et al., 2012; Rotem et al., 2012). For our study, we measured 2 RAPL domains: PKG energy consumed by an entire socket (including the core and uncore domains); PP0 energy consumed by the CPU core.

This study was executed on a laptop with Ubuntu 14.04.5 LTS, 6GB of RAM, and Intel(R) Core(TM) i5-2430 CPU @ 2.40GHz. Both the Java compiler and interpreter were versions 1.8.0_101.

We ran each project's test suite 25 times (Hogg and Tanis, 1977), and for each execution, we extracted the energy consumed in Joules (J) for both RAPL domains, and the execution time in milliseconds (ms). The number of tests and test coverage percentage are listed in Table 5.5 under the *#Tests* and *%Coverage* column respectively.

The energy consumption improvements are shown in Table 5.5. Column *%PKG (J)* and

*%CPU (J)* show the energy improvement percentage relative to the original project measured by the package and CPU respectively. Column *%ms* shows the time improvement percentage. Each value is calculated as the average of the 25 executions, excluding outliers, that is, values outside of the range $[Q1 - 1.5 \times IQ, Q3 + 1.5 \times IQ]$, where *Q1* and *Q3* are the first and the third quartiles, respectively, and $IQ = Q3 - Q1$ (Tukey, 1977). Indeed it is common to remove outliers for energy measurements (Cruz and Abreu, 2017). For instance, we know, from experience, that the first few runs of Java programs tend to spent more energy than the remaining runs (Pereira et al., 2017b, 2016; Couto et al., 2014).

Using *jStanley*, we were able to achieve between 2%-17% with an average of 6.7% energy savings for PKG, and between 2%-11% with an average of 6.2% energy savings for CPU. Additionally, the performance was also improved upon with an average of 5%. *jStanley* spent on average 3 seconds to analyze our projects, with the fastest and slowest spending 0.5 and 10 seconds respectively.

These values are positive, especially when compared to the little work required by the developer. Moreover, in some cases the tests' coverage was small, maybe too small to actually stress the collections enough to make the gains more evident. Also, the fact that we used unit tests instead of actual software runs may impact the results, but most likely in a negative way. A real usage of the applications would make more use of the collections thus most likely making the gains higher. In any case, building on these positive results, we will run an in depth evaluation to fully understand the potential of *jStanley*.

## 5.5 Threats to Validity

The goal of our experiments was to define the energy consumption profile of JCF implementations and validate such results. As in any experiment, there are a few threats to its validity. We start by presenting the validity threats for the first experiment, that is, the evaluation of the energy consumption of several Java data structure methods. We divide these threats in four categories as defined in (Cook and Campbell, 1979), namely: conclusion validity, internal validity, construct validity, and external validity.

### 5.5.1 JCF Implementations Profile

We start by discussing the threats to validity of the first experiment.

**Conclusion Validity** We used the energy consumption measurements to establish a simplistic order between the different implementations. To do so, we have based ourselves on an existing benchmark (although developed to measure different things). To perform the actual measurements, we used RAPL which is known to be a quite reliable tool (Hähnel et al., 2012; Rotem et al., 2012). Thus, we believe the finding are quite reliable.

**Internal Validity** The energy consumption measurements we have for the different implementations/methods could have been influenced by other factors other than just their source code execution. To mitigate this issue, for every test we added a "warm-up" run, and ran every test 10 times, taking the average values for these runs so we could minimize particular states of the machine and other software (e.g. operating system daemons). Moreover, we ran the tests in a Linux server with no other software running except for the operating system and its services in order to isolate the energy consumption values for the code we were running.

**Construct Validity** We have designed a set of tests to evaluate the energy consumption of the methods of the different JCF implementations. As software engineers ourselves, we have done the best we can and know to make them as real and interesting as possible. However, these experiments could have been done in many other different ways. In particular, we have only used strings to perform our evaluation. We have also fixed the size of the collections in 25K, 250K, and 1M. Nevertheless, we believe that since all the tests are the same for all the implementations (of a particular interface), different tests would probably produce the same relationship between the consumption of the different implementations and their methods. Still, we make all our material publicly available for better analysis of our peers.

**External Validity** The experiment we performed can easily be extended to include other collections. The method can also be easily adapted to other programming languages. However, until such execution are done, nothing can be said about such results.

### 5.5.2 Validating the Measurements

Next we present the threats to validity, again divided in four categories, for the experiment we performed to evaluate the impact of the finding of the first study when changing the implementations in a complete program.

**Conclusion Validity** Our validation assumed that each method is on the same level of importance or weight, and does not distinguish between possible gain of optimizing for one method or another (for instance, there might be more gain in optimizing for a commonly used add method over a retainAll method). Thus, the method of choosing the best alternative implementation would need fine tuning. Nevertheless, it is consistent that changing an implementation by another influences the energy consumption of the code in the same line with the results found for the implementations/methods in the first experiment.

**Internal Validity** The energy consumptions measures we have for the different projects (before and after changing the used implementations) could have influence from other factors. However, the most important thing is the relationship between the value before and after changing the implementations. Nevertheless, we have executed each project 10 times and calculated the average so particular states of the machine could be mitigated as much as possible in the final results.

**Construct Validity** We used 5 different (project) implementation of a single problem developed by students in the second semester of an under-graduation in Computer Science. This gave us different solutions for the same problem that can be directly compared as they all passed a set of functional tests defined in the corresponding course. However, different kinds of projects could have different results. Nevertheless, there is no basis to suspect that these projects are best or worst than any other kind. Thus, we expect to continue to see gains/losses when changing implementations in any other kinds of software projects according to our findings.

**External Validity** The used source code has no particular characteristics which could influence our findings. The main characteristic is possibly the fact that it was developed by novice programmers. Nevertheless, we could see the impact of changing data structure implementations in both the good and bad (project) implementations. Thus, we believe that these results can be further generalized for other projects. Nevertheless, we intend to further study this issue and perform a wider evaluation.

## 5.6 Conclusions

This chapter presented a detailed study of the energy consumption of the Sets, Lists, and Maps data structures included in the Java collections framework. We presented a quantification of the energy spent by each API method of each of those data structures.

Moreover, we introduced a very simple methodology to optimize Java programs. Based on their JCF data structures and methods, and our energy quantifications, a transformation to decrease the energy consumption is suggested. We have presented our first experimental results that show a decrease of 6.2% in energy consumption.

We also presented *jStanley*, a tool capable of quickly discovering the usage of energy-inefficient Java collections. It can also suggest and automatically evolve the code with better alternatives. The initial evaluation we performed shows promising results with savings between 2% and 17%.

As future work, we will extend the available collections, consider their memory usage, and make suggestions of collections of different kinds (e.g. list to map), performing the corresponding evolution, leading to even greater savings.

# Chapter 6

# Towards a Catalog of Energy Smells

*This chapter briefly looks at an overview of simple and quick changes in the Java programming language which help improve the energy consumption. These findings were observed throughout various stages during this thesis, in particular during the empirical studies performed in Chapter 4.*

*We look at the energy consumption of Java primitives in different scenarios. The work here is to further understand these scenarios, even if in most cases the results are not too surprising, thus helping Java developers if energy is a concern. Here we look at different numerical primitives, array initialization examples, and String concatenation alternatives.*

*These alternatives originated from gathering observations when working on other works. Preliminary tests show us very promising energy efficient alternatives. Nevertheless, the next step would be a thorough and large validation.*

## 6.1 Java Primitives

We have seen in Chapter 5 how data structures have a large impact on the energy efficiency of Java programs. Another very common way to store data in a Java program is to use primitive types for numerical variables, Arrays, and Strings. These too behave different in terms of energy consumption, sometimes slightly while other times very drastically.

In this section, we will look at the energy efficiency of different numerical primitive types, different array initializations, and alternative ways to concatenate Strings.

For these, we once again used RAPL to perform the energy measurements for our tests. The following used system allowed us to measure the Package (PKG), CPU, and DRAM energy consumption: Linux Ubuntu Server 16.10 operating system, kernel version 4.8.0-22-generic, with 16GB of RAM, a Haswell Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz.

For each of the numerical primitives, arrays, and String tests, a specific usage scenario was defined (defined further on). These scenarios were repeated 20 times on our measuring system.

As in Section 5.4.2, each final measurement value is calculated as the average of the 20 executions, excluding outliers, that is, values outside of the range $[Q1-1.5\times IQ, Q3+1.5\times IQ]$, where *Q1* and *Q3* are the first and the third quartiles, respectively, and $IQ = Q3 - Q1$ (Tukey, 1977). Additionally, the time spent was also measured.

For the numerical primitive types, each usage scenario consisted of a summation of 1 to the previous value (starting with the value 1). This was repeated 2,147,483,647[1] times. Additionally, as the `Integer` data type is also very widely used in Java, it too was added in the analysis to better understand how different it performs when compared to an `int`. After viewing the results, we added a scenario using a `static int` to understand the impact of a static value in terms of energy consumption.

As we will see further on, the `int` primitive was the best performing one. Thus, we chose `int` to be the array type for our second set of tests. In Java, there are two possible ways of initializing an array. One can declare a variable with an array of a certain type (for example,

[1] This is the max value an int may have

int[] x), or one can declare an array variable of a certain type (for example, int x[]). While these two possibilities do not tend to have much of a difference in the performance (in fact there should be almost no difference), we wanted to explore the different in terms of energy consumption. The defined usage scenario consisted of storing the array's index as its held value. The only difference between the two is how the array was declared (int[] x or int x[]). Finally, two versions were tested: one with 1,000,000 elements and one with 5,000,000 elements.

The common form of storing textual values in Java is by using the `String` data type. While there are no real alternatives to this, there are many forms of string concatenation. The most basic and common form is by using the (+) operator. Unfortunately, this form is not very efficient in terms of performance and programmers tend to use `StringBuilder` or `StringBuffer` for their concatenation needs. Another alternative to these is by using the `String.concat()` method from the `String` API. The usage scenario consisted of the concatenation of *"this is a string"*. Finally, two versions were tested: one with 20,000 repetitions and one with 100,000 repetitions.

### 6.1.1 Results

The tabled results for our tests can be seen in Tables 6.1-6.3, and also visually in Figures 6.1-6.7. The measured values for the PKG, CPU, and DRAM are shown in Joules while the execution time is presented in milliseconds (MS). The last two columns on each table respectively represent the normalized values anchored to the most energy and performance efficient solution. Thus for example, in Table 6.1, the `double` Java primitive is 498.20x more energy inefficient, and 713.56x slower than the `int` primitive.

Looking at Table 6.1 and Figure 6.1, we quickly see that there is no competition for the `int` type, with an energy consumption of 0.060J and 0.047J for the Package and CPU respectively. Following behind is the `Long` type with 0.963J and 0.492, respectively for the Package and CPU energy consumption. The worst performing is the `Integer` data type, as was expected. Defining `static int` increased the energy consumption by 35.75x in comparison to a standard `int` declaration. This increase is expected to be replicated on the other primitives.

Looking at the normalized results, we see that the alternatives tend to be more performance inefficient than energy inefficient, with the exception of the `Integer` data type where the opposite is true.

Table 6.1: Energy consumption and Time for different numerical primitives

| Name | PKG J | CPU J | DRAM J | MS | Norm. PKG | Norm. MS |
|---|---|---|---|---|---|---|
| Double | 29.698 | 12.733 | 4.597 | 1902.833 | 498.20 | 713.56 |
| Float | 29.701 | 12.746 | 4.597 | 1902.500 | 498.24 | 713.44 |
| Int | 0.060 | 0.047 | 0.007 | 2.667 | 1.00 | 1.00 |
| Integer | 112.214 | 57.839 | 23.985 | 4844.667 | 1882.44 | 1816.75 |
| Long | 0.963 | 0.492 | 0.134 | 55.500 | 16.15 | 20.81 |
| Short | 19.428 | 8.230 | 3.068 | 1270.000 | 325.90 | 476.25 |
| StaticInt | 2.145 | 1.069 | 0.297 | 122.667 | 36.99 | 46.00 |

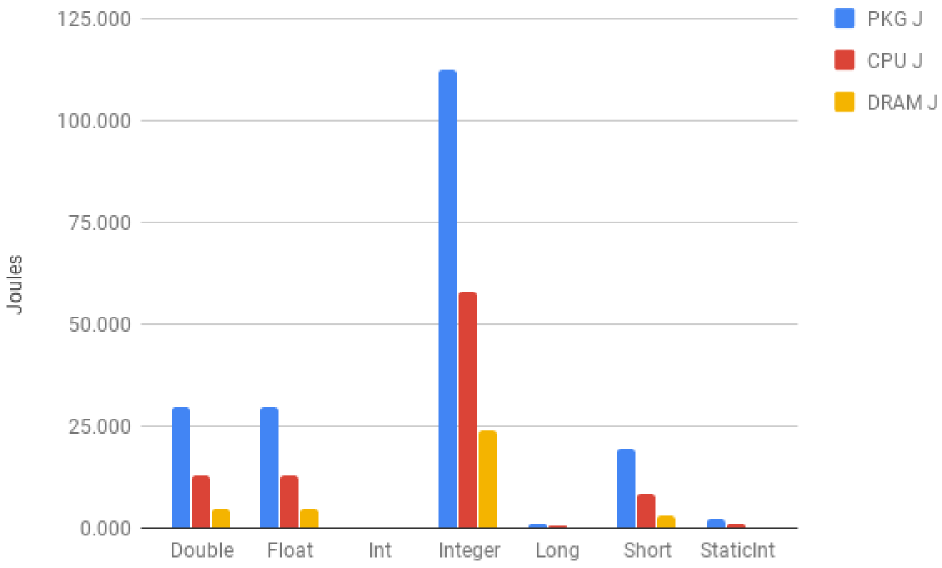


Figure 6.1: Energy consumption and Time for different numerical primitives

The results for the different Array initialization alternatives (Table 6.2 and Figures 6.2-6.3), show that there is small, yet existent, difference between the two. Interestingly, in the two differently sized scenarios, we see that the execution time is in favor of the second declaration (int x[]), yet the energy consumption is less in the first declaration (int[] x). These two approaches should be converted to the same byte code, but they seem to have different results. The normalized results show how the second declaration (int x[]) is 1.06x more energy costly, while the first (int[] x) is 1.08x slower. These differences are small, they should be considered if these are operations which are repeated quite often and energy is a concern. Finally, the larger the array size, the more similar the results become.

Table 6.2: Energy consumption and Time for different Array initializations

| Size | Initialization | PKG J | CPU J | DRAM J | MS | Norm. PKG | Norm. MS |
|---|---|---|---|---|---|---|---|
| **1M** | int[size] x | 0.096 | 0.081 | 0.013 | 4.667 | 1.00 | 1.08 |
| | int x[size] | 0.102 | 0.081 | 0.014 | 4.333 | 1.06 | 1.00 |
| **5M** | int[size] x | 0.193 | 0.132 | 0.031 | 8.000 | 1.00 | 1.02 |
| | int x[size] | 0.196 | 0.133 | 0.031 | 7.833 | 1.01 | 1.00 |

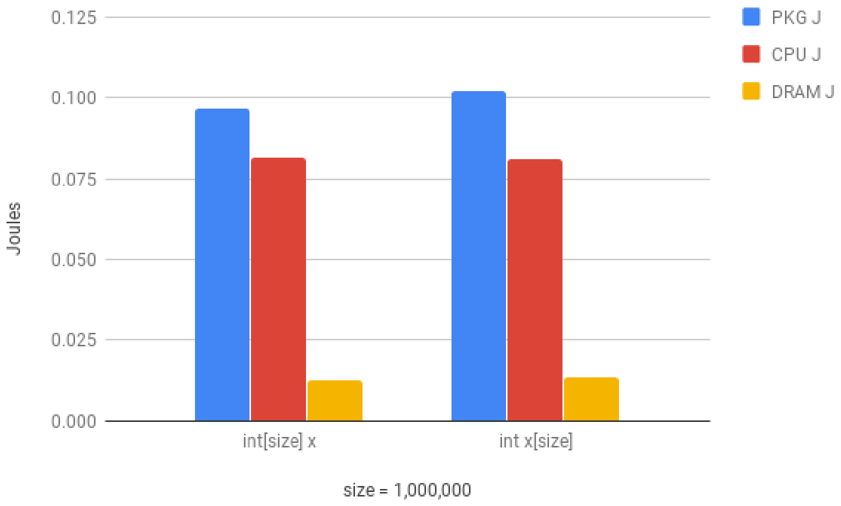


Figure 6.2: Energy consumption and Time for different Array initializations (1M)

Table 6.3 and Figures 6.4-6.7 show the results for the different `String` concatenation alternatives. Looking at the results, we quickly notice how much more energy and time costly is a `String` concatenation using the (+) operator when compared to the other alternatives. The `String.concat()` method, developed to be a more efficient alternative to using the (+) operator, still does not compete with using the `StringBuilder` and `StringBuffer` API. These latter two compete very closely, with `StringBuilder` being the more energy efficient one. With the smaller repetition size of 20,000, we see that `StringBuffer` is slightly faster, but consumes slightly more energy. While with a larger repetition size of 100,000, these two have the same execution time, while `StringBuilder` is still the more energy efficient one.

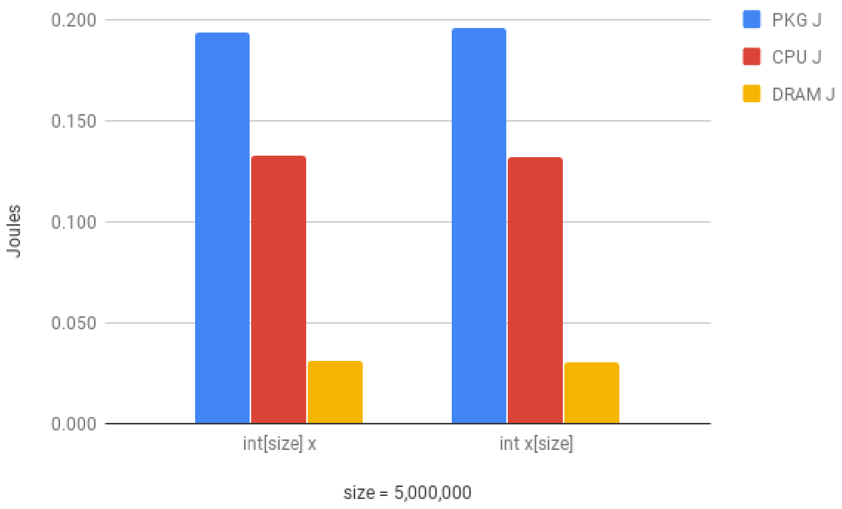


Figure 6.3: Energy consumption and Time for different Array initializations (5M)

Table 6.3: Energy consumption and Time for different String concatenations

| Size | Name | PKG J | CPU J | DRAM J | MS | Norm. PKG | Norm. MS |
|---|---|---|---|---|---|---|---|
| 20k | String.concat() | 16.859 | 9.018 | 3.908 | 748.333 | 333.53 | 320.71 |
| | StringConcat (+) | 54.901 | 29.525 | 12.927 | 2436.833 | 1086.13 | 1044.36 |
| | StringBuffer | 0.073 | 0.061 | 0.008 | 2.333 | 1.45 | 1.00 |
| | StringBuilder | 0.051 | 0.050 | 0.006 | 2.500 | 1.00 | 1.07 |
| 100k | String.concat() | 461.053 | 237.611 | 114.082 | 20039.000 | 2434.68 | 3206.24 |
| | StringConcat (+) | 1526.655 | 797.131 | 372.493 | 67398.000 | 8061.78 | 10783.68 |
| | StringBuffer | 0.227 | 0.171 | 0.024 | 7.250 | 1.20 | 1.16 |
| | StringBuilder | 0.189 | 0.148 | 0.020 | 7.250 | 1.00 | 1.00 |

Finally, the normalized values show us just how much worse (1086.13x energy inefficient) is using the concatenation operator (+) compared to using a `StringBuilder`. We also see that while with 20k repetitions `StringBuilder` is slower (1.07x) than `StringBuffer`, at 100k repetitions `StringBuilder` is both the fastest and most energy efficient alternative.

Looking at these simple preliminary tests, we see several alternative approaches, for the presented problems, with promising energy efficient results. While in some cases there may be small differences between two approaches, if one is concerned with energy, it is important to know which has the edge. The next step would be to apply these changes and test them in real programs, and observe if the behaviors are replicated.

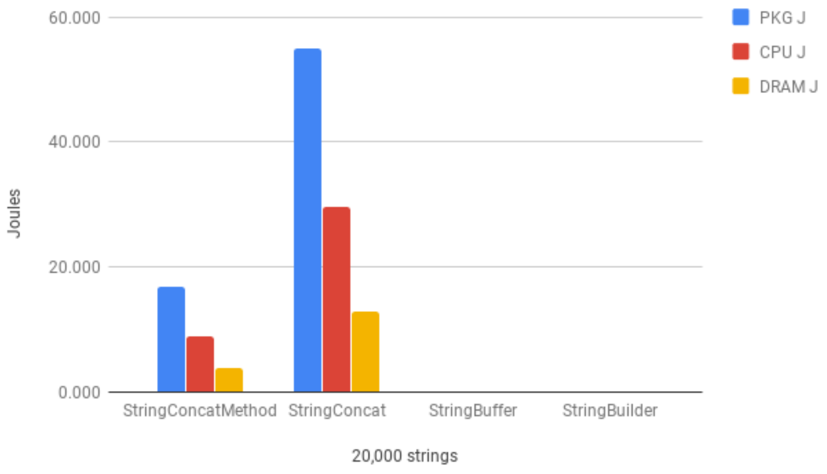


Figure 6.4: Energy consumption and Time for different String concatenations (20k)

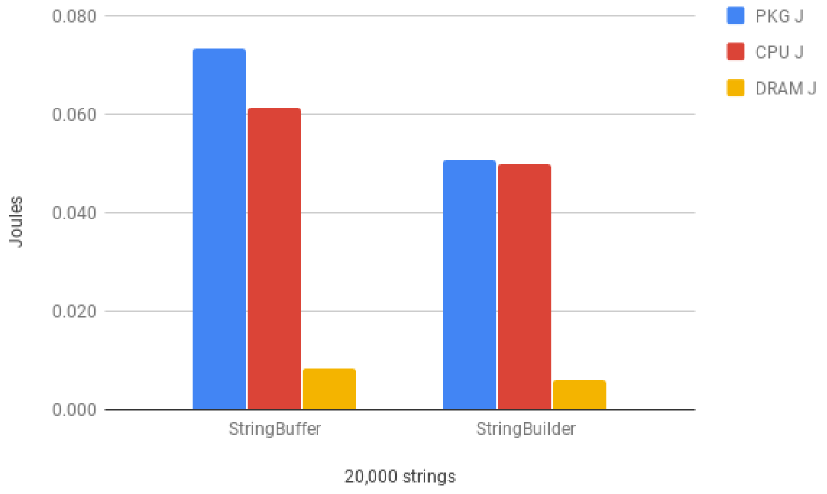


Figure 6.5: Energy consumption and Time for different StringBuilder and StringBuffer (20k)

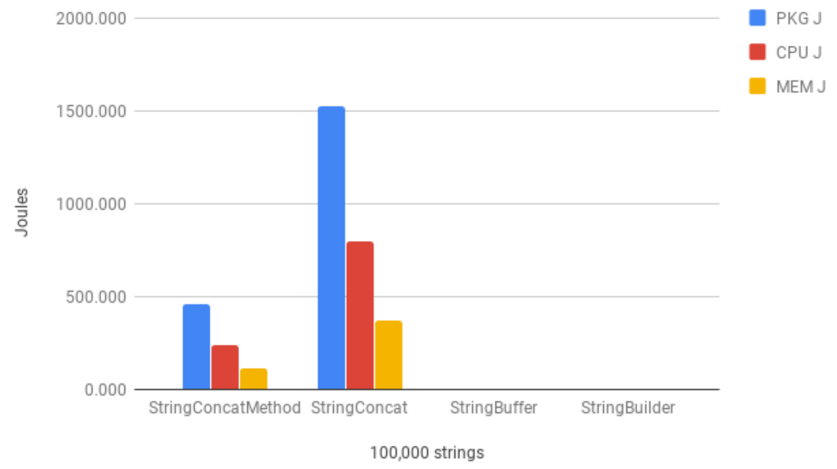


Figure 6.6: Energy consumption and Time for different String concatenations (100k)

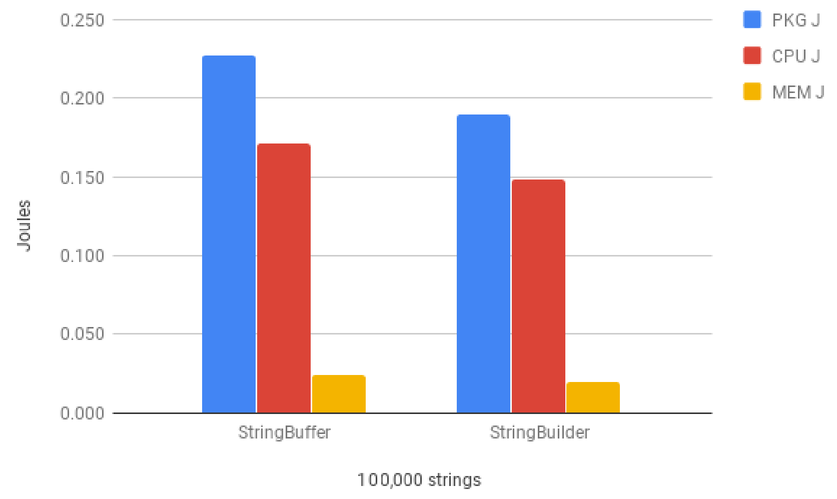


Figure 6.7: Energy consumption and Time for different String concatenations (100k)

# Chapter 7

# Conclusions and Future Work

*This chapter shares the final thoughts and conclusions on the work presented in this document. The research work presented in the previous chapters is revisited and summarized, with a look back at each of this thesis's research questions.*

*Finally, a view on several possible research paths of which both ourselves and other researchers may explore is also presented.*

We have seen throughout this thesis how simple changes can have great impacts on the energy consumption in software. Unfortunately, software developers do not usually have the knowledge and tools needed for energy-aware software development, an argument shared by almost all within the community. Additionally, there are many misconceptions as to what can be done to reduce the energy consumption, how to interpret their energy measurements, and how optimizing for time is enough to make software more energy efficient. This thesis makes a new dent in the bubble of knowledge in the research area of green software.

In Chapter 3, we began looking at the first step a developer takes, choosing a programming language. We saw how different languages perform in terms of energy consumption, time, and peak memory usage from our executed study. Indeed, there are many differences between various languages' energy efficiency (and performance). Currently, there seems to be no clear winner in terms of energy efficiency, a spot which is shared between three. Additionally, we looked at if a faster language is always the most energy efficient, how peak memory usage relates to energy consumption, and offered a way to help developers choose a language depending on what their main objectives are. This chapter dealt with *the lack of knowledge*, and shows how different programming languages have very different energy consumption profiles, thus answering **TRQ1**: *What influence do different programming languages have on energy consumption?*

Chapter 4 looks at a language and context independent technique and tool called SPELL. This technique aims at helping developers identify energy leaks, or energy hotspots, within a program's source code. This technique is based on Spectrum-based Fault Localization, a state of the art technique to identify code bugs or faults. The technique was developed into a prototype, along with a tool-kit for Java developers, to help find energy inefficient code blocks. A study was performed where we saw how using SPELL can drastically help improve the overall energy efficiency of programs. Additionally, we also showed how programs improved using SPELL are significantly more energy efficient than those using a profiler, once again showing how looking at performance efficiency is not enough to tackle the energy consumption problem. This chapter dealt with *the lack of tools*, and answers our second research question: **TRQ2**: *Can fault localization techniques be adapted to detect energy hotspots in source code?*

Chapter 5 looks at a more specific problem, the energy efficiency of Java collections. A detailed study was performed on the energy consumption of the full Java Collection Framework implementation. The energy consumptions of each method were extracted throughout different population sizes. By collecting this data, we were able to produce different data tables, divided into the different interfaces of Sets, Lists, and Maps of varying population sizes. A methodology based on method usage was presented to use this information to choose the most appropriate collection if energy consumption is a concern. Additionally, a tool named *jStanley* was developed to automatically find JCF collections and methods in a Java program, and suggest alternatives to improve the code based on energy usage, time, or a balance of both. Finally, we were able to show how doing such changes allowed us to reduce the energy consumption considerably. This chapter dealt with both *the lack of knowledge* and *the lack of tools*, and answers our final research question: **TRQ3**: *What influence do different Java data structures and their methods have on energy consumption?*

Chapter 6 discusses some of the interesting observations found during work on this thesis. The analyses and tests presented in this chapter are preliminary in nature, yet show very promising energy efficient approaches to certain issues. Nevertheless, a more thorough validation is needed to certify the consistency across other scenarios.

## 7.1 Future Work

This thesis focused on two main problems in regards to energy efficient software development: **the lack of knowledge** and **the lack of tools**. While the presented contributions have furthered the advancement of *Energyware Engineering*, there is still more which can be done on the lack of knowledge and tools.

Chapter 5 looked at data structures in one specific programming language (Java) for desktop programs. It would be very interesting to explore the implications of these same data structures in an Android setting (which is also based off of Java). Additionally, we only looked at the data structures from the Java Collection Framework, but there are other popular data structures in other libraries such as the Apache Commons Collections[1]. Finally, researching the energy efficiency of data structures in other languages (for example in the C language) would also bring about a better understanding of this topic.

The tool presented in Chapter 4, SPELL - SPectrum-based Energy Leak Localization, currently only indicates what code fragments are most probably to have an energy leak. Currently, we are looking at ways to incorporate jStanely within the SPELL analysis. The idea is to have the jStanely analysis performed if SPELL detects a code fragment containing a JCF data structure, and thus quickly suggesting an alternative one. We are also looking at having SPELL suggest alternatives based on the observations in Chapter 6.

This thesis looked at many misconceptions on the energy problem, and tackled the *lack of knowledge* by answering many questions on *what* contributes to high energy consumption and by offering guidelines for programmers to follow. While out of the scope of this thesis, an interesting research path based on the presented work would be to answer *why* such things occur as they do. A low-level analysis on programming languages and data structures would be complementary to this thesis. Additionally, the application of the produced knowledge and tools within concepts such as wearable devices, harvesting devices, and the Internet of Things (IoT), would bring about compelling and impactful contributions.

[1] Apache Commons Collections:
`https://commons.apache.org/proper/commons-collections/`

# Appendix A

# Chapter 3 Appendix

## A.1 Compiler versions

Table A.1: Compilers for languages in the Chapter 3 studies

| | **Version** | | **Version** |
|---|---|---|---|
| Ada | GNAT 6.2.0 | Jruby | jruby 9.1.7.0 |
| C | gcc 6.2.0 | Lisp | SBCL 1.3.3.debian |
| C# | dotnet 1.0.1 | Lua | Lua 5.3.3 |
| C++ | g++ 6.2.0 | Ocaml | ocamlopt 4.05.0 |
| Chapel | chpl 1.15.0 | Pascal | fpc 3.0.2 |
| Dart | Dart VM 1.24.0-dev.0.0 | Perl | perl 5.24.0 |
| Erlang | Erlang 7.3.1.2 | PHP | php 7.1.4 |
| F# | dotnet 1.0.1 | Python | python 3.5.2 |
| Fortran | ifort 17.0.3 | Racket | raco 6.8 |
| Go | go go1.6.3 | Ruby | ruby 2.4.1 |
| Hack | HipHop VM 3.19.2 | Rust | rustc 1.16.0 |
| Haskell | ghc 8.0.2 | Swift | swift 4.0-dev |
| Java | jdk 1.8.0_121 | TypeScript | node 7.9.0 |
| JavaScript | node 7.9.0 | | |

## A.2 Data tables

Table A.2: Results for binary-trees, fannkuch-redux, and fasta

| | binary-trees | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) C | 39.80 | 1125 | 0.035 | 131 |
| (c) C++ | 41.23 | 1129 | 0.037 | 132 |
| (c) Rust $\Downarrow_{2}$ | 49.07 | 1263 | 0.039 | 180 |
| (c) Fortran $\Uparrow_{1}$ | 69.82 | 2112 | 0.033 | 133 |
| (c) Ada $\Downarrow_{1}$ | 95.02 | 2822 | 0.034 | 197 |
| (c) Ocaml $\downarrow_{1}$ $\Uparrow_{2}$ | 100.74 | 3525 | 0.029 | 148 |
| (v) Java $\uparrow_{1}$ $\Downarrow_{16}$ | 111.84 | 3306 | 0.034 | 1120 |
| (v) Lisp $\downarrow_{3}$ $\Downarrow_{3}$ | 149.55 | 10570 | 0.014 | 373 |
| (v) Racket $\downarrow_{4}$ $\Downarrow_{6}$ | 155.81 | 11261 | 0.014 | 467 |
| (i) Hack $\uparrow_{2}$ $\Downarrow_{9}$ | 156.71 | 4497 | 0.035 | 502 |
| (v) C# $\downarrow_{1}$ $\Downarrow_{1}$ | 189.74 | 10797 | 0.018 | 427 |
| (v) F# $\downarrow_{3}$ $\Downarrow_{1}$ | 207.13 | 15637 | 0.013 | 432 |
| (c) Pascal $\downarrow_{3}$ $\Uparrow_{5}$ | 214.64 | 16079 | 0.013 | 256 |
| (c) Chapel $\uparrow_{5}$ $\Uparrow_{4}$ | 237.29 | 7265 | 0.033 | 335 |
| (v) Erlang $\uparrow_{5}$ $\Uparrow_{1}$ | 266.14 | 7327 | 0.036 | 433 |
| (c) Haskell $\uparrow_{2}$ $\Downarrow_{2}$ | 270.15 | 11582 | 0.023 | 494 |
| (i) Dart $\downarrow_{1}$ $\Uparrow_{1}$ | 290.27 | 17197 | 0.017 | 475 |
| (i) JavaScript $\downarrow_{2}$ $\Downarrow_{4}$ | 312.14 | 21349 | 0.015 | 916 |
| (i) TypeScript $\downarrow_{2}$ $\Downarrow_{2}$ | 315.10 | 21686 | 0.015 | 915 |
| (c) Go $\uparrow_{3}$ $\Uparrow_{13}$ | 636.71 | 16292 | 0.039 | 228 |
| (i) Jruby $\uparrow_{2}$ $\Downarrow_{3}$ | 720.53 | 19276 | 0.037 | 1671 |
| (i) Ruby $\Uparrow_{5}$ | 855.12 | 26634 | 0.032 | 482 |
| (i) PHP $\Uparrow_{3}$ | 1,397.51 | 42316 | 0.033 | 786 |
| (i) Python $\Uparrow_{15}$ | 1,793.46 | 45003 | 0.040 | 275 |
| (i) Lua $\downarrow_{1}$ | 2,452.04 | 209217 | 0.012 | 1961 |
| (i) Perl $\uparrow_{1}$ | 3,542.20 | 96097 | 0.037 | 2148 |
| (c) Swift | n.e. | | | |

| | fannkuch-redux | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) C $\Downarrow_{2}$ | 215.92 | 6076 | 0.036 | 2 |
| (c) C++ $\Uparrow_{1}$ | 219.89 | 6123 | 0.036 | 1 |
| (c) Rust $\Downarrow_{11}$ | 238.30 | 6628 | 0.036 | 16 |
| (c) Swift $\Downarrow_{5}$ | 243.81 | 6712 | 0.036 | 7 |
| (c) Ada $\Downarrow_{2}$ | 264.98 | 7351 | 0.036 | 4 |
| (c) Ocaml $\downarrow_{1}$ | 277.27 | 7895 | 0.035 | 3 |
| (c) Chapel $\uparrow_{1}$ $\Downarrow_{18}$ | 285.39 | 7853 | 0.036 | 53 |
| (v) Lisp $\downarrow_{3}$ $\Downarrow_{15}$ | 309.02 | 9154 | 0.034 | 43 |
| (v) Java $\uparrow_{1}$ $\Downarrow_{13}$ | 311.38 | 8241 | 0.038 | 35 |
| (c) Fortran $\Downarrow_{1}$ | 316.50 | 8665 | 0.037 | 12 |
| (c) Go $\uparrow_{2}$ $\Uparrow_{7}$ | 318.51 | 8487 | 0.038 | 2 |
| (c) Pascal $\Uparrow_{10}$ | 343.55 | 9807 | 0.035 | 2 |
| (v) F# $\downarrow_{1}$ $\Downarrow_{7}$ | 395.03 | 10950 | 0.036 | 34 |
| (v) C# $\uparrow_{1}$ $\Downarrow_{5}$ | 399.33 | 10840 | 0.037 | 29 |
| (i) JavaScript $\downarrow_{1}$ $\Downarrow_{2}$ | 413.90 | 33663 | 0.012 | 26 |
| (c) Haskell $\uparrow_{1}$ $\Uparrow_{8}$ | 433.68 | 14666 | 0.030 | 7 |
| (i) Dart $\Downarrow_{7}$ | 487.29 | 38678 | 0.013 | 46 |
| (v) Racket $\Uparrow_{3}$ | 1,941.53 | 43680 | 0.044 | 18 |
| (v) Erlang $\Uparrow_{3}$ | 4,148.38 | 101839 | 0.041 | 18 |
| (i) Hack $\Downarrow_{6}$ | 5,286.77 | 115490 | 0.046 | 119 |
| (i) PHP | 5,731.88 | 125975 | 0.046 | 34 |
| (i) TypeScript $\downarrow_{4}$ $\Uparrow_{4}$ | 6,898.48 | 516541 | 0.013 | 26 |
| (i) Jruby $\uparrow_{1}$ $\Downarrow_{4}$ | 7,819.03 | 219148 | 0.036 | 669 |
| (i) Lua $\downarrow_{3}$ $\Uparrow_{19}$ | 8,277.87 | 635023 | 0.013 | 2 |
| (i) Perl $\uparrow_{2}$ $\Uparrow_{12}$ | 11,133.49 | 249418 | 0.045 | 12 |
| (i) Python $\uparrow_{2}$ $\Uparrow_{14}$ | 12,784.09 | 279544 | 0.046 | 12 |
| (i) Ruby $\uparrow_{2}$ $\Uparrow_{17}$ | 14,064.98 | 315583 | 0.045 | 8 |

| | fasta | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) Rust $\Downarrow_{9}$ | 26.15 | 931 | 0.028 | 16 |
| (c) Fortran $\downarrow_{6}$ | 27.62 | 1661 | 0.017 | 1 |
| (c) C $\uparrow_{1}$ $\Downarrow_{1}$ | 27.64 | 973 | 0.028 | 3 |
| (c) C++ $\uparrow_{1}$ $\Downarrow_{2}$ | 34.88 | 1164 | 0.030 | 4 |
| (v) Java $\uparrow_{1}$ $\Downarrow_{12}$ | 35.86 | 1249 | 0.029 | 41 |
| (c) Swift $\Downarrow_{9}$ | 37.06 | 1405 | 0.026 | 31 |
| (c) Go $\downarrow_{2}$ | 40.45 | 1838 | 0.022 | 4 |
| (c) Ada $\downarrow_{2}$ $\Uparrow_{3}$ | 40.45 | 2765 | 0.015 | 3 |
| (c) Ocaml $\downarrow_{2}$ $\Downarrow_{15}$ | 40.78 | 3171 | 0.013 | 201 |
| (c) Chapel $\uparrow_{5}$ $\Downarrow_{10}$ | 40.88 | 1379 | 0.030 | 53 |
| (v) C# $\uparrow_{4}$ $\Downarrow_{5}$ | 45.35 | 1549 | 0.029 | 35 |
| (i) Dart $\Downarrow_{6}$ | 63.61 | 4787 | 0.013 | 49 |
| (i) JavaScript $\Downarrow_{1}$ | 64.84 | 5098 | 0.013 | 30 |
| (c) Pascal $\downarrow_{1}$ $\Uparrow_{13}$ | 68.63 | 5478 | 0.013 | 0 |
| (i) TypeScript $\downarrow_{2}$ $\Downarrow_{10}$ | 82.72 | 6909 | 0.012 | 271 |
| (v) F# $\uparrow_{2}$ $\Uparrow_{3}$ | 93.11 | 5360 | 0.017 | 27 |
| (v) Racket $\downarrow_{1}$ $\Uparrow_{5}$ | 120.90 | 8255 | 0.015 | 21 |
| (c) Haskell $\uparrow_{2}$ $\Downarrow_{8}$ | 205.52 | 5728 | 0.036 | 446 |
| (v) Lisp $\Downarrow_{2}$ | 231.49 | 15763 | 0.015 | 75 |
| (i) Hack $\Downarrow_{3}$ | 237.70 | 17203 | 0.014 | 120 |
| (i) Lua $\Uparrow_{18}$ | 347.37 | 24617 | 0.014 | 3 |
| (i) PHP $\downarrow_{1}$ $\Uparrow_{13}$ | 430.73 | 29508 | 0.015 | 14 |
| (v) Erlang $\uparrow_{1}$ $\Uparrow_{12}$ | 477.81 | 27852 | 0.017 | 18 |
| (i) Ruby $\downarrow_{1}$ $\Uparrow_{2}$ | 852.30 | 61216 | 0.014 | 104 |
| (i) JRuby $\uparrow_{1}$ $\Downarrow_{2}$ | 912.93 | 49509 | 0.018 | 705 |
| (i) Python $\downarrow_{1}$ $\Uparrow_{18}$ | 1,061.41 | 74111 | 0.014 | 9 |
| (i) Perl $\uparrow_{1}$ $\Uparrow_{8}$ | 2,684.33 | 61463 | 0.044 | 53 |

Table A.3: Results for k-nucleotide, mandelbrot, and n-body

| | k-nucleotide | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) Rust $\Downarrow_2$ | 72.10 | 2353 | 0.031 | 155 |
| (c) C $\Uparrow_1$ | 89.57 | 2958 | 0.030 | 127 |
| (c) C++ $\Downarrow_1$ | 110.87 | 3802 | 0.029 | 160 |
| (v) Java $\Downarrow_{13}$ | 138.91 | 4116 | 0.034 | 542 |
| (c) Ada $\downarrow_1$ $\Downarrow_4$ | 149.75 | 7142 | 0.021 | 274 |
| (v) C# $\uparrow_1$ $\Uparrow_1$ | 210.22 | 7139 | 0.029 | 175 |
| (c) Chapel $\Downarrow_6$ | 262.32 | 7919 | 0.033 | 360 |
| (c) Go $\Uparrow_6$ | 270.38 | 8004 | 0.034 | 146 |
| (c) Swift $\Uparrow_2$ | 281.72 | 8497 | 0.033 | 256 |
| (c) Ocaml $\Downarrow_4$ | 334.39 | 13847 | 0.024 | 365 |
| (v) F# $\downarrow_1$ $\Downarrow_8$ | 405.29 | 14533 | 0.028 | 828 |
| (i) Dart $\uparrow_1$ $\Downarrow_{11}$ | 426.67 | 14500 | 0.029 | 2203 |
| (c) Fortran $\downarrow_5$ $\Uparrow_7$ | 500.44 | 41656 | 0.012 | 198 |
| (v) Racket $\downarrow_5$ $\Uparrow_2$ | 619.08 | 44239 | 0.014 | 358 |
| (i) Hack $\uparrow_2$ $\Uparrow_4$ | 899.18 | 26385 | 0.034 | 357 |
| (i) PHP $\uparrow_2$ $\Uparrow_6$ | 914.90 | 29324 | 0.031 | 274 |
| (i) JavaScript $\uparrow_2$ $\Uparrow_1$ | 1,106.59 | 34263 | 0.032 | 495 |
| (i) Lua $\downarrow_4$ | 1,196.70 | 88075 | 0.014 | 582 |
| (i) Perl $\uparrow_3$ $\Downarrow_2$ | 1,331.66 | 35616 | 0.037 | 1106 |
| (i) Python $\uparrow_3$ $\Uparrow_{12}$ | 1,601.23 | 39047 | 0.041 | 257 |
| (i) Ruby $\uparrow_1$ $\Uparrow_6$ | 2,354.49 | 60541 | 0.039 | 372 |
| (v) Erlang $\uparrow_1$ $\Uparrow_2$ | 2,581.83 | 86070 | 0.030 | 1091 |
| (i) Jruby $\Uparrow_1$ | 2,594.58 | 88523 | 0.029 | 1559 |
| (c) Haskell | n.a. | | | |
| (c) Lisp | n.e. | | | |
| (i) Pascal | n.a. | | | |
| (v) TypeScript | n.e. | | | |

| | mandelbrot | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) C++ $\Downarrow_4$ | 32.44 | 864 | 0.038 | 34 |
| (c) C $\Downarrow_1$ | 35.39 | 1142 | 0.031 | 32 |
| (c) Rust $\Downarrow_8$ | 41.13 | 1179 | 0.035 | 48 |
| (c) Swift $\Downarrow_5$ | 72.20 | 1799 | 0.040 | 38 |
| (v) Lisp $\downarrow_2$ $\Downarrow_8$ | 92.86 | 3385 | 0.027 | 62 |
| (c) Ada | 97.59 | 3192 | 0.031 | 35 |
| (c) Chapel $\uparrow_2$ $\Downarrow_{14}$ | 99.98 | 2449 | 0.041 | 122 |
| (v) Java $\downarrow_1$ $\Downarrow_8$ | 114.23 | 3657 | 0.031 | 68 |
| (c) Go $\uparrow_1$ $\Uparrow_2$ | 115.89 | 3451 | 0.034 | 36 |
| (v) C# $\Downarrow_2$ | 150.09 | 3961 | 0.038 | 60 |
| (c) Fortran $\downarrow_5$ $\Downarrow_4$ | 167.50 | 8633 | 0.019 | 64 |
| (c) Haskell $\uparrow_1$ $\Uparrow_2$ | 169.60 | 6028 | 0.028 | 44 |
| (v) F# $\Downarrow_1$ | 197.21 | 6877 | 0.029 | 63 |
| (c) Ocaml $\uparrow_2$ $\Uparrow_{13}$ | 211.18 | 6863 | 0.031 | 3 |
| (c) Pascal $\uparrow_1$ $\Uparrow_{11}$ | 212.67 | 7190 | 0.030 | 32 |
| (i) Dart $\downarrow_1$ $\Downarrow_4$ | 214.74 | 10046 | 0.021 | 118 |
| (i) JavaScript $\uparrow_2$ $\Downarrow_7$ | 233.04 | 8278 | 0.028 | 298 |
| (v) Racket $\downarrow_1$ $\Uparrow_{16}$ | 526.34 | 44517 | 0.012 | 21 |
| (i) Hack $\uparrow_1$ $\Downarrow_3$ | 694.66 | 15296 | 0.045 | 153 |
| (i) PHP $\Uparrow_3$ | 3,374.29 | 73741 | 0.046 | 68 |
| (i) Lua $\Downarrow_2$ | 3,635.69 | 100442 | 0.036 | 276 |
| (i) Python $\Uparrow_{14}$ | 7,790.16 | 163332 | 0.048 | 36 |
| (i) Jruby $\downarrow_1$ $\Downarrow_3$ | 8,401.95 | 217118 | 0.039 | 2259 |
| (v) Erlang $\uparrow_1$ $\Downarrow_1$ | 9,172.72 | 194726 | 0.047 | 1278 |
| (i) Ruby $\Uparrow_7$ | 11,050.98 | 245248 | 0.045 | 71 |
| (i) Perl $\Uparrow_7$ | 16,965.06 | 389587 | 0.044 | 74 |
| (i) TypeScript | n.e. | | | |

| | n-body | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) Rust $\Downarrow_9$ | 37.79 | 3328 | 0.011 | 6 |
| (c) Fortran | 50.34 | 3581 | 0.014 | 1 |
| (c) Ada $\downarrow_1$ $\Downarrow_5$ | 51.79 | 4098 | 0.013 | 3 |
| (c) C++ $\uparrow_1$ | 57.30 | 3770 | 0.015 | 2 |
| (c) C $\Uparrow_2$ | 59.45 | 4190 | 0.014 | 2 |
| (c) Chapel $\Downarrow_{18}$ | 60.16 | 5203 | 0.012 | 42 |
| (c) Pascal $\Uparrow_6$ | 64.87 | 5702 | 0.011 | 1 |
| (v) Java $\Downarrow_{14}$ | 65.15 | 5839 | 0.011 | 30 |
| (c) Ocaml $\Uparrow_3$ | 65.75 | 5857 | 0.011 | 2 |
| (c) Go $\Uparrow_5$ | 67.23 | 5899 | 0.011 | 2 |
| (v) C# $\downarrow_1$ $\Downarrow_8$ | 68.16 | 6117 | 0.011 | 26 |
| (c) Swift $\uparrow_1$ | 73.06 | 6036 | 0.012 | 7 |
| (v) Lisp $\Downarrow_3$ | 75.25 | 6685 | 0.011 | 15 |
| (i) JavaScript $\Downarrow_6$ | 78.74 | 6763 | 0.012 | 27 |
| (v) F# $\downarrow_2$ $\Downarrow_8$ | 79.41 | 7105 | 0.011 | 32 |
| (i) TypeScript $\Downarrow_5$ | 80.11 | 6861 | 0.012 | 28 |
| (i) Dart $\uparrow_2$ $\Downarrow_8$ | 84.33 | 6827 | 0.012 | 47 |
| (c) Haskell $\Uparrow_7$ | 127.82 | 10037 | 0.013 | 6 |
| (v) Racket $\Uparrow_1$ | 281.87 | 22260 | 0.013 | 21 |
| (i) Jruby $\Downarrow_7$ | 1,889.85 | 98407 | 0.019 | 689 |
| (v) Erlang $\Uparrow_4$ | 1,998.66 | 150698 | 0.013 | 18 |
| (i) PHP $\downarrow_1$ $\Uparrow_7$ | 2,468.30 | 179360 | 0.014 | 14 |
| (i) Hack $\downarrow_1$ $\Downarrow_3$ | 3,399.70 | 243356 | 0.014 | 123 |
| (i) Ruby $\downarrow_1$ $\Uparrow_{11}$ | 3,870.67 | 281470 | 0.014 | 8 |
| (i) Lua $\uparrow_3$ $\Uparrow_{18}$ | 3,981.88 | 177251 | 0.022 | 2 |
| (i) Perl $\Uparrow_{17}$ | 4,663.48 | 335391 | 0.014 | 5 |
| (i) Python $\Uparrow_{13}$ | 7,879.92 | 559214 | 0.014 | 8 |

Table A.4: Results for pidigits, regex-redux, and spectral-norm

| | pidigits | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) C $\Downarrow_2$ | 7.51 | 546 | 0.014 | 2 |
| (c) Ada $\Downarrow_3$ | 7.73 | 546 | 0.014 | 4 |
| (c) Pascal $\Uparrow_1$ | 7.79 | 547 | 0.014 | 2 |
| (c) C++ | 7.80 | 549 | 0.014 | 4 |
| (c) Rust $\Downarrow_2$ | 8.11 | 559 | 0.015 | 8 |
| (v) Racket $\Downarrow_4$ | 10.21 | 732 | 0.014 | 20 |
| (c) Go $\Downarrow_1$ | 13.14 | 773 | 0.017 | 9 |
| (v) C# $\Downarrow_3$ | 13.73 | 944 | 0.015 | 28 |
| (v) F# $\Downarrow_4$ | 14.19 | 953 | 0.015 | 30 |
| (i) Python $\Uparrow_1$ | 17.58 | 1191 | 0.015 | 11 |
| (i) Perl $\Uparrow_5$ | 19.27 | 1307 | 0.015 | 7 |
| (v) Lisp $\Downarrow_1$ | 35.89 | 2467 | 0.015 | 125 |
| (v) Erlang $\Uparrow_1$ | 102.39 | 7362 | 0.014 | 29 |
| (i) Jruby | 147.81 | 8198 | 0.018 | 724 |
| (c) Haskell | | n.a. | | |
| (c) Ocaml | | n.e. | | |
| (c) Fortran | | n.e. | | |
| (c) Chapel | | n.e. | | |
| (c) Swift | | n.e. | | |
| (i) Dart | | n.a. | | |
| (i) Ruby | | n.e. | | |
| (i) Lua | | n.e. | | |
| (i) Hack | | n.a. | | |
| (i) JavaScript | | n.a. | | |
| (i) PHP | | n.e. | | |
| (i) TypeScript | | n.a. | | |
| (v) Java | | n.e. | | |

| | regex-redux | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (i) TypeScript $\downarrow_2$ $\Downarrow_{12}$ | 24.65 | 2009 | 0.012 | 587 |
| (c) C $\uparrow_1$ | 24.83 | 805 | 0.031 | 151 |
| (i) JavaScript $\downarrow_2$ $\Downarrow_9$ | 25.68 | 2096 | 0.012 | 525 |
| (i) PHP $\uparrow_2$ $\Downarrow_1$ | 34.57 | 1667 | 0.021 | 182 |
| (c) Pascal $\downarrow_1$ $\Uparrow_4$ | 35.20 | 2282 | 0.015 | 106 |
| (i) Hack $\uparrow_2$ $\Downarrow_2$ | 38.96 | 2052 | 0.019 | 268 |
| (c) Rust | 40.26 | 2287 | 0.018 | 218 |
| (c) Chapel $\Downarrow_{11}$ | 97.19 | 4534 | 0.021 | 1055 |
| (c) Ada $\Uparrow_5$ | 148.66 | 5157 | 0.029 | 157 |
| (i) Python $\downarrow_1$ | 161.62 | 7116 | 0.023 | 429 |
| (c) Ocaml $\downarrow_3$ $\Downarrow_6$ | 172.43 | 12978 | 0.013 | 948 |
| (c) C++ $\downarrow_1$ $\Uparrow_6$ | 176.24 | 10656 | 0.017 | 216 |
| (i) Ruby $\downarrow_4$ $\Uparrow_4$ | 192.88 | 14282 | 0.014 | 305 |
| (v) Java $\uparrow_4$ $\Downarrow_6$ | 194.65 | 5694 | 0.034 | 1225 |
| (i) Dart $\downarrow_1$ $\Uparrow_4$ | 197.92 | 13485 | 0.015 | 459 |
| (i) Perl $\uparrow_4$ $\Uparrow_{13}$ | 236.24 | 7164 | 0.033 | 154 |
| (i) Jruby $\uparrow_2$ $\Downarrow_4$ | 348.44 | 13477 | 0.026 | 1369 |
| (v) Racket $\downarrow_1$ | 358.20 | 26152 | 0.014 | 983 |
| (v) C# $\uparrow_1$ $\Uparrow_3$ | 522.59 | 14723 | 0.035 | 851 |
| (c) Swift $\Uparrow_5$ | 538.11 | 41703 | 0.013 | 677 |
| (v) F# $\Uparrow_7$ | 650.51 | 46905 | 0.014 | 667 |
| (c) Haskell | | n.a. | | |
| (c) Fortran | | n.a. | | |
| (c) Go | | n.e. | | |
| (i) Lua | | n.e. | | |
| (v) Erlang | | n.a. | | |
| (v) Lisp | | n.e. | | |

| | spectral-norm | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) C++ $\Downarrow_2$ | 17.23 | 665 | 0.026 | 2 |
| (c) Fortran $\Downarrow_3$ | 20.98 | 667 | 0.031 | 3 |
| (c) C $\Uparrow_1$ | 21.66 | 677 | 0.032 | 2 |
| (c) Rust $\Downarrow_{11}$ | 24.37 | 677 | 0.036 | 16 |
| (c) Ada $\Downarrow_2$ | 27.03 | 775 | 0.035 | 4 |
| (c) Haskell $\downarrow_5$ $\Downarrow_3$ | 34.74 | 1373 | 0.025 | 6 |
| (c) Swift $\Downarrow_3$ | 35.30 | 1345 | 0.026 | 9 |
| (c) Go $\uparrow_2$ $\Uparrow_2$ | 35.38 | 1332 | 0.027 | 3 |
| (v) C# $\downarrow_3$ $\Downarrow_{10}$ | 35.68 | 1374 | 0.026 | 28 |
| (c) Chapel $\uparrow_1$ $\Downarrow_{15}$ | 36.42 | 1352 | 0.027 | 54 |
| (c) Pascal $\uparrow_3$ $\Uparrow_{10}$ | 37.65 | 1349 | 0.028 | 2 |
| (c) Ocaml $\downarrow_3$ $\Uparrow_4$ | 40.65 | 1683 | 0.024 | 5 |
| (v) Java $\downarrow_1$ $\Downarrow_{10}$ | 41.18 | 1659 | 0.025 | 35 |
| (v) Lisp $\uparrow_4$ $\Downarrow_3$ | 42.61 | 1371 | 0.031 | 24 |
| (v) F# $\uparrow_2$ $\Downarrow_7$ | 48.62 | 1416 | 0.034 | 33 |
| (i) JavaScript $\downarrow_1$ $\Downarrow_2$ | 51.46 | 5048 | 0.010 | 28 |
| (i) Dart $\downarrow_1$ $\Downarrow_7$ | 55.15 | 5102 | 0.011 | 52 |
| (i) TypeScript $\downarrow_1$ $\Downarrow_2$ | 62.25 | 5750 | 0.011 | 28 |
| (v) Racket $\uparrow_3$ $\Uparrow_3$ | 84.97 | 2341 | 0.036 | 23 |
| (i) Hack $\Downarrow_6$ | 405.26 | 11652 | 0.035 | 122 |
| (v) Erlang | 767.43 | 17998 | 0.043 | 29 |
| (i) Perl $\downarrow_1$ $\Uparrow_{11}$ | 862.49 | 19782 | 0.044 | 10 |
| (i) PHP $\uparrow_1$ $\Uparrow_{10}$ | 880.45 | 19128 | 0.046 | 14 |
| (i) Lua $\downarrow_2$ $\Uparrow_{20}$ | 1,249.98 | 95406 | 0.013 | 3 |
| (i) Jruby $\Downarrow_2$ | 1,420.10 | 79449 | 0.018 | 677 |
| (i) Ruby $\uparrow_2$ $\Uparrow_{14}$ | 3,122.89 | 71540 | 0.044 | 11 |
| (i) Python $\Uparrow_{13}$ | 6,410.10 | 137738 | 0.047 | 15 |

Table A.5: Results for reverse-complement

| reverse-complement | | | | |
|---|---|---|---|---|
| | Energy | Time | Ratio | Mb |
| (c) C++ $\Downarrow_{9}$ | 6.16 | 224 | 0.028 | 243 |
| (c) C $\Downarrow_{9}$ | 6.52 | 228 | 0.029 | 245 |
| (c) Rust $\Downarrow_{9}$ | 6.96 | 284 | 0.024 | 260 |
| (c) Ada $\downarrow_{2}$ $\Downarrow_{4}$ | 8.31 | 367 | 0.023 | 196 |
| (c) Ocaml $\uparrow_{1}$ $\Uparrow_{4}$ | 8.77 | 287 | 0.031 | 2 |
| (c) Go $\uparrow_{1}$ $\Uparrow_{1}$ | 9.17 | 366 | 0.025 | 133 |
| (c) Swift $\Downarrow_{11}$ | 9.40 | 410 | 0.023 | 407 |
| (c) Fortran $\downarrow_{5}$ $\Downarrow_{1}$ | 12.62 | 938 | 0.013 | 243 |
| (v) C# $\uparrow_{1}$ $\Uparrow_{7}$ | 12.66 | 581 | 0.022 | 60 |
| (c) Haskell $\Uparrow_{7}$ | 13.76 | 808 | 0.017 | 129 |
| (c) Pascal $\Uparrow_{7}$ | 15.05 | 913 | 0.016 | 129 |
| (c) Chapel $\Downarrow_{2}$ | 15.73 | 915 | 0.017 | 299 |
| (v) Java $\uparrow_{4}$ $\Downarrow_{6}$ | 16.97 | 629 | 0.027 | 476 |
| (v) Lisp $\Uparrow_{1}$ | 16.99 | 1135 | 0.015 | 294 |
| (i) Perl | 21.04 | 1187 | 0.018 | 369 |
| (i) PHP $\Uparrow_{10}$ | 26.09 | 1214 | 0.021 | 140 |
| (v) Racket $\downarrow_{3}$ $\Uparrow_{10}$ | 30.52 | 2112 | 0.014 | 177 |
| (i) JavaScript $\Uparrow_{2}$ | 31.73 | 1930 | 0.016 | 380 |
| (i) Python $\uparrow_{2}$ $\Downarrow_{1}$ | 32.31 | 1498 | 0.022 | 620 |
| (i) Ruby $\uparrow_{1}$ $\Downarrow_{3}$ | 42.77 | 1978 | 0.022 | 999 |
| (v) F# $\downarrow_{1}$ $\Uparrow_{4}$ | 74.89 | 5627 | 0.013 | 405 |
| (i) Jruby $\uparrow_{1}$ $\Downarrow_{3}$ | 131.45 | 4533 | 0.029 | 2182 |
| (v) Erlang $\Uparrow_{1}$ | 137.09 | 6508 | 0.021 | 916 |
| (i) Lua $\Uparrow_{3}$ | 142.96 | 9305 | 0.015 | 719 |
| (i) Dart $\Uparrow_{1}$ | 173.60 | 10634 | 0.016 | 1679 |
| (i) Hack | | n.a. | | |
| (i) TypeScript | | n.e. | | |

Table A.6: Normalized global results for Energy, Time, and Memory

| Total | | | | | |
|---|---|---|---|---|---|
| | Energy | | Time | | Mb |
| (c) C | 1.00 | (c) C | 1.00 | (c) Pascal | 1.00 |
| (c) Rust | 1.03 | (c) Rust | 1.04 | (c) Go | 1.05 |
| (c) C++ | 1.34 | (c) C++ | 1.56 | (c) C | 1.17 |
| (c) Ada | 1.70 | (c) Ada | 1.85 | (c) Fortran | 1.24 |
| (v) Java | 1.98 | (v) Java | 1.89 | (c) C++ | 1.34 |
| (c) Pascal | 2.14 | (c) Chapel | 2.14 | (c) Ada | 1.47 |
| (c) Chapel | 2.18 | (c) Go | 2.83 | (c) Rust | 1.54 |
| (v) Lisp | 2.27 | (c) Pascal | 3.02 | (v) Lisp | 1.92 |
| (c) Ocaml | 2.40 | (c) Ocaml | 3.09 | (c) Haskell | 2.45 |
| (c) Fortran | 2.52 | (v) C# | 3.14 | (i) PHP | 2.57 |
| (c) Swift | 2.79 | (v) Lisp | 3.40 | (c) Swift | 2.71 |
| (c) Haskell | 3.10 | (c) Haskell | 3.55 | (i) Python | 2.80 |
| (v) C# | 3.14 | (c) Swift | 4.20 | (c) Ocaml | 2.82 |
| (c) Go | 3.23 | (c) Fortran | 4.20 | (v) C# | 2.85 |
| (i) Dart | 3.83 | (v) F# | 6.30 | (i) Hack | 3.34 |
| (v) F# | 4.13 | (i) JavaScript | 6.52 | (v) Racket | 3.52 |
| (i) JavaScript | 4.45 | (i) Dart | 6.67 | (i) Ruby | 3.97 |
| (v) Racket | 7.91 | (v) Racket | 11.27 | (c) Chapel | 4.00 |
| (i) TypeScript | 21.50 | (i) Hack | 26.99 | (v) F# | 4.25 |
| (i) Hack | 24.02 | (i) PHP | 27.64 | (i) JavaScript | 4.59 |
| (i) PHP | 29.30 | (v) Erlang | 36.71 | (i) TypeScript | 4.69 |
| (v) Erlang | 42.23 | (i) Jruby | 43.44 | (v) Java | 6.01 |
| (i) Lua | 45.98 | (i) TypeScript | 46.20 | (i) Perl | 6.62 |
| (i) Jruby | 46.54 | (i) Ruby | 59.34 | (i) Lua | 6.72 |
| (i) Ruby | 69.91 | (i) Perl | 65.79 | (v) Erlang | 7.20 |
| (i) Python | 75.88 | (i) Python | 71.90 | (i) Dart | 8.64 |
| (i) Perl | 79.58 | (i) Lua | 82.91 | (i) Jruby | 19.84 |

## A.3 Energy and Time Graphs

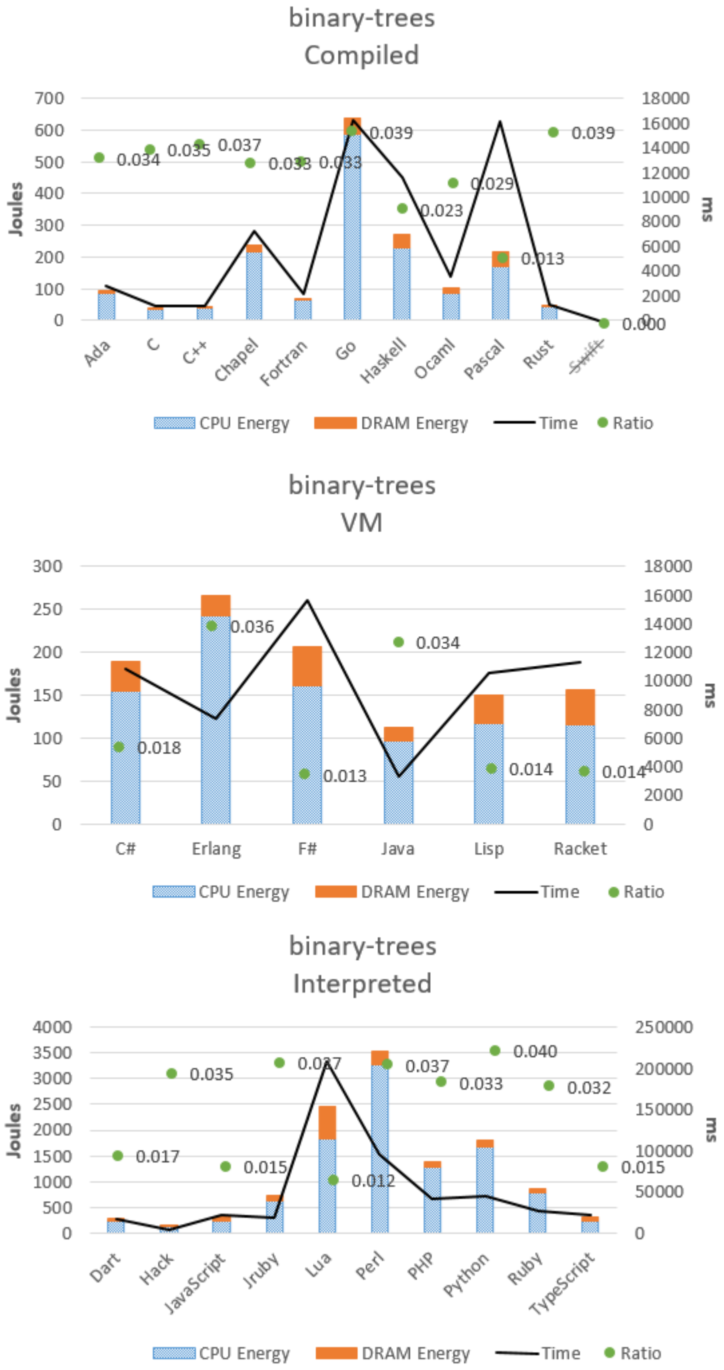


Figure A.1: Energy and time graphical data for binary-trees

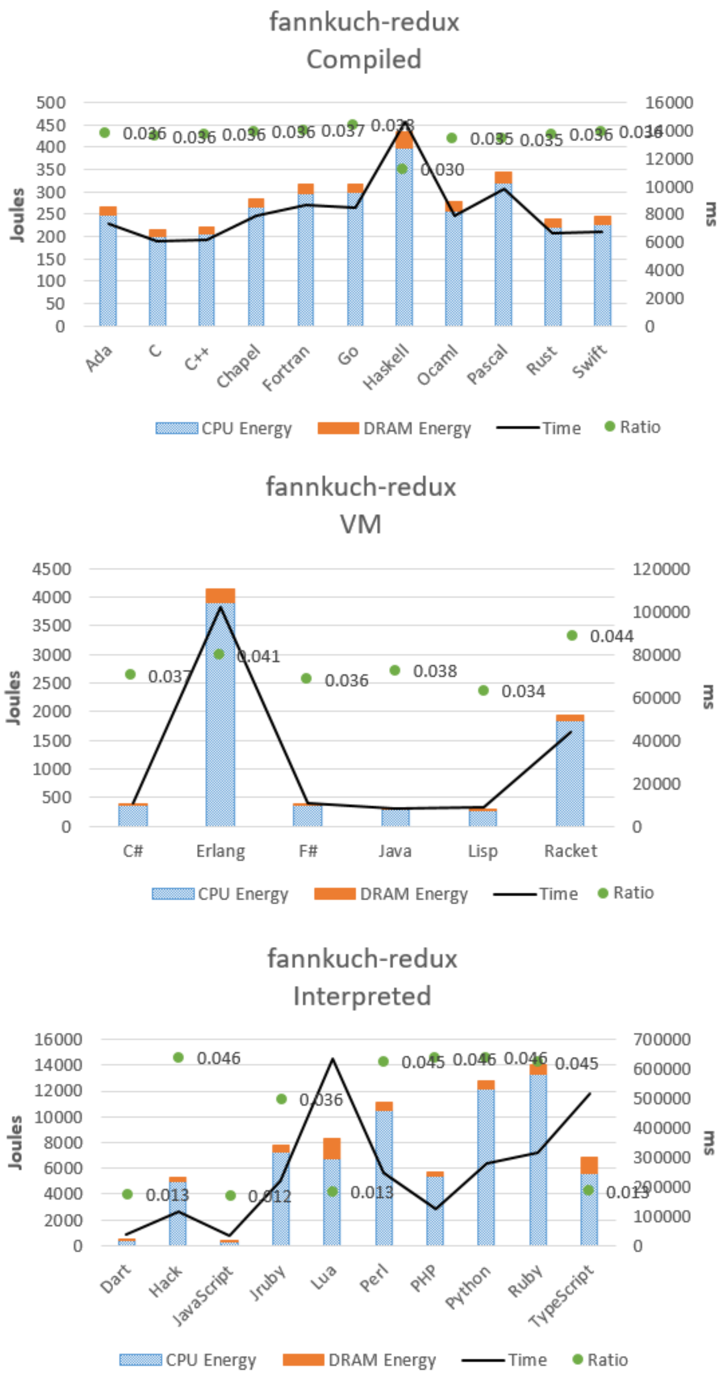


Figure A.2: Energy and time graphical data for fannkuch-redux

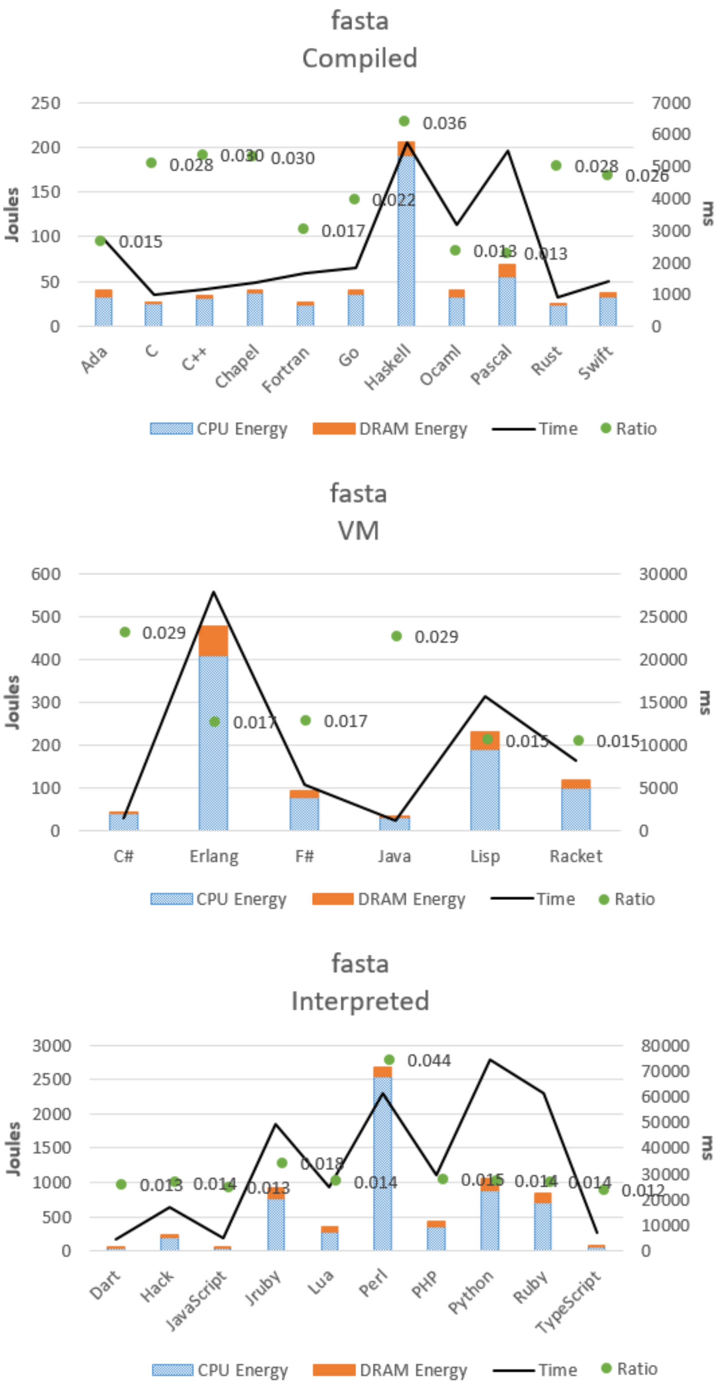


Figure A.3: Energy and time graphical data for fasta

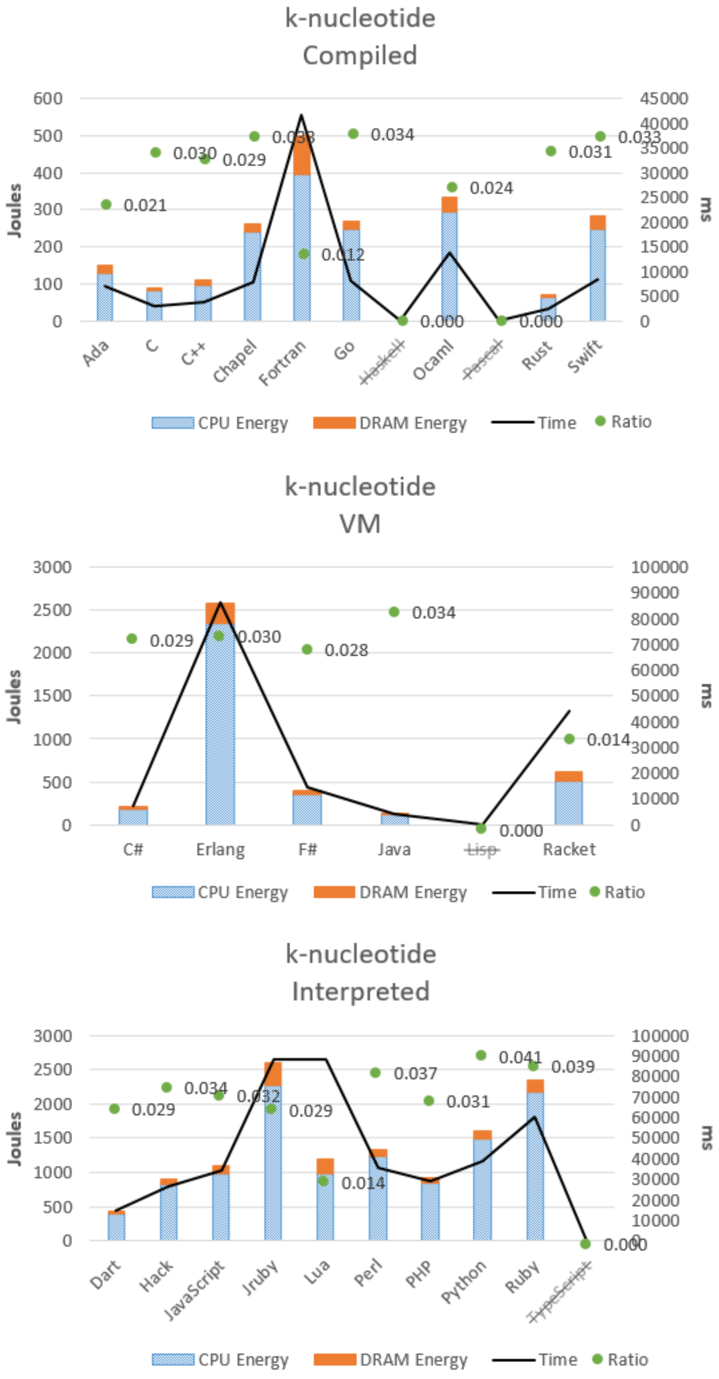


Figure A.4: Energy and time graphical data for k-nucleotide

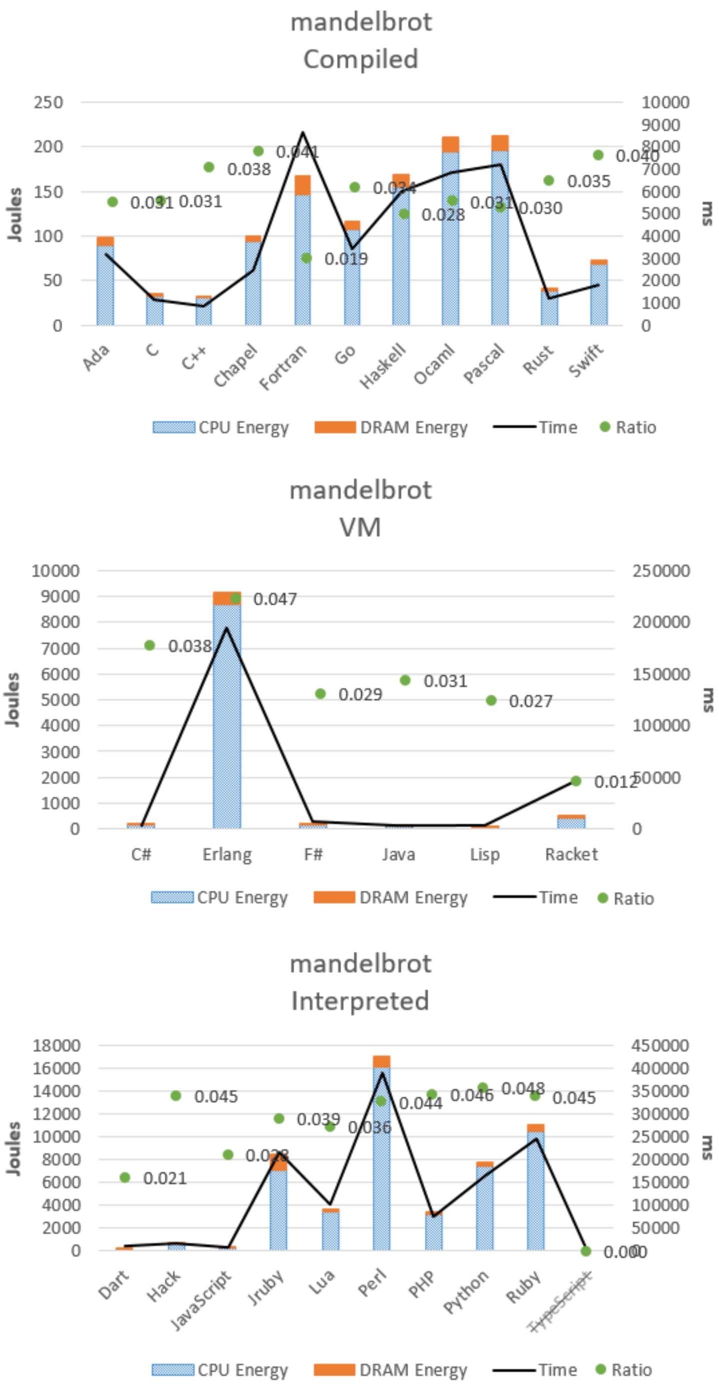


Figure A.5: Energy and time graphical data for mandelbrot

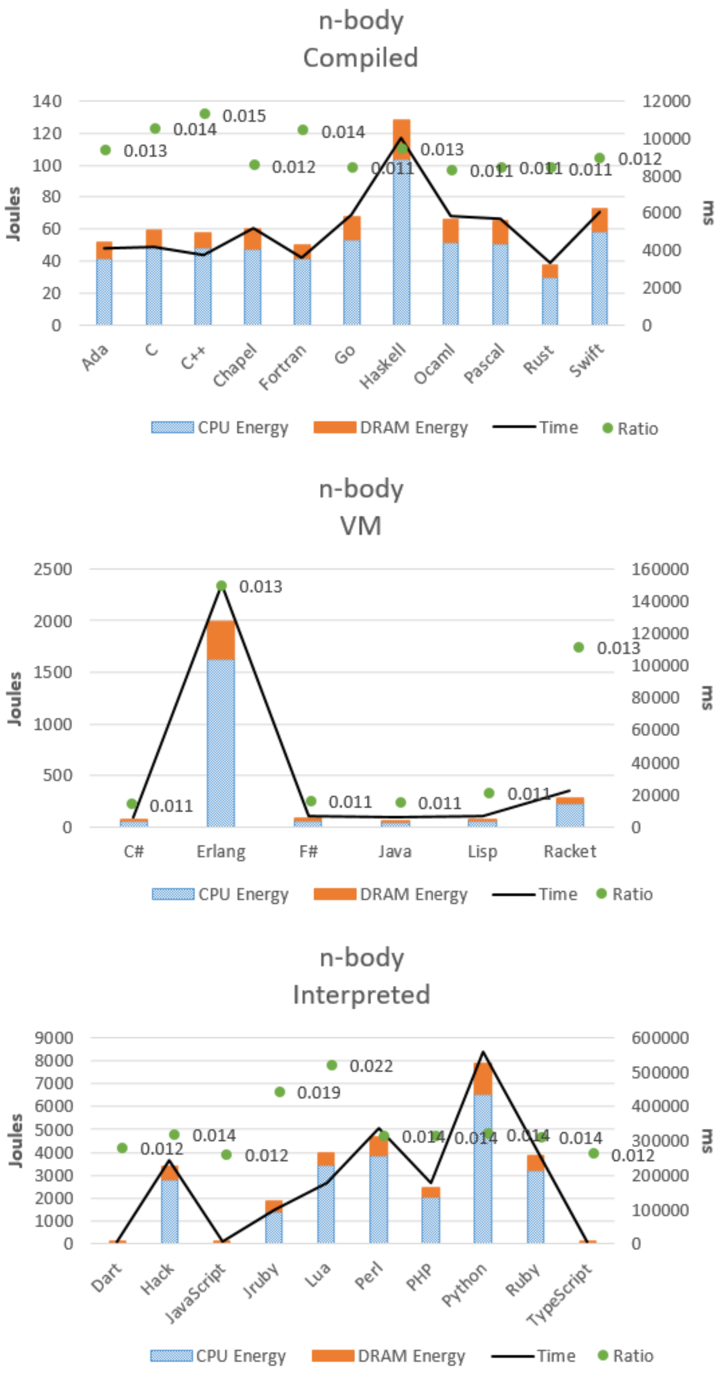


Figure A.6: Energy and time graphical data for n-body

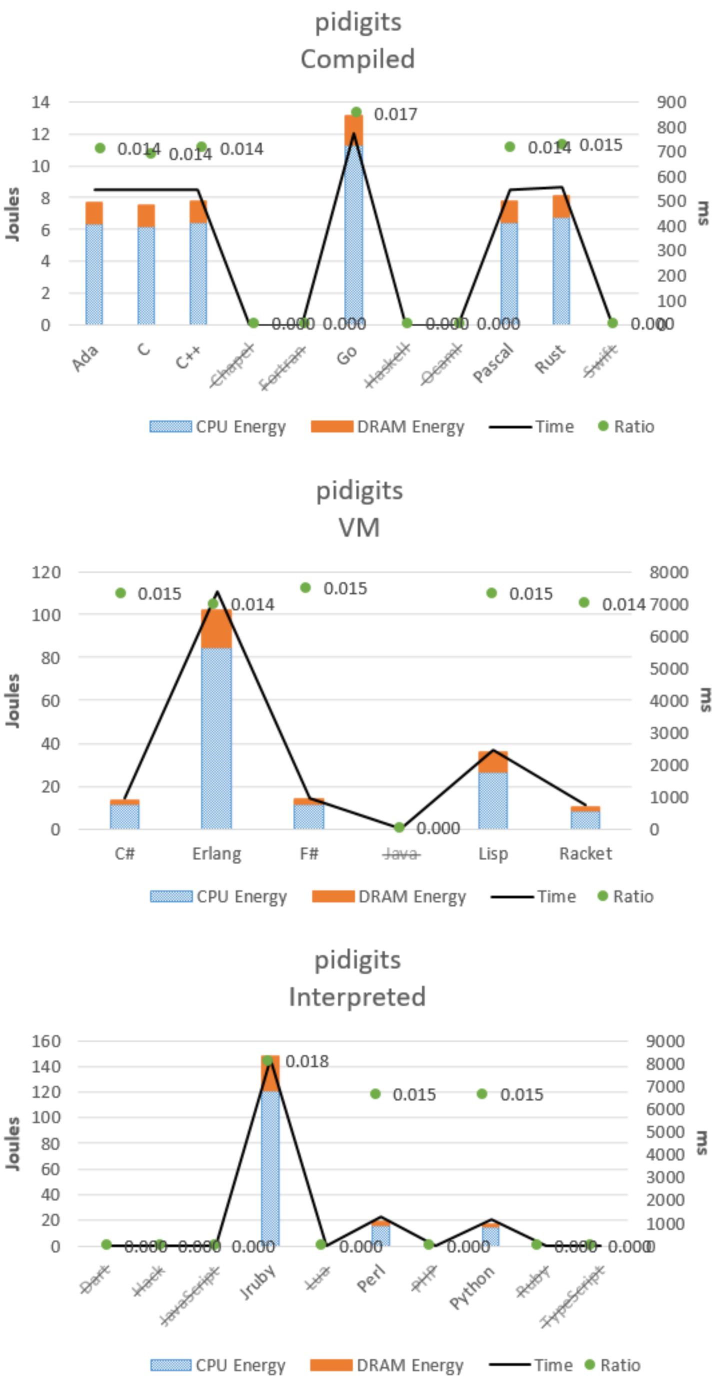


Figure A.7: Energy and time graphical data for pidigits

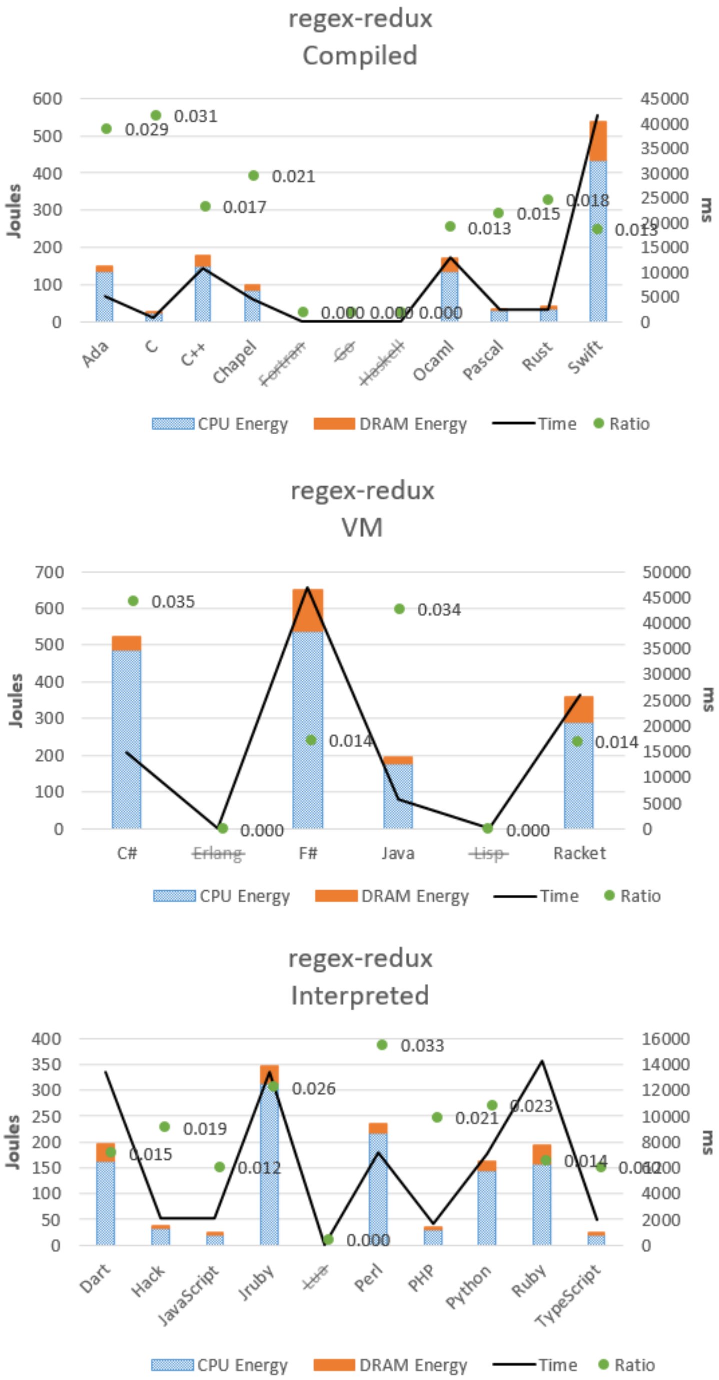


Figure A.8: Energy and time graphical data for regex-redux

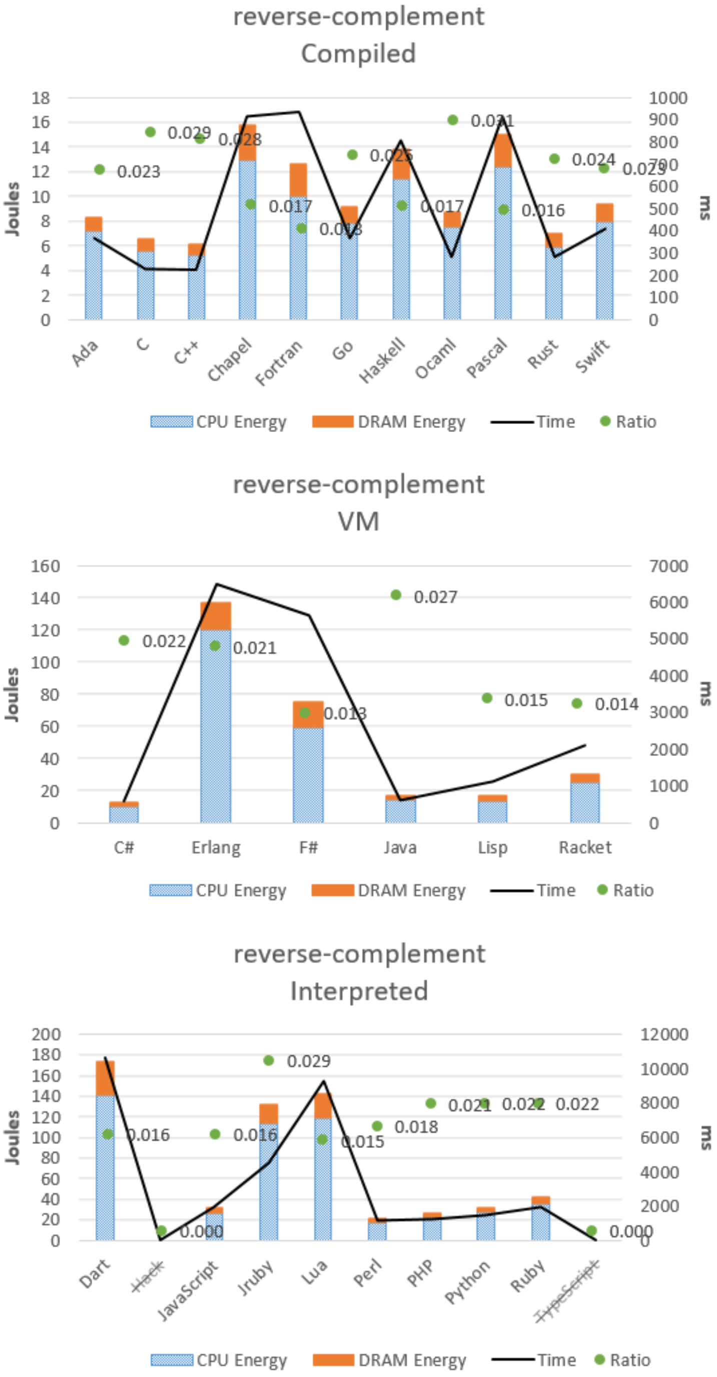


Figure A.9: Energy and time graphical data for reverse-complement

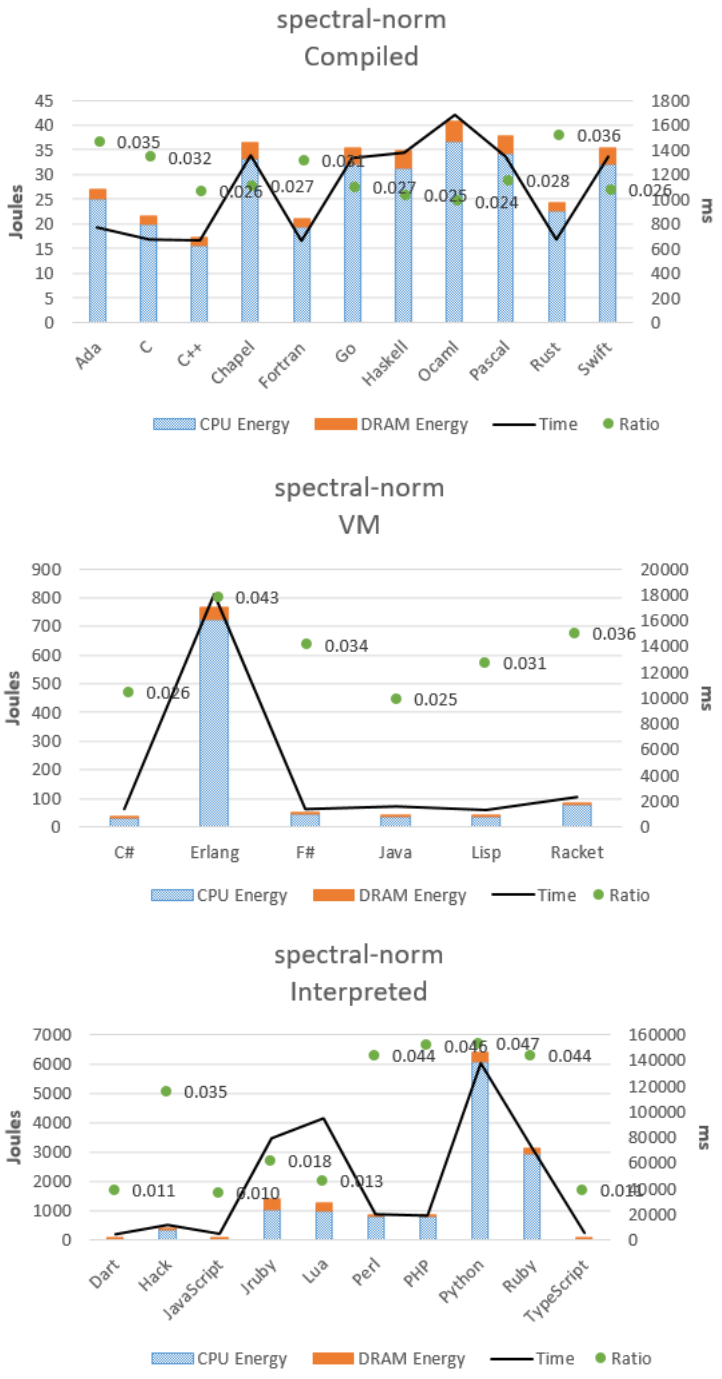


Figure A.10: Energy and time graphical data for spectral-norm

## A.4 Energy and Memory Graphs

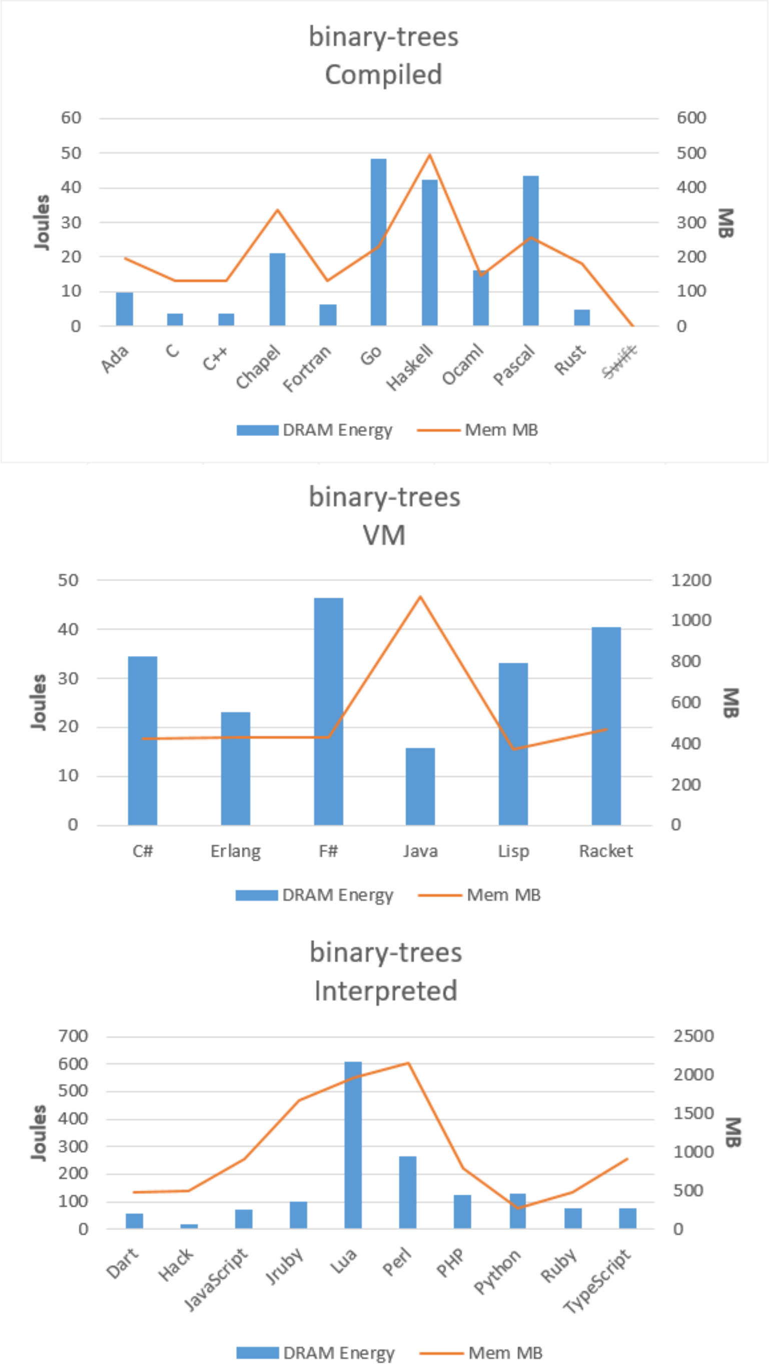


Figure A.11: Graphical DRAM energy and memory data for binary-trees

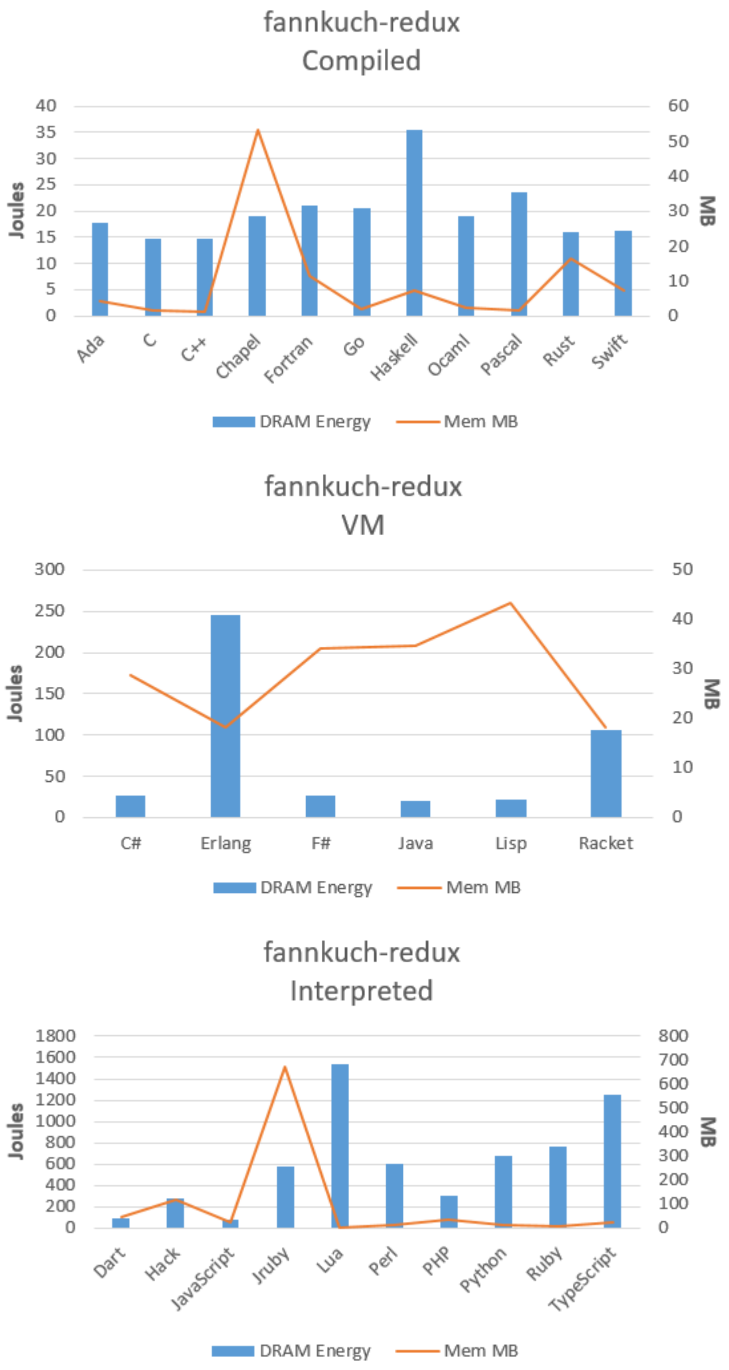


Figure A.12: Graphical DRAM energy and memory data for fannkuch-redux

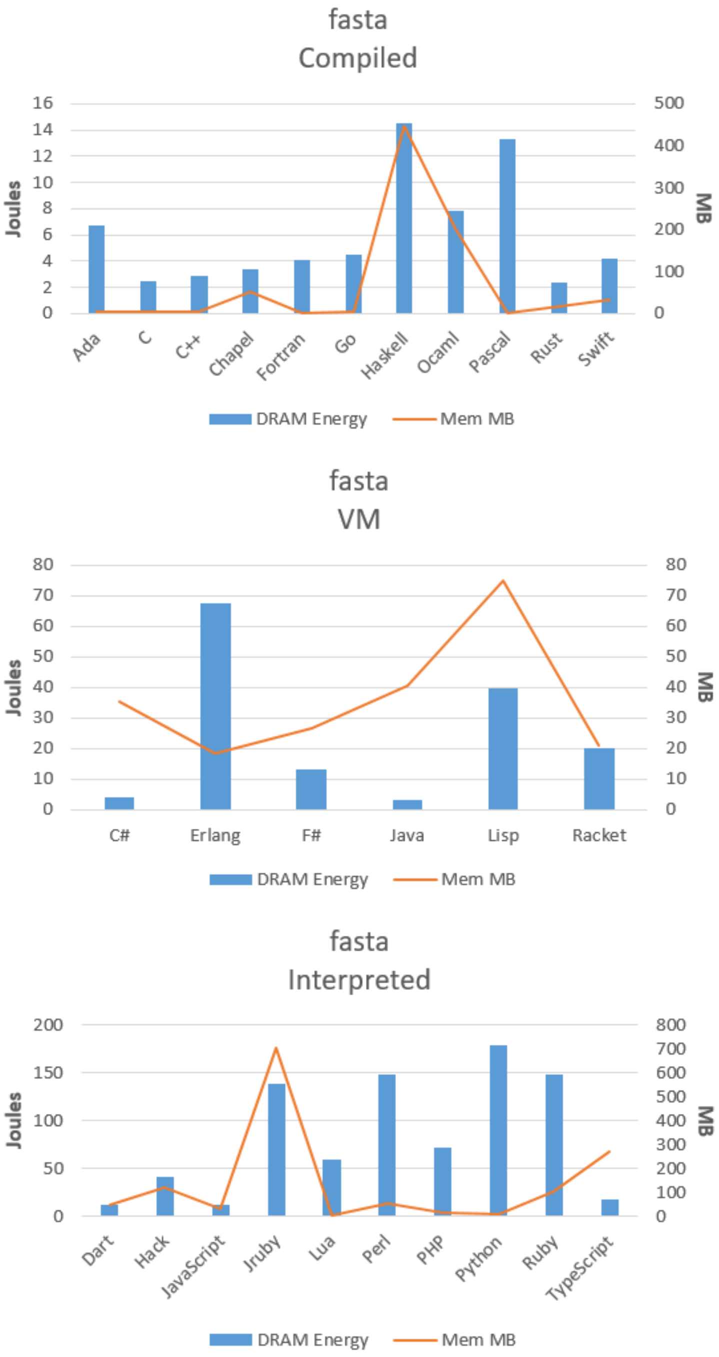


Figure A.13: Graphical DRAM energy and memory data for fasta

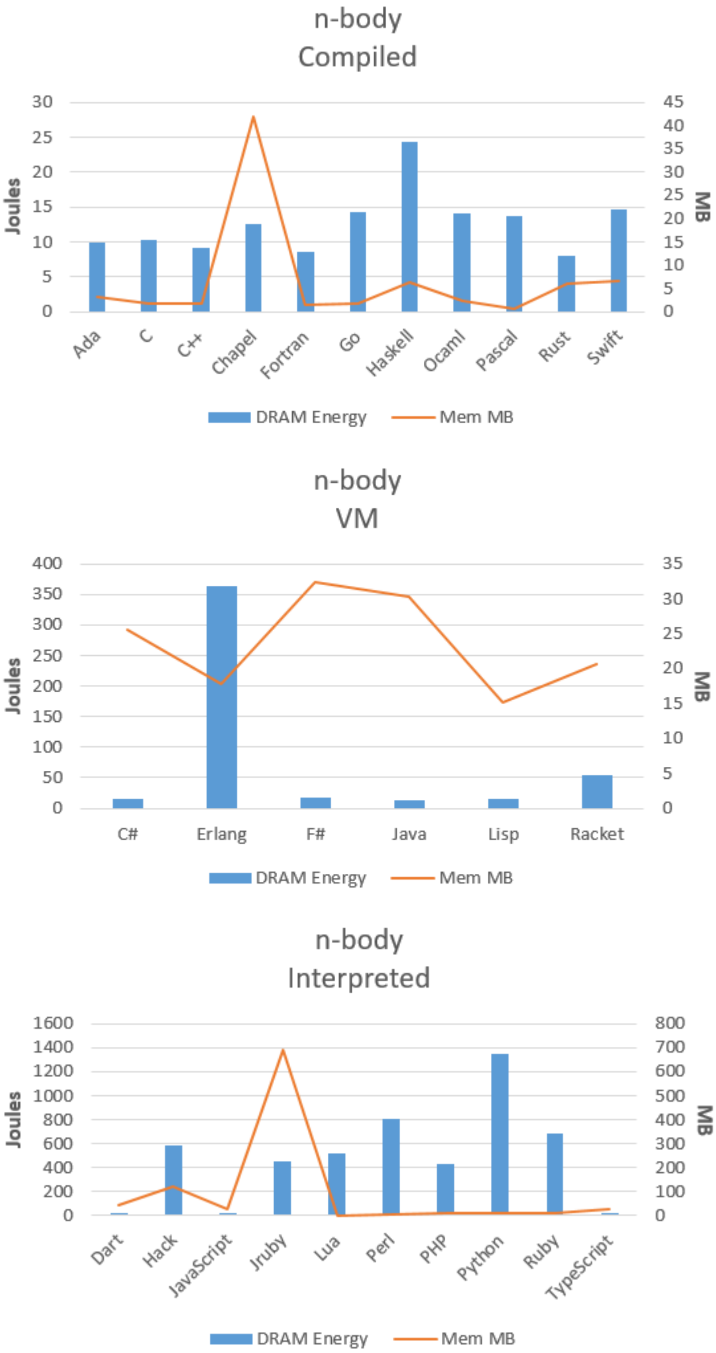


Figure A.14: Graphical DRAM energy and memory data for nbody

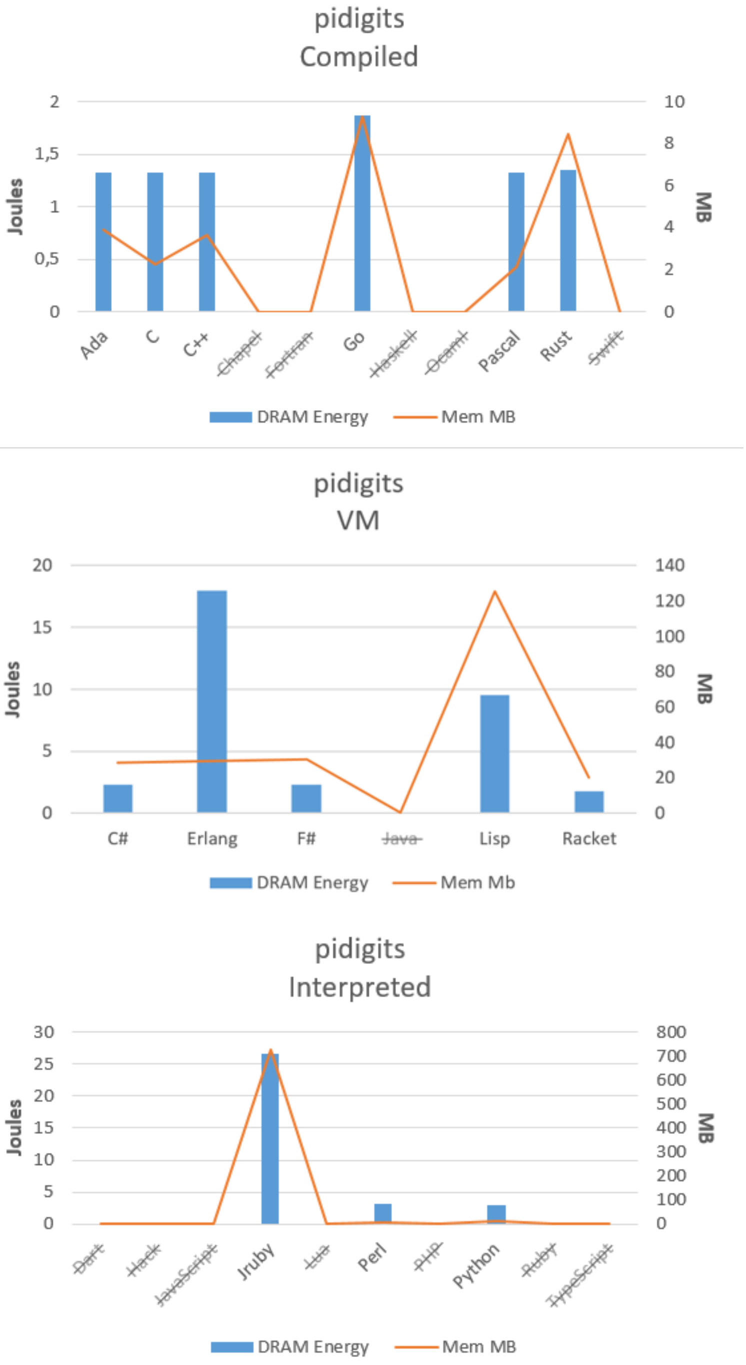


Figure A.15: Graphical DRAM energy and memory data for pidigits

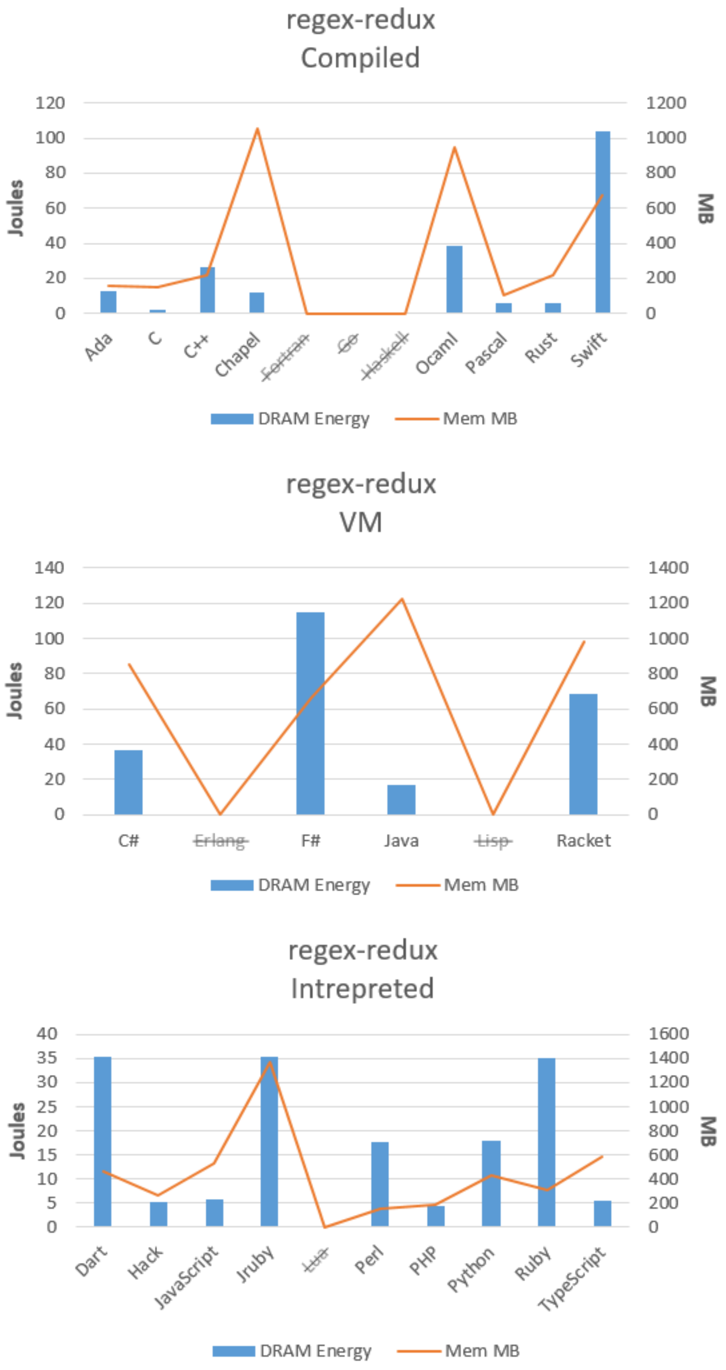


Figure A.16: Graphical DRAM energy and memory data for regex-redux

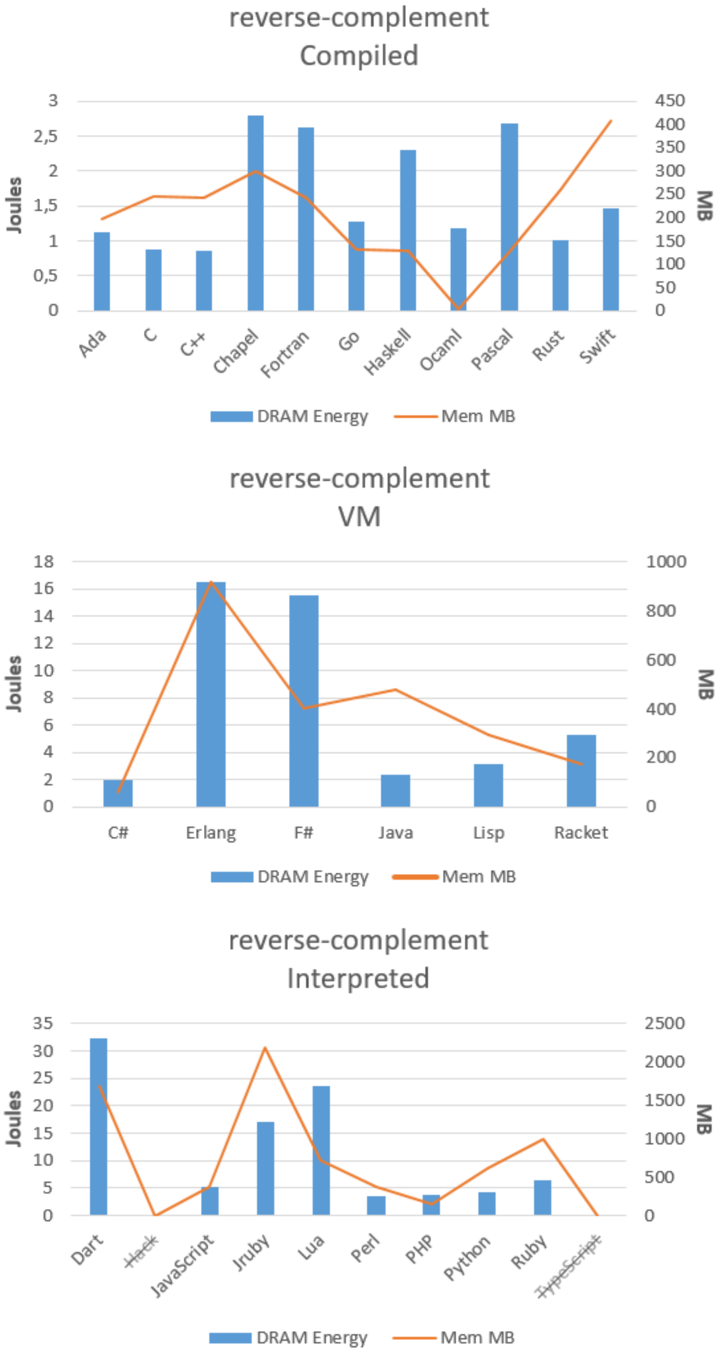


Figure A.17: Graphical DRAM energy and memory data for reverse-complement

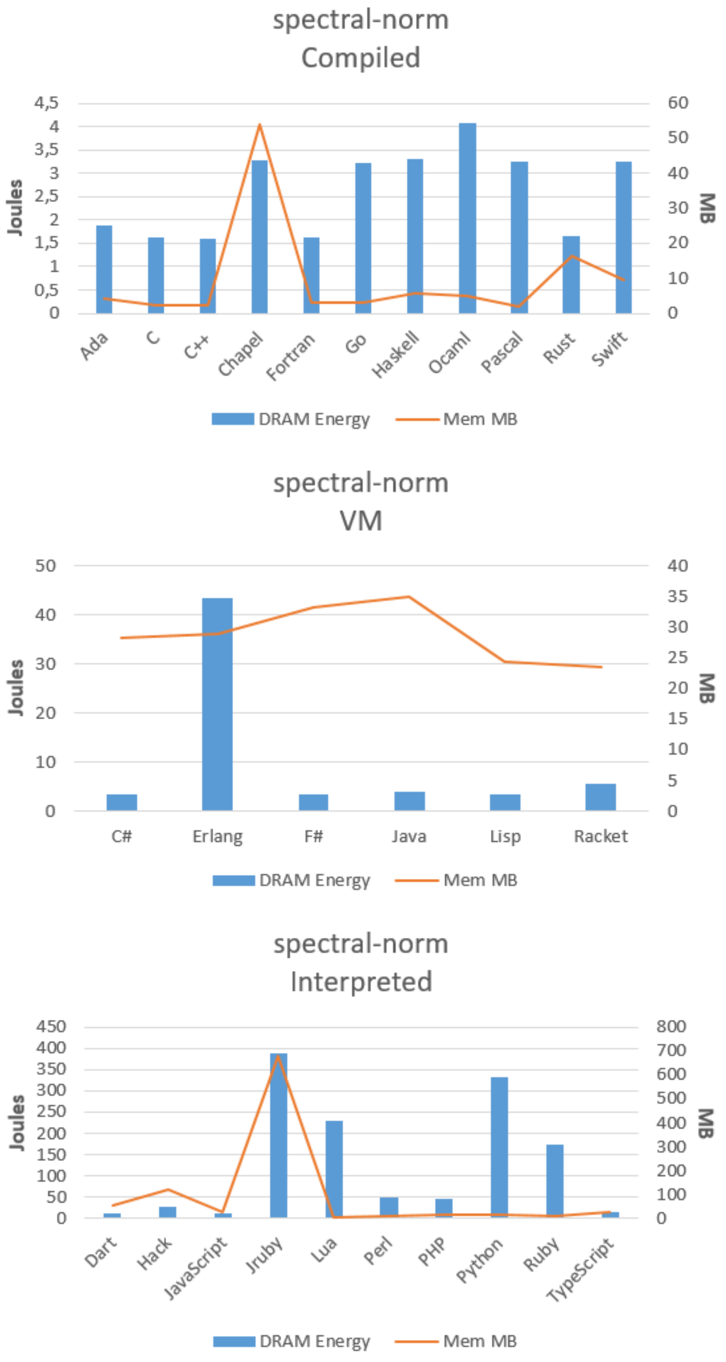
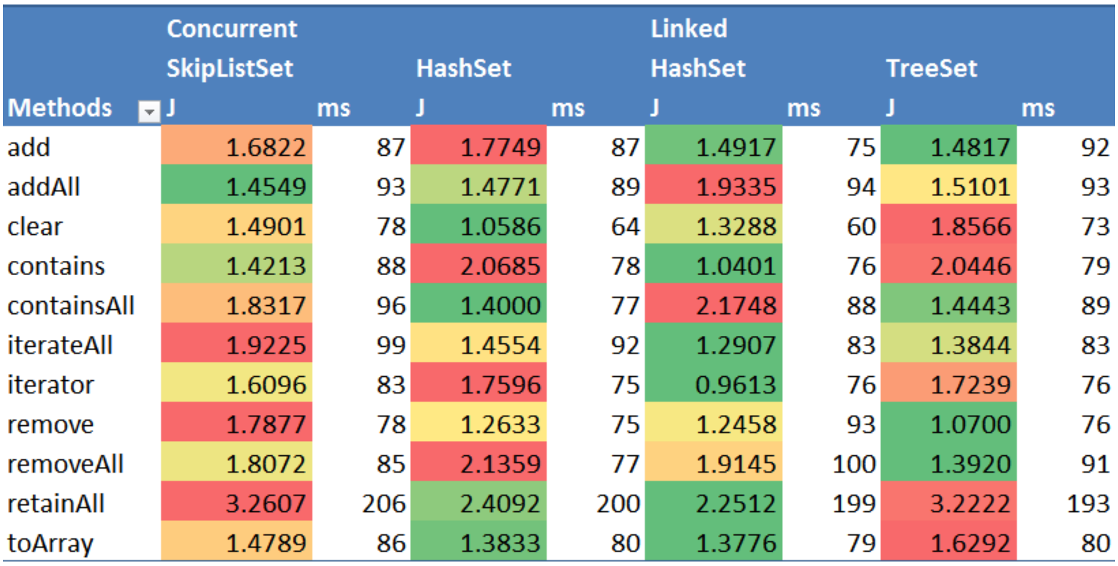


Figure A.18: Graphical DRAM energy and memory data for spectral-norm

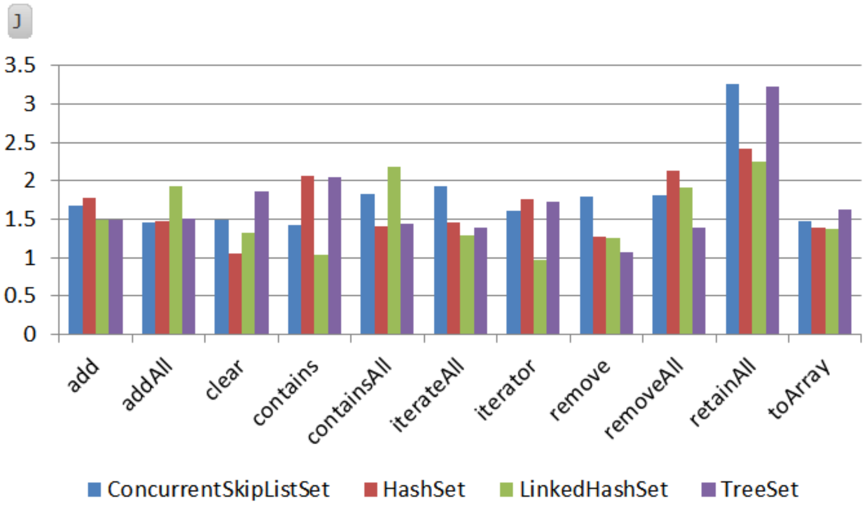
# Appendix B

# Chapter 5 Appendix

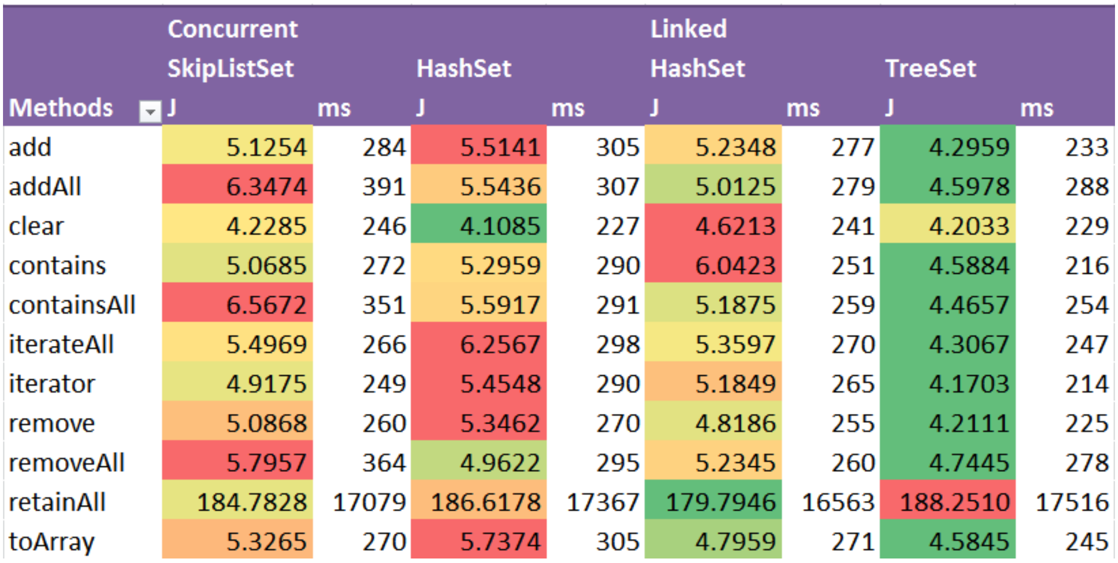
## B.1 Set data for 25k population



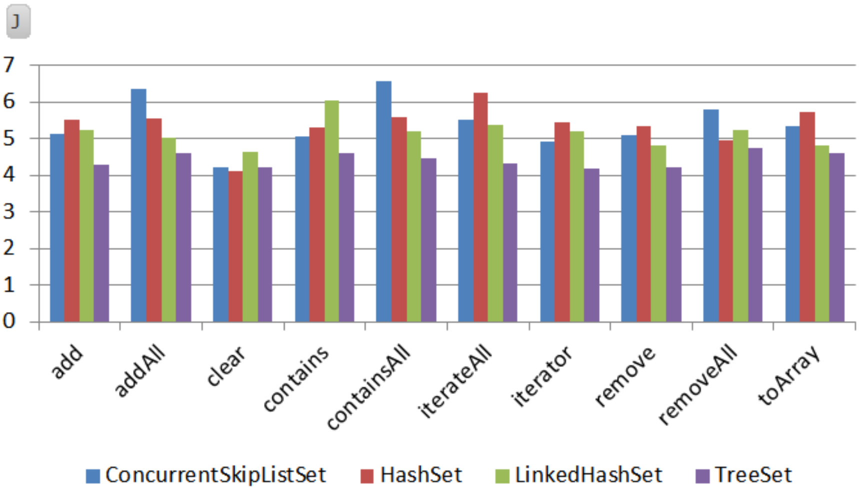
| | Concurrent SkipListSet | | HashSet | | Linked HashSet | | TreeSet | |
|---|---|---|---|---|---|---|---|---|
| Methods | J | ms | J | ms | J | ms | J | ms |
| add | 1.6822 | 87 | 1.7749 | 87 | 1.4917 | 75 | 1.4817 | 92 |
| addAll | 1.4549 | 93 | 1.4771 | 89 | 1.9335 | 94 | 1.5101 | 93 |
| clear | 1.4901 | 78 | 1.0586 | 64 | 1.3288 | 60 | 1.8566 | 73 |
| contains | 1.4213 | 88 | 2.0685 | 78 | 1.0401 | 76 | 2.0446 | 79 |
| containsAll | 1.8317 | 96 | 1.4000 | 77 | 2.1748 | 88 | 1.4443 | 89 |
| iterateAll | 1.9225 | 99 | 1.4554 | 92 | 1.2907 | 83 | 1.3844 | 83 |
| iterator | 1.6096 | 83 | 1.7596 | 75 | 0.9613 | 76 | 1.7239 | 76 |
| remove | 1.7877 | 78 | 1.2633 | 75 | 1.2458 | 93 | 1.0700 | 76 |
| removeAll | 1.8072 | 85 | 2.1359 | 77 | 1.9145 | 100 | 1.3920 | 91 |
| retainAll | 3.2607 | 206 | 2.4092 | 200 | 2.2512 | 199 | 3.2222 | 193 |
| toArray | 1.4789 | 86 | 1.3833 | 80 | 1.3776 | 79 | 1.6292 | 80 |

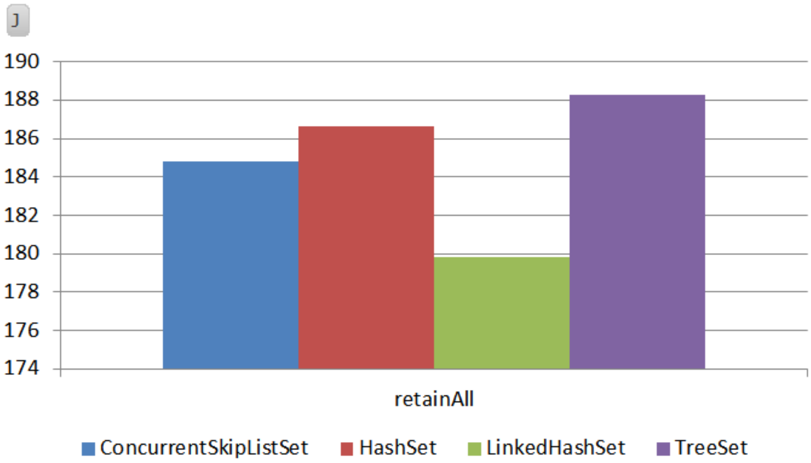


## B.2 Set data for 250k population

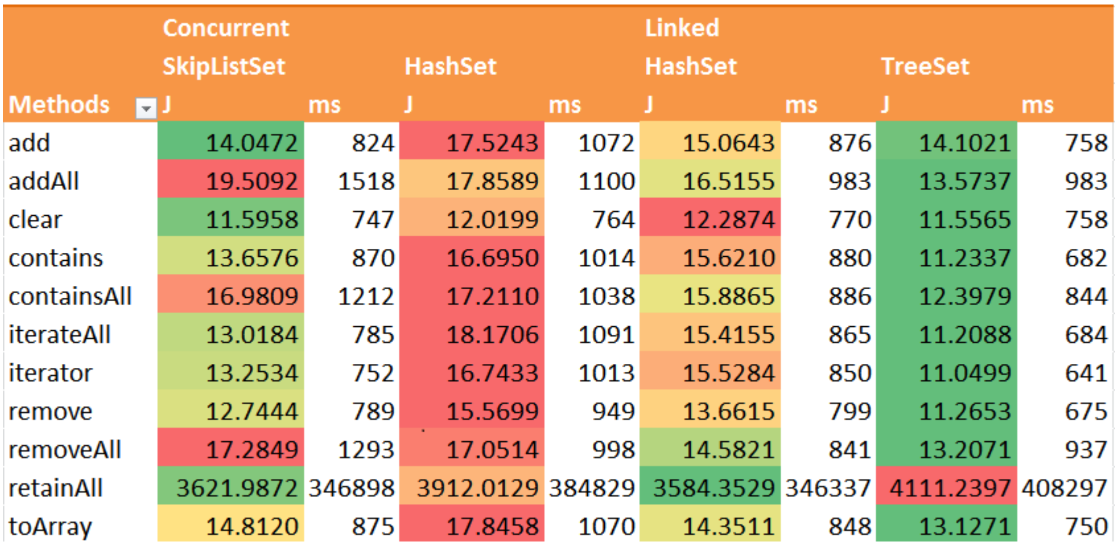


| | Concurrent SkipListSet | | HashSet | | Linked HashSet | | TreeSet | |
|---|---|---|---|---|---|---|---|---|
| Methods | J | ms | J | ms | J | ms | J | ms |
| add | 5.1254 | 284 | 5.5141 | 305 | 5.2348 | 277 | 4.2959 | 233 |
| addAll | 6.3474 | 391 | 5.5436 | 307 | 5.0125 | 279 | 4.5978 | 288 |
| clear | 4.2285 | 246 | 4.1085 | 227 | 4.6213 | 241 | 4.2033 | 229 |
| contains | 5.0685 | 272 | 5.2959 | 290 | 6.0423 | 251 | 4.5884 | 216 |
| containsAll | 6.5672 | 351 | 5.5917 | 291 | 5.1875 | 259 | 4.4657 | 254 |
| iterateAll | 5.4969 | 266 | 6.2567 | 298 | 5.3597 | 270 | 4.3067 | 247 |
| iterator | 4.9175 | 249 | 5.4548 | 290 | 5.1849 | 265 | 4.1703 | 214 |
| remove | 5.0868 | 260 | 5.3462 | 270 | 4.8186 | 255 | 4.2111 | 225 |
| removeAll | 5.7957 | 364 | 4.9622 | 295 | 5.2345 | 260 | 4.7445 | 278 |
| retainAll | 184.7828 | 17079 | 186.6178 | 17367 | 179.7946 | 16563 | 188.2510 | 17516 |
| toArray | 5.3265 | 270 | 5.7374 | 305 | 4.7959 | 271 | 4.5845 | 245 |

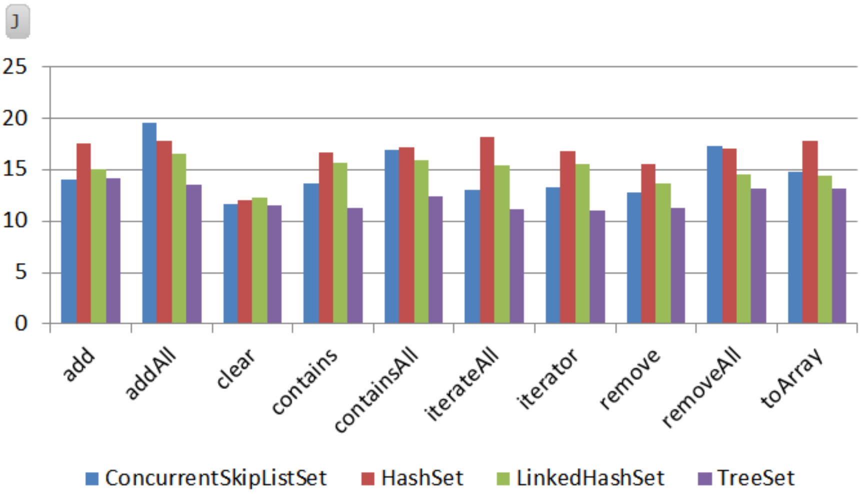

J
7
6
5
4
3
2
1
0
add
addAll
clear
contains
containsAll
iterateAll
iterator
remove
removeAll
toArray
ConcurrentSkipListSet
HashSet
LinkedHashSet
TreeSet

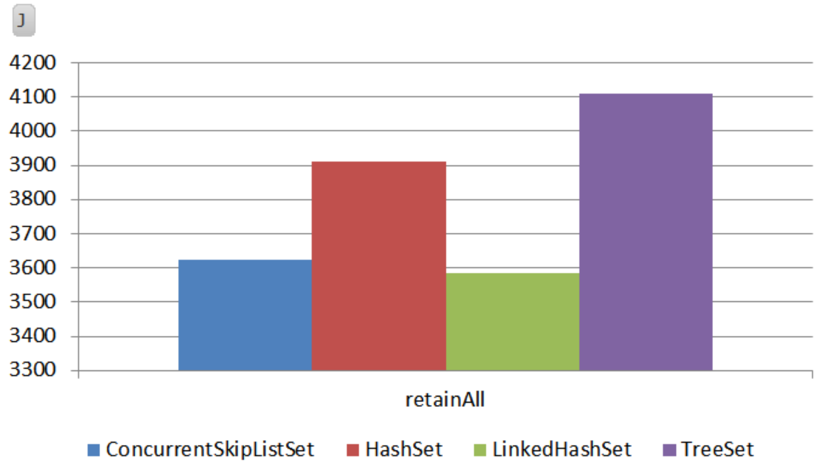

J
190
188
186
184
182
180
178
176
174
retainAll
ConcurrentSkipListSet
HashSet
LinkedHashSet
TreeSet

## B.3 Set data for 1m population



|  | Concurrent |  |  |  | Linked |  |  |  |
|---|---|---|---|---|---|---|---|---|
|  | SkipListSet |  | HashSet |  | HashSet |  | TreeSet |  |
| Methods | J | ms | J | ms | J | ms | J | ms |
| add | 14.0472 | 824 | 17.5243 | 1072 | 15.0643 | 876 | 14.1021 | 758 |
| addAll | 19.5092 | 1518 | 17.8589 | 1100 | 16.5155 | 983 | 13.5737 | 983 |
| clear | 11.5958 | 747 | 12.0199 | 764 | 12.2874 | 770 | 11.5565 | 758 |
| contains | 13.6576 | 870 | 16.6950 | 1014 | 15.6210 | 880 | 11.2337 | 682 |
| containsAll | 16.9809 | 1212 | 17.2110 | 1038 | 15.8865 | 886 | 12.3979 | 844 |
| iterateAll | 13.0184 | 785 | 18.1706 | 1091 | 15.4155 | 865 | 11.2088 | 684 |
| iterator | 13.2534 | 752 | 16.7433 | 1013 | 15.5284 | 850 | 11.0499 | 641 |
| remove | 12.7444 | 789 | 15.5699 | 949 | 13.6615 | 799 | 11.2653 | 675 |
| removeAll | 17.2849 | 1293 | 17.0514 | 998 | 14.5821 | 841 | 13.2071 | 937 |
| retainAll | 3621.9872 | 346898 | 3912.0129 | 384829 | 3584.3529 | 346337 | 4111.2397 | 408297 |
| toArray | 14.8120 | 875 | 17.8458 | 1070 | 14.3511 | 848 | 13.1271 | 750 |

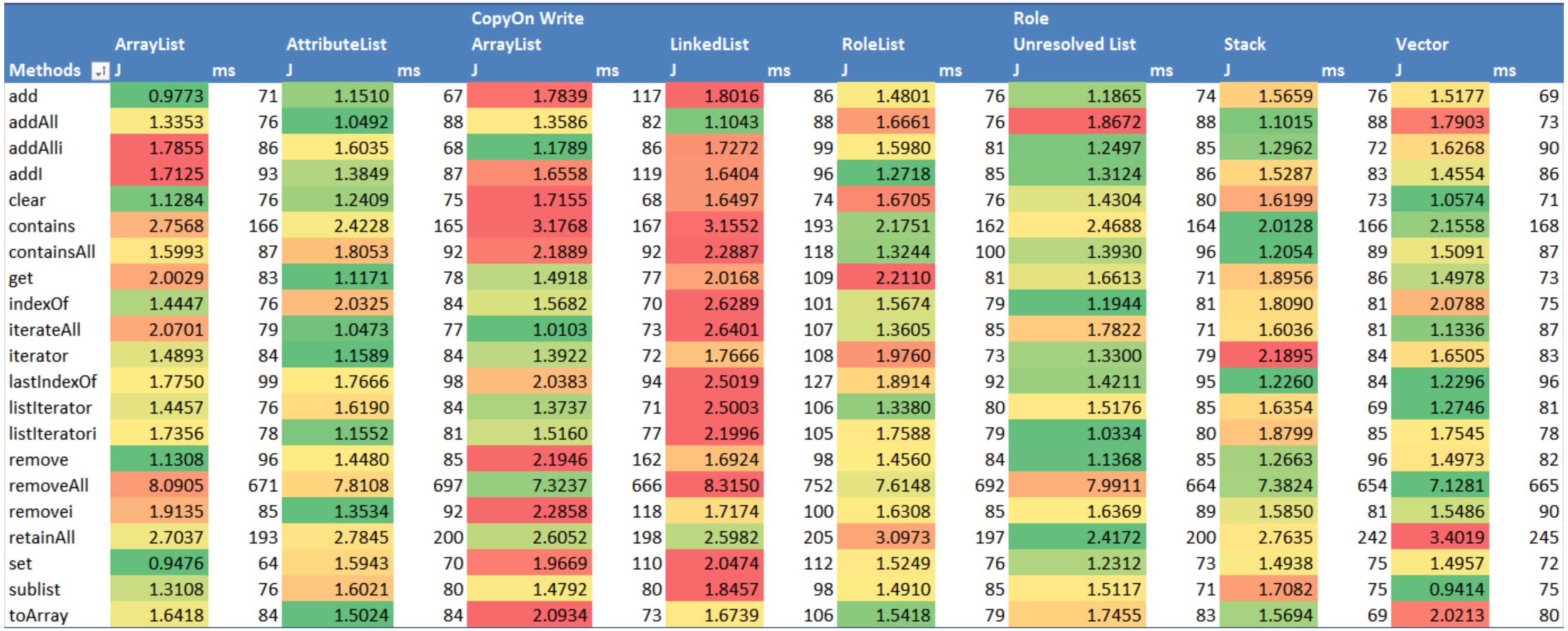




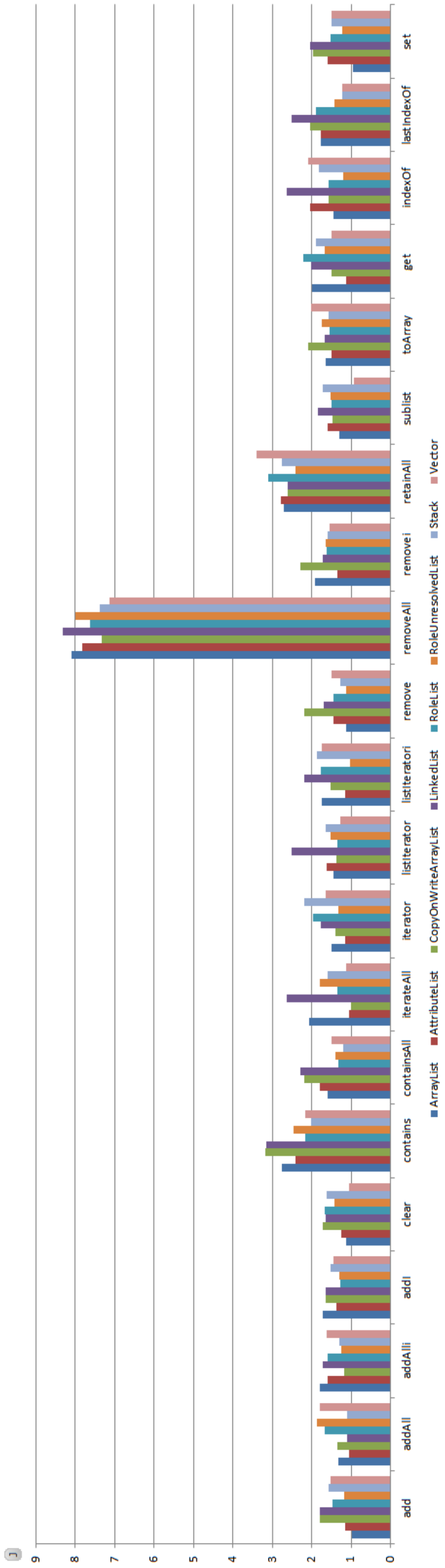
## B.4 List data for 25k population

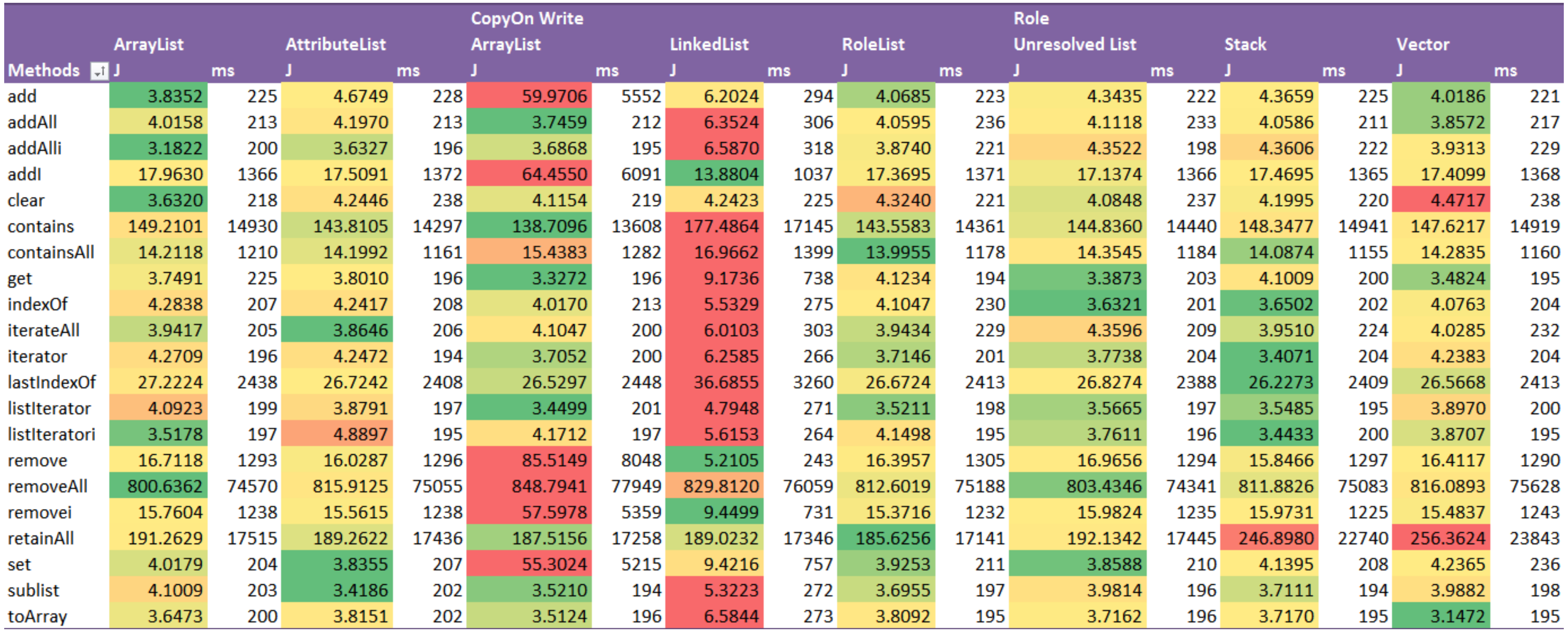
## B.5 List data for 250k population

## B.6 List data for 1m population

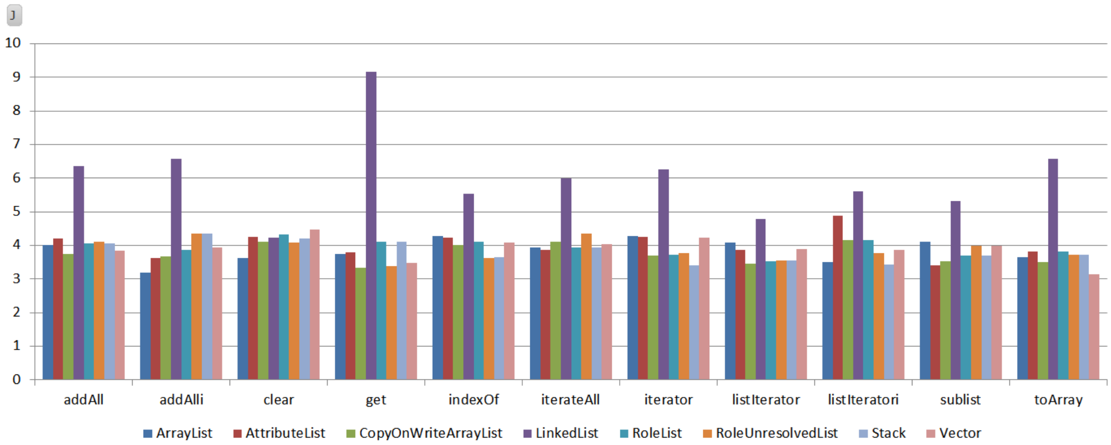


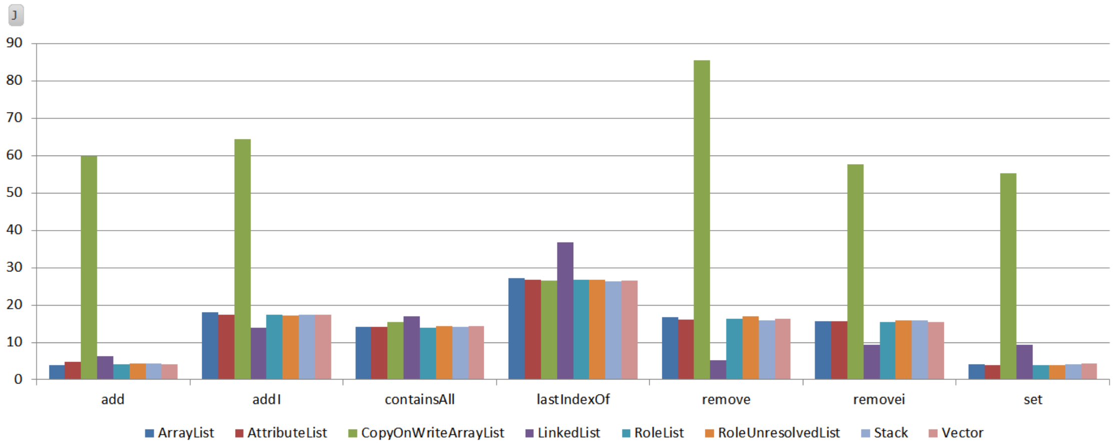
| | ArrayList | | AttributeList | | CopyOn Write ArrayList | | LinkedList | | RoleList | | Role Unresolved List | | Stack | | Vector | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Methods | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms |
| add | 0.9773 | 71 | 1.1510 | 67 | 1.7839 | 117 | 1.8016 | 86 | 1.4801 | 76 | 1.1865 | 74 | 1.5659 | 76 | 1.5177 | 69 |
| addAll | 1.3353 | 76 | 1.0492 | 88 | 1.3586 | 82 | 1.1043 | 88 | 1.6661 | 76 | 1.8672 | 88 | 1.1015 | 88 | 1.7903 | 73 |
| addAlli | 1.7855 | 86 | 1.6035 | 68 | 1.1789 | 86 | 1.7272 | 99 | 1.5980 | 81 | 1.2497 | 85 | 1.2962 | 72 | 1.6268 | 90 |
| addI | 1.7125 | 93 | 1.3849 | 87 | 1.6558 | 119 | 1.6404 | 96 | 1.2718 | 85 | 1.3124 | 86 | 1.5287 | 83 | 1.4554 | 86 |
| clear | 1.1284 | 76 | 1.2409 | 75 | 1.7155 | 68 | 1.6497 | 74 | 1.6705 | 76 | 1.4304 | 80 | 1.6199 | 73 | 1.0574 | 71 |
| contains | 2.7568 | 166 | 2.4228 | 165 | 3.1768 | 167 | 3.1552 | 193 | 2.1751 | 162 | 2.4688 | 164 | 2.0128 | 166 | 2.1558 | 168 |
| containsAll | 1.5993 | 87 | 1.8053 | 92 | 2.1889 | 92 | 2.2887 | 118 | 1.3244 | 100 | 1.3930 | 96 | 1.2054 | 89 | 1.5091 | 87 |
| get | 2.0029 | 83 | 1.1171 | 78 | 1.4918 | 77 | 2.0168 | 109 | 2.2110 | 81 | 1.6613 | 71 | 1.8956 | 86 | 1.4978 | 73 |
| indexOf | 1.4447 | 76 | 2.0325 | 84 | 1.5682 | 70 | 2.6289 | 101 | 1.5674 | 79 | 1.1944 | 81 | 1.8090 | 81 | 2.0788 | 75 |
| iterateAll | 2.0701 | 79 | 1.0473 | 77 | 1.0103 | 73 | 2.6401 | 107 | 1.3605 | 85 | 1.7822 | 71 | 1.6036 | 81 | 1.1336 | 87 |
| iterator | 1.4893 | 84 | 1.1589 | 84 | 1.3922 | 72 | 1.7666 | 108 | 1.9760 | 73 | 1.3300 | 79 | 2.1895 | 84 | 1.6505 | 83 |
| lastIndexOf | 1.7750 | 99 | 1.7666 | 98 | 2.0383 | 94 | 2.5019 | 127 | 1.8914 | 92 | 1.4211 | 95 | 1.2260 | 84 | 1.2296 | 96 |
| listIterator | 1.4457 | 76 | 1.6190 | 84 | 1.3737 | 71 | 2.5003 | 106 | 1.3380 | 80 | 1.5176 | 85 | 1.6354 | 69 | 1.2746 | 81 |
| listIteratori | 1.7356 | 78 | 1.1552 | 81 | 1.5160 | 77 | 2.1996 | 105 | 1.7588 | 79 | 1.0334 | 80 | 1.8799 | 85 | 1.7545 | 78 |
| remove | 1.1308 | 96 | 1.4480 | 85 | 2.1946 | 162 | 1.6924 | 98 | 1.4560 | 84 | 1.1368 | 85 | 1.2663 | 96 | 1.4973 | 82 |
| removeAll | 8.0905 | 671 | 7.8108 | 697 | 7.3237 | 666 | 8.3150 | 752 | 7.6148 | 692 | 7.9911 | 664 | 7.3824 | 654 | 7.1281 | 665 |
| removei | 1.9135 | 85 | 1.3534 | 92 | 2.2858 | 118 | 1.7174 | 100 | 1.6308 | 85 | 1.6369 | 89 | 1.5850 | 81 | 1.5486 | 90 |
| retainAll | 2.7037 | 193 | 2.7845 | 200 | 2.6052 | 198 | 2.5982 | 205 | 3.0973 | 197 | 2.4172 | 200 | 2.7635 | 242 | 3.4019 | 245 |
| set | 0.9476 | 64 | 1.5943 | 70 | 1.9669 | 110 | 2.0474 | 112 | 1.5249 | 76 | 1.2312 | 73 | 1.4938 | 75 | 1.4957 | 72 |
| sublist | 1.3108 | 76 | 1.6021 | 80 | 1.4792 | 80 | 1.8457 | 98 | 1.4910 | 85 | 1.5117 | 71 | 1.7082 | 75 | 0.9414 | 75 |
| toArray | 1.6418 | 84 | 1.5024 | 84 | 2.0934 | 73 | 1.6739 | 106 | 1.5418 | 79 | 1.7455 | 83 | 1.5694 | 69 | 2.0213 | 80 |

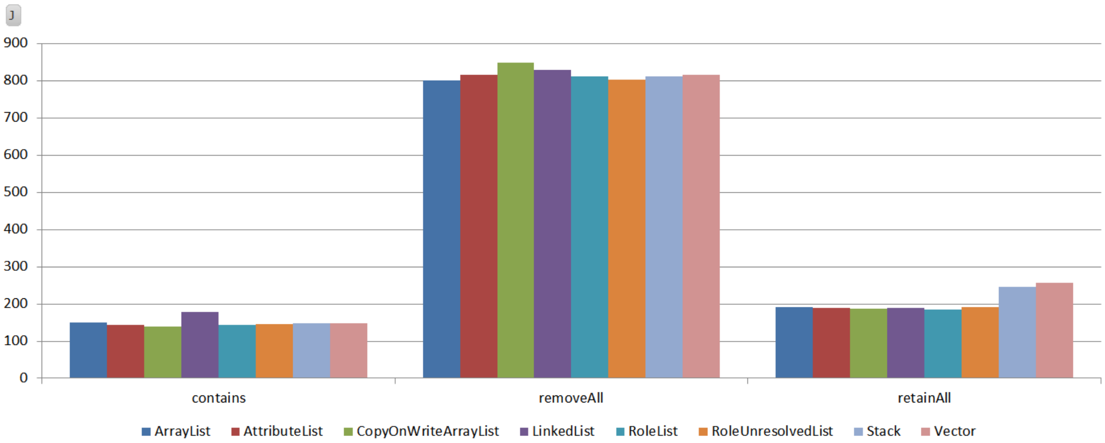

0
1
2
3
4
5
6
7
8
9
add
addAll
addAlli
addI
clear
contains
containsAll
iterateAll
iterator
listIterator
listIteratori
remove
removeAll
removei
retainAll
sublist
toArray
get
indexOf
lastIndexOf
set
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList
RoleList
RoleUnresolvedList
Stack
Vector

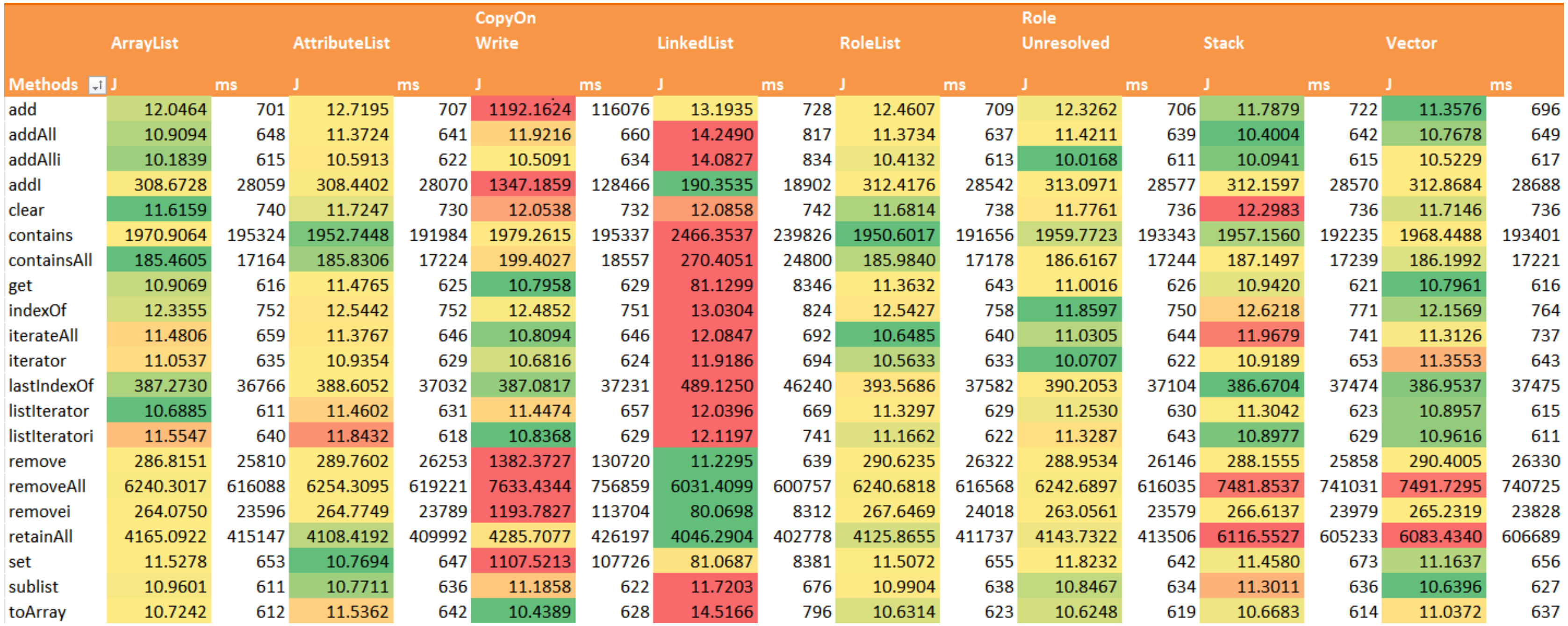


| | | | | | CopyOn Write | | | | | | Role | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ArrayList | | AttributeList | | ArrayList | | LinkedList | | RoleList | | Unresolved List | | Stack | | Vector | |
| Methods | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms |
| add | 3.8352 | 225 | 4.6749 | 228 | 59.9706 | 5552 | 6.2024 | 294 | 4.0685 | 223 | 4.3435 | 222 | 4.3659 | 225 | 4.0186 | 221 |
| addAll | 4.0158 | 213 | 4.1970 | 213 | 3.7459 | 212 | 6.3524 | 306 | 4.0595 | 236 | 4.1118 | 233 | 4.0586 | 211 | 3.8572 | 217 |
| addAlli | 3.1822 | 200 | 3.6327 | 196 | 3.6868 | 195 | 6.5870 | 318 | 3.8740 | 221 | 4.3522 | 198 | 4.3606 | 222 | 3.9313 | 229 |
| addI | 17.9630 | 1366 | 17.5091 | 1372 | 64.4550 | 6091 | 13.8804 | 1037 | 17.3695 | 1371 | 17.1374 | 1366 | 17.4695 | 1365 | 17.4099 | 1368 |
| clear | 3.6320 | 218 | 4.2446 | 238 | 4.1154 | 219 | 4.2423 | 225 | 4.3240 | 221 | 4.0848 | 237 | 4.1995 | 220 | 4.4717 | 238 |
| contains | 149.2101 | 14930 | 143.8105 | 14297 | 138.7096 | 13608 | 177.4864 | 17145 | 143.5583 | 14361 | 144.8360 | 14440 | 148.3477 | 14941 | 147.6217 | 14919 |
| containsAll | 14.2118 | 1210 | 14.1992 | 1161 | 15.4383 | 1282 | 16.9662 | 1399 | 13.9955 | 1178 | 14.3545 | 1184 | 14.0874 | 1155 | 14.2835 | 1160 |
| get | 3.7491 | 225 | 3.8010 | 196 | 3.3272 | 196 | 9.1736 | 738 | 4.1234 | 194 | 3.3873 | 203 | 4.1009 | 200 | 3.4824 | 195 |
| indexOf | 4.2838 | 207 | 4.2417 | 208 | 4.0170 | 213 | 5.5329 | 275 | 4.1047 | 230 | 3.6321 | 201 | 3.6502 | 202 | 4.0763 | 204 |
| iterateAll | 3.9417 | 205 | 3.8646 | 206 | 4.1047 | 200 | 6.0103 | 303 | 3.9434 | 229 | 4.3596 | 209 | 3.9510 | 224 | 4.0285 | 232 |
| iterator | 4.2709 | 196 | 4.2472 | 194 | 3.7052 | 200 | 6.2585 | 266 | 3.7146 | 201 | 3.7738 | 204 | 3.4071 | 204 | 4.2383 | 204 |
| lastIndexOf | 27.2224 | 2438 | 26.7242 | 2408 | 26.5297 | 2448 | 36.6855 | 3260 | 26.6724 | 2413 | 26.8274 | 2388 | 26.2273 | 2409 | 26.5668 | 2413 |
| listIterator | 4.0923 | 199 | 3.8791 | 197 | 3.4499 | 201 | 4.7948 | 271 | 3.5211 | 198 | 3.5665 | 197 | 3.5485 | 195 | 3.8970 | 200 |
| listIteratori | 3.5178 | 197 | 4.8897 | 195 | 4.1712 | 197 | 5.6153 | 264 | 4.1498 | 195 | 3.7611 | 196 | 3.4433 | 200 | 3.8707 | 195 |
| remove | 16.7118 | 1293 | 16.0287 | 1296 | 85.5149 | 8048 | 5.2105 | 243 | 16.3957 | 1305 | 16.9656 | 1294 | 15.8466 | 1297 | 16.4117 | 1290 |
| removeAll | 800.6362 | 74570 | 815.9125 | 75055 | 848.7941 | 77949 | 829.8120 | 76059 | 812.6019 | 75188 | 803.4346 | 74341 | 811.8826 | 75083 | 816.0893 | 75628 |
| removei | 15.7604 | 1238 | 15.5615 | 1238 | 57.5978 | 5359 | 9.4499 | 731 | 15.3716 | 1232 | 15.9824 | 1235 | 15.9731 | 1225 | 15.4837 | 1243 |
| retainAll | 191.2629 | 17515 | 189.2622 | 17436 | 187.5156 | 17258 | 189.0232 | 17346 | 185.6256 | 17141 | 192.1342 | 17445 | 246.8980 | 22740 | 256.3624 | 23843 |
| set | 4.0179 | 204 | 3.8355 | 207 | 55.3024 | 5215 | 9.4216 | 757 | 3.9253 | 211 | 3.8588 | 210 | 4.1395 | 208 | 4.2365 | 236 |
| sublist | 4.1009 | 203 | 3.4186 | 202 | 3.5210 | 194 | 5.3223 | 272 | 3.6955 | 197 | 3.9814 | 196 | 3.7111 | 194 | 3.9882 | 198 |
| toArray | 3.6473 | 200 | 3.8151 | 202 | 3.5124 | 196 | 6.5844 | 273 | 3.8092 | 195 | 3.7162 | 196 | 3.7170 | 195 | 3.1472 | 195 |

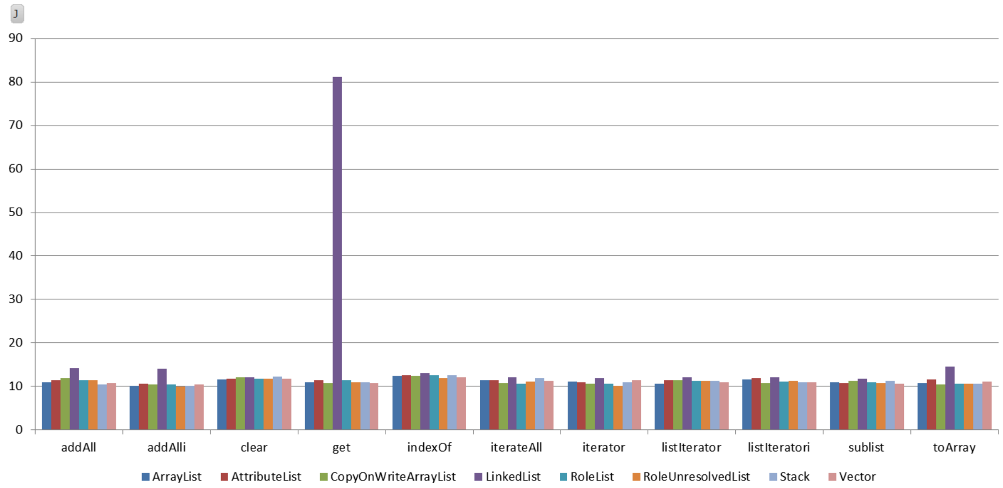

J
10
9
8
7
6
5
4
3
2
1
0
addAll
addAlli
clear
get
indexOf
iterateAll
iterator
listIterator
listIteratori
sublist
toArray
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList
RoleList
RoleUnresolvedList
Stack
Vector

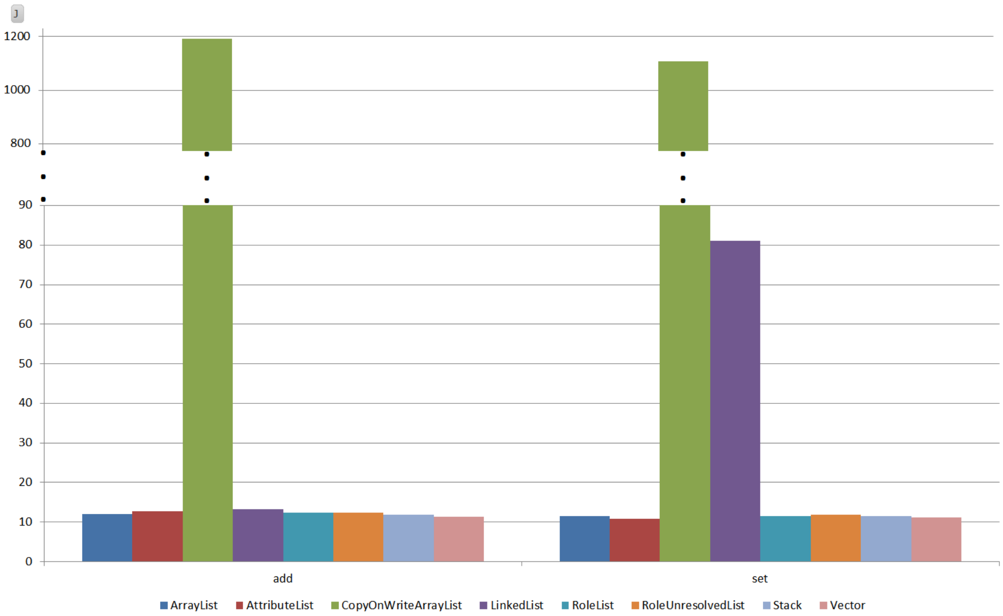

J
90
80
70
60
50
40
30
20
10
0
add
addI
containsAll
lastIndexOf
remove
removei
set
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList
RoleList
RoleUnresolvedList
Stack
Vector

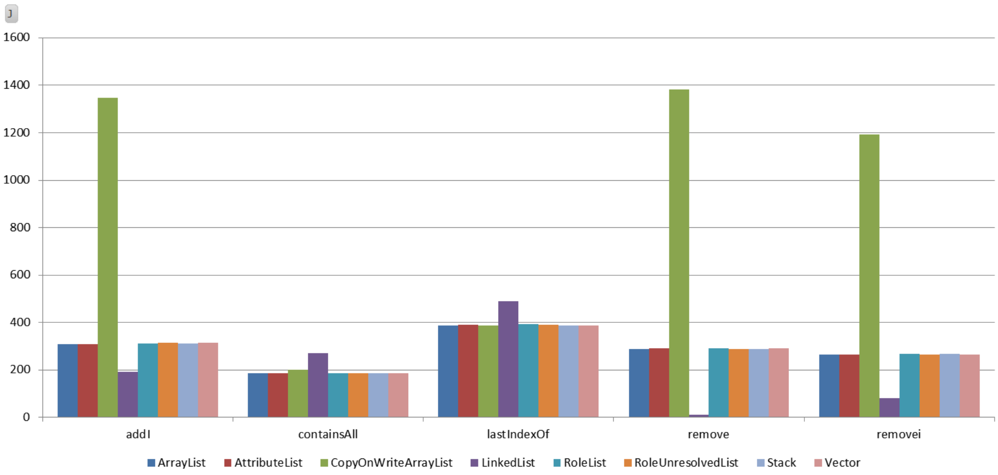

J
900
800
700
600
500
400
300
200
100
0
contains
removeAll
retainAll
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList
RoleList
RoleUnresolvedList
Stack
Vector

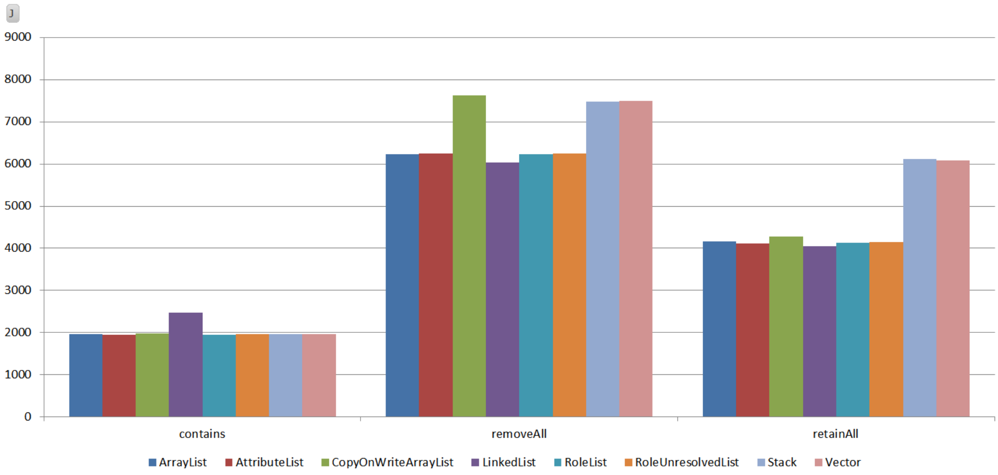


| | ArrayList | | AttributeList | | CopyOn Write | | LinkedList | | RoleList | | Role Unresolved | | Stack | | Vector | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Methods | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms |
| add | 12.0464 | 701 | 12.7195 | 707 | 1192.1624 | 116076 | 13.1935 | 728 | 12.4607 | 709 | 12.3262 | 706 | 11.7879 | 722 | 11.3576 | 696 |
| addAll | 10.9094 | 648 | 11.3724 | 641 | 11.9216 | 660 | 14.2490 | 817 | 11.3734 | 637 | 11.4211 | 639 | 10.4004 | 642 | 10.7678 | 649 |
| addAlli | 10.1839 | 615 | 10.5913 | 622 | 10.5091 | 634 | 14.0827 | 834 | 10.4132 | 613 | 10.0168 | 611 | 10.0941 | 615 | 10.5229 | 617 |
| addI | 308.6728 | 28059 | 308.4402 | 28070 | 1347.1859 | 128466 | 190.3535 | 18902 | 312.4176 | 28542 | 313.0971 | 28577 | 312.1597 | 28570 | 312.8684 | 28688 |
| clear | 11.6159 | 740 | 11.7247 | 730 | 12.0538 | 732 | 12.0858 | 742 | 11.6814 | 738 | 11.7761 | 736 | 12.2983 | 736 | 11.7146 | 736 |
| contains | 1970.9064 | 195324 | 1952.7448 | 191984 | 1979.2615 | 195337 | 2466.3537 | 239826 | 1950.6017 | 191656 | 1959.7723 | 193343 | 1957.1560 | 192235 | 1968.4488 | 193401 |
| containsAll | 185.4605 | 17164 | 185.8306 | 17224 | 199.4027 | 18557 | 270.4051 | 24800 | 185.9840 | 17178 | 186.6167 | 17244 | 187.1497 | 17239 | 186.1992 | 17221 |
| get | 10.9069 | 616 | 11.4765 | 625 | 10.7958 | 629 | 81.1299 | 8346 | 11.3632 | 643 | 11.0016 | 626 | 10.9420 | 621 | 10.7961 | 616 |
| indexOf | 12.3355 | 752 | 12.5442 | 752 | 12.4852 | 751 | 13.0304 | 824 | 12.5427 | 758 | 11.8597 | 750 | 12.6218 | 771 | 12.1569 | 764 |
| iterateAll | 11.4806 | 659 | 11.3767 | 646 | 10.8094 | 646 | 12.0847 | 692 | 10.6485 | 640 | 11.0305 | 644 | 11.9679 | 741 | 11.3126 | 737 |
| iterator | 11.0537 | 635 | 10.9354 | 629 | 10.6816 | 624 | 11.9186 | 694 | 10.5633 | 633 | 10.0707 | 622 | 10.9189 | 653 | 11.3553 | 643 |
| lastIndexOf | 387.2730 | 36766 | 388.6052 | 37032 | 387.0817 | 37231 | 489.1250 | 46240 | 393.5686 | 37582 | 390.2053 | 37104 | 386.6704 | 37474 | 386.9537 | 37475 |
| listIterator | 10.6885 | 611 | 11.4602 | 631 | 11.4474 | 657 | 12.0396 | 669 | 11.3297 | 629 | 11.2530 | 630 | 11.3042 | 623 | 10.8957 | 615 |
| listIteratori | 11.5547 | 640 | 11.8432 | 618 | 10.8368 | 629 | 12.1197 | 741 | 11.1662 | 622 | 11.3287 | 643 | 10.8977 | 629 | 10.9616 | 611 |
| remove | 286.8151 | 25810 | 289.7602 | 26253 | 1382.3727 | 130720 | 11.2295 | 639 | 290.6235 | 26322 | 288.9534 | 26146 | 288.1555 | 25858 | 290.4005 | 26330 |
| removeAll | 6240.3017 | 616088 | 6254.3095 | 619221 | 7633.4344 | 756859 | 6031.4099 | 600757 | 6240.6818 | 616568 | 6242.6897 | 616035 | 7481.8537 | 741031 | 7491.7295 | 740725 |
| removei | 264.0750 | 23596 | 264.7749 | 23789 | 1193.7827 | 113704 | 80.0698 | 8312 | 267.6469 | 24018 | 263.0561 | 23579 | 266.6137 | 23979 | 265.2319 | 23828 |
| retainAll | 4165.0922 | 415147 | 4108.4192 | 409992 | 4285.7077 | 426197 | 4046.2904 | 402778 | 4125.8655 | 411737 | 4143.7322 | 413506 | 6116.5527 | 605233 | 6083.4340 | 606689 |
| set | 11.5278 | 653 | 10.7694 | 647 | 1107.5213 | 107726 | 81.0687 | 8381 | 11.5072 | 655 | 11.8232 | 642 | 11.4580 | 673 | 11.1637 | 656 |
| sublist | 10.9601 | 611 | 10.7711 | 636 | 11.1858 | 622 | 11.7203 | 676 | 10.9904 | 638 | 10.8467 | 634 | 11.3011 | 636 | 10.6396 | 627 |
| toArray | 10.7242 | 612 | 11.5362 | 642 | 10.4389 | 628 | 14.5166 | 796 | 10.6314 | 623 | 10.6248 | 619 | 10.6683 | 614 | 11.0372 | 637 |


J
90
80
70
60
50
40
30
20
10
0
addAll
addAlli
clear
get
indexOf
iterateAll
iterator
listIterator
listIteratori
sublist
toArray
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList
RoleList
RoleUnresolvedList
Stack
Vector


J
1200
1000
800
90
80
70
60
50
40
30
20
10
0
add
set
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList
RoleList
RoleUnresolvedList
Stack
Vector


1600
1400
1200
1000
800
600
400
200
0
addI
containsAll
lastIndexOf
remove
removei
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList
RoleList
RoleUnresolvedList
Stack
Vector

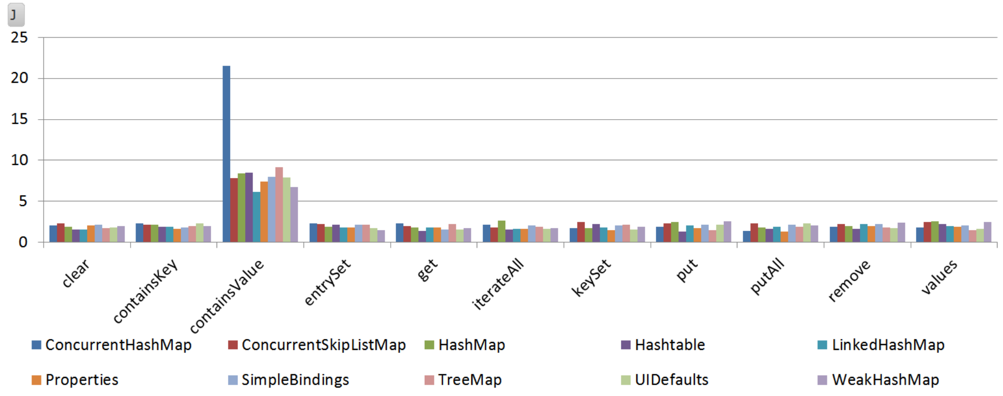

9000
8000
7000
6000
5000
4000
3000
2000
1000
0
contains
removeAll
retainAll
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList
RoleList
RoleUnresolvedList
Stack
Vector

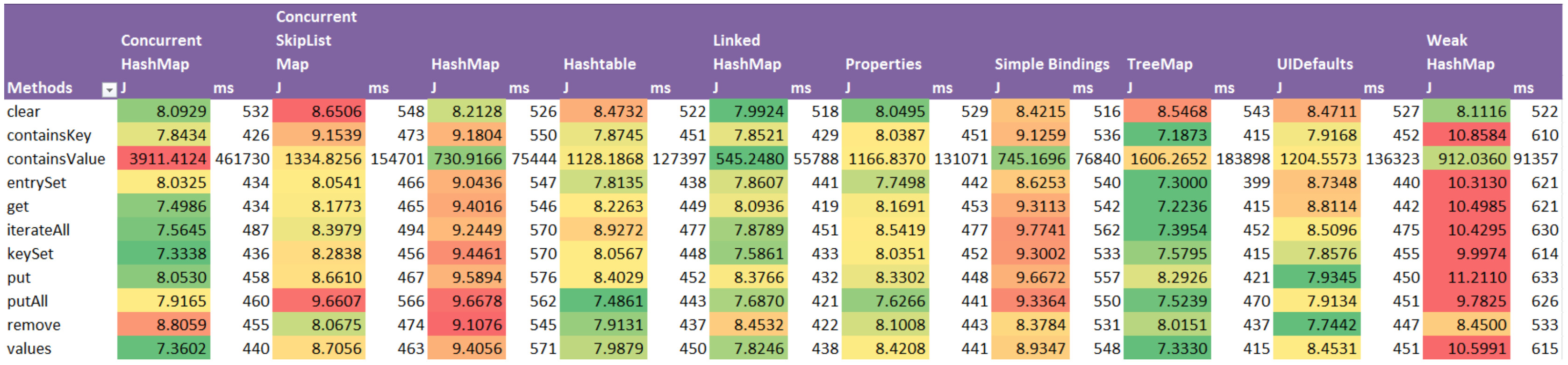
## B.7 Map data for 25k population

## B.8 Map data for 250k population

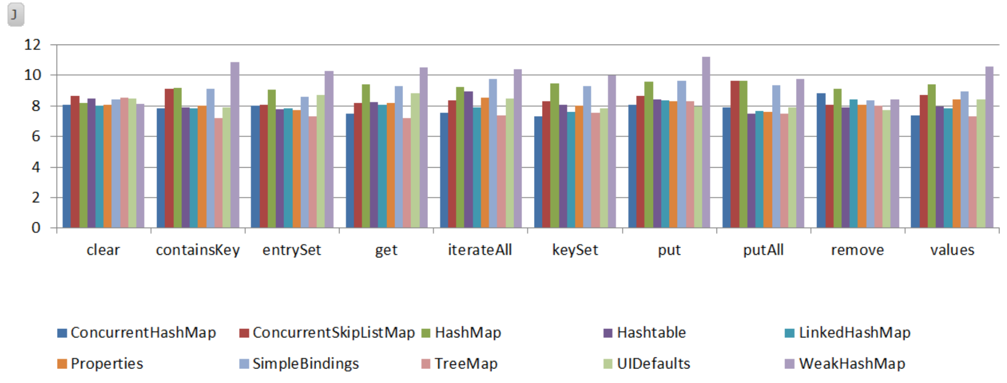
## B.9 Map data for 1m population

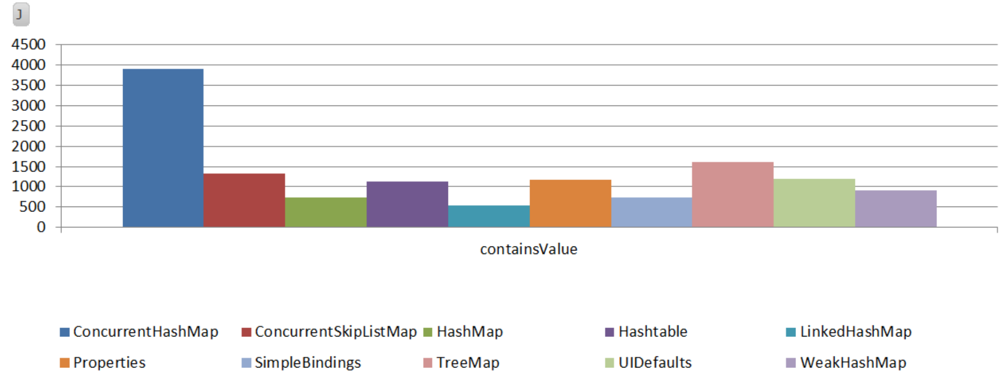
| | Concurrent HashMap | | Concurrent SkipListMap | | HashMap | | Hashtable | | Linked HashMap | | Properties | | Simple Bindings | | TreeMap | | UIDefaults | | Weak HashMap | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Methods | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms |
| clear | 2.0276 | 94 | 2.2961 | 88 | 1.8395 | 104 | 1.5761 | 94 | 1.5025 | 97 | 2.0777 | 98 | 2.1401 | 106 | 1.6706 | 98 | 1.8143 | 105 | 1.9941 | 95 |
| containsKey | 2.3132 | 105 | 2.1693 | 123 | 2.1343 | 103 | 1.8582 | 94 | 1.8726 | 103 | 1.6018 | 107 | 1.8055 | 99 | 1.9452 | 100 | 2.3366 | 89 | 1.9675 | 108 |
| containsValue | 21.5611 | 2305 | 7.8032 | 643 | 8.3615 | 683 | 8.4957 | 765 | 6.1326 | 462 | 7.3755 | 692 | 7.9912 | 678 | 9.1771 | 847 | 7.9341 | 714 | 6.7072 | 562 |
| entrySet | 2.2878 | 93 | 2.2363 | 116 | 1.8531 | 108 | 2.1332 | 107 | 1.8362 | 113 | 1.7800 | 97 | 2.1557 | 102 | 2.1617 | 115 | 1.7087 | 105 | 1.4666 | 102 |
| get | 2.3106 | 103 | 1.9972 | 119 | 1.8120 | 102 | 1.4071 | 100 | 1.8252 | 116 | 1.7851 | 97 | 1.5359 | 100 | 2.2331 | 115 | 1.5252 | 89 | 1.7185 | 103 |
| iterateAll | 2.1041 | 96 | 1.8353 | 115 | 2.6673 | 100 | 1.5343 | 91 | 1.6462 | 111 | 1.6362 | 100 | 2.0472 | 116 | 1.9122 | 111 | 1.6574 | 95 | 1.7139 | 106 |
| keySet | 1.7287 | 95 | 2.4889 | 124 | 1.6813 | 114 | 2.2226 | 99 | 1.8328 | 103 | 1.4866 | 92 | 2.0630 | 106 | 2.1680 | 110 | 1.5547 | 99 | 1.8749 | 105 |
| put | 1.8591 | 104 | 2.2888 | 102 | 2.4628 | 92 | 1.3123 | 96 | 2.0338 | 108 | 1.7038 | 107 | 2.1646 | 102 | 1.4355 | 91 | 2.1204 | 93 | 2.5784 | 105 |
| putAll | 1.4147 | 95 | 2.2852 | 122 | 1.7564 | 100 | 1.5949 | 105 | 1.8608 | 113 | 1.3097 | 95 | 2.1461 | 112 | 1.8914 | 116 | 2.3094 | 87 | 2.0750 | 108 |
| remove | 1.8574 | 92 | 2.2131 | 105 | 1.9256 | 109 | 1.6067 | 97 | 2.2300 | 106 | 1.9660 | 98 | 2.2178 | 106 | 1.8133 | 101 | 1.6888 | 92 | 2.4103 | 103 |
| values | 1.8279 | 85 | 2.4690 | 116 | 2.5755 | 109 | 2.2266 | 94 | 2.0009 | 107 | 1.9120 | 111 | 2.0692 | 108 | 1.4467 | 105 | 1.6533 | 100 | 2.4628 | 111 |

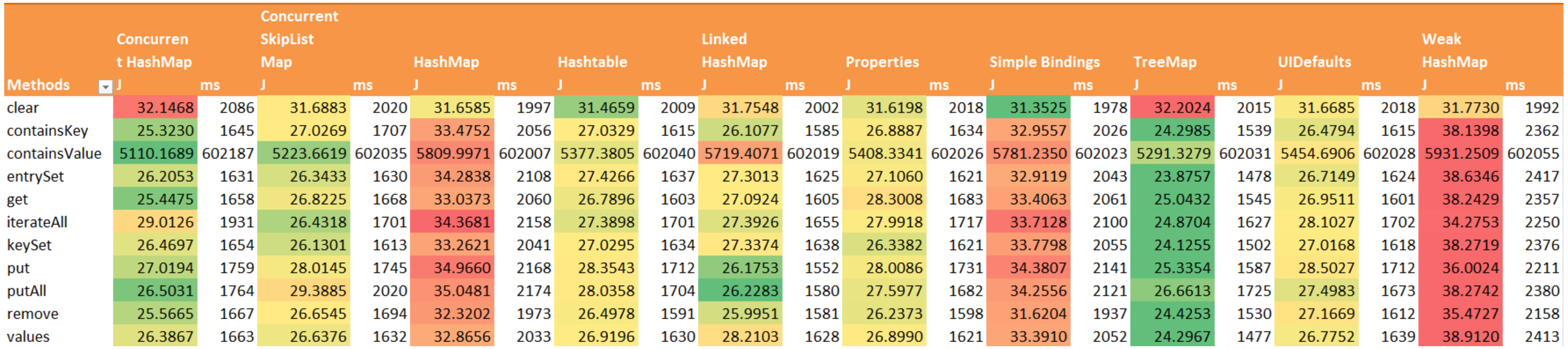




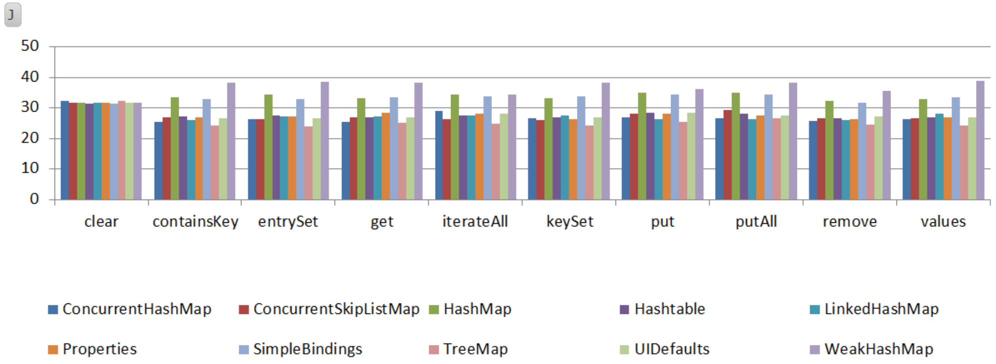
| Methods | Concurrent HashMap |  | Concurrent SkipList Map |  | HashMap |  | Hashtable |  | Linked HashMap |  | Properties |  | Simple Bindings |  | TreeMap |  | UIDefaults |  | Weak HashMap |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms | J | ms |
| clear | 8.0929 | 532 | 8.6506 | 548 | 8.2128 | 526 | 8.4732 | 522 | 7.9924 | 518 | 8.0495 | 529 | 8.4215 | 516 | 8.5468 | 543 | 8.4711 | 527 | 8.1116 | 522 |
| containsKey | 7.8434 | 426 | 9.1539 | 473 | 9.1804 | 550 | 7.8745 | 451 | 7.8521 | 429 | 8.0387 | 451 | 9.1259 | 536 | 7.1873 | 415 | 7.9168 | 452 | 10.8584 | 610 |
| containsValue | 3911.4124 | 461730 | 1334.8256 | 154701 | 730.9166 | 75444 | 1128.1868 | 127397 | 545.2480 | 55788 | 1166.8370 | 131071 | 745.1696 | 76840 | 1606.2652 | 183898 | 1204.5573 | 136323 | 912.0360 | 91357 |
| entrySet | 8.0325 | 434 | 8.0541 | 466 | 9.0436 | 547 | 7.8135 | 438 | 7.8607 | 441 | 7.7498 | 442 | 8.6253 | 540 | 7.3000 | 399 | 8.7348 | 440 | 10.3130 | 621 |
| get | 7.4986 | 434 | 8.1773 | 465 | 9.4016 | 546 | 8.2263 | 449 | 8.0936 | 419 | 8.1691 | 453 | 9.3113 | 542 | 7.2236 | 415 | 8.8114 | 442 | 10.4985 | 621 |
| iterateAll | 7.5645 | 487 | 8.3979 | 494 | 9.2449 | 570 | 8.9272 | 477 | 7.8789 | 451 | 8.5419 | 477 | 9.7741 | 562 | 7.3954 | 452 | 8.5096 | 475 | 10.4295 | 630 |
| keySet | 7.3338 | 436 | 8.2838 | 456 | 9.4461 | 570 | 8.0567 | 448 | 7.5861 | 433 | 8.0351 | 452 | 9.3002 | 533 | 7.5795 | 415 | 7.8576 | 455 | 9.9974 | 614 |
| put | 8.0530 | 458 | 8.6610 | 467 | 9.5894 | 576 | 8.4029 | 452 | 8.3766 | 432 | 8.3302 | 448 | 9.6672 | 557 | 8.2926 | 421 | 7.9345 | 450 | 11.2110 | 633 |
| putAll | 7.9165 | 460 | 9.6607 | 566 | 9.6678 | 562 | 7.4861 | 443 | 7.6870 | 421 | 7.6266 | 441 | 9.3364 | 550 | 7.5239 | 470 | 7.9134 | 451 | 9.7825 | 626 |
| remove | 8.8059 | 455 | 8.0675 | 474 | 9.1076 | 545 | 7.9131 | 437 | 8.4532 | 422 | 8.1008 | 443 | 8.3784 | 531 | 8.0151 | 437 | 7.7442 | 447 | 8.4500 | 533 |
| values | 7.3602 | 440 | 8.7056 | 463 | 9.4056 | 571 | 7.9879 | 450 | 7.8246 | 438 | 8.4208 | 441 | 8.9347 | 548 | 7.3330 | 415 | 8.4531 | 451 | 10.5991 | 615 |

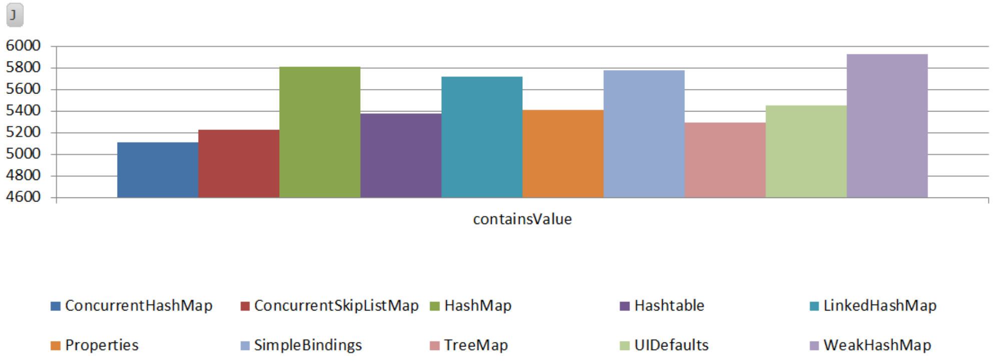






| Methods | Concurrent HashMap J | ms | Concurrent SkipList Map J | ms | HashMap J | ms | Hashtable J | ms | Linked HashMap J | ms | Properties J | ms | Simple Bindings J | ms | TreeMap J | ms | UIDefaults J | ms | Weak HashMap J | ms |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| clear | 32.1468 | 2086 | 31.6883 | 2020 | 31.6585 | 1997 | 31.4659 | 2009 | 31.7548 | 2002 | 31.6198 | 2018 | 31.3525 | 1978 | 32.2024 | 2015 | 31.6685 | 2018 | 31.7730 | 1992 |
| containsKey | 25.3230 | 1645 | 27.0269 | 1707 | 33.4752 | 2056 | 27.0329 | 1615 | 26.1077 | 1585 | 26.8887 | 1634 | 32.9557 | 2026 | 24.2985 | 1539 | 26.4794 | 1615 | 38.1398 | 2362 |
| containsValue | 5110.1689 | 602187 | 5223.6619 | 602035 | 5809.9971 | 602007 | 5377.3805 | 602040 | 5719.4071 | 602019 | 5408.3341 | 602026 | 5781.2350 | 602023 | 5291.3279 | 602031 | 5454.6906 | 602028 | 5931.2509 | 602055 |
| entrySet | 26.2053 | 1631 | 26.3433 | 1630 | 34.2838 | 2108 | 27.4266 | 1637 | 27.3013 | 1625 | 27.1060 | 1621 | 32.9119 | 2043 | 23.8757 | 1478 | 26.7149 | 1624 | 38.6346 | 2417 |
| get | 25.4475 | 1658 | 26.8225 | 1668 | 33.0373 | 2060 | 26.7896 | 1603 | 27.0924 | 1605 | 28.3008 | 1683 | 33.4063 | 2061 | 25.0432 | 1545 | 26.9511 | 1601 | 38.2429 | 2357 |
| iterateAll | 29.0126 | 1931 | 26.4318 | 1701 | 34.3681 | 2158 | 27.3898 | 1701 | 27.3926 | 1655 | 27.9918 | 1717 | 33.7128 | 2100 | 24.8704 | 1627 | 28.1027 | 1702 | 34.2753 | 2250 |
| keySet | 26.4697 | 1654 | 26.1301 | 1613 | 33.2621 | 2041 | 27.0295 | 1634 | 27.3374 | 1638 | 26.3382 | 1621 | 33.7798 | 2055 | 24.1255 | 1502 | 27.0168 | 1618 | 38.2719 | 2376 |
| put | 27.0194 | 1759 | 28.0145 | 1745 | 34.9660 | 2168 | 28.3543 | 1712 | 26.1753 | 1552 | 28.0086 | 1731 | 34.3807 | 2141 | 25.3354 | 1587 | 28.5027 | 1712 | 36.0024 | 2211 |
| putAll | 26.5031 | 1764 | 29.3885 | 2020 | 35.0481 | 2174 | 28.0358 | 1704 | 26.2283 | 1580 | 27.5977 | 1682 | 34.2556 | 2121 | 26.6613 | 1725 | 27.4983 | 1673 | 38.2742 | 2380 |
| remove | 25.5665 | 1667 | 26.6545 | 1694 | 32.3202 | 1973 | 26.4978 | 1591 | 25.9951 | 1581 | 26.2373 | 1598 | 31.6204 | 1937 | 24.4253 | 1530 | 27.1669 | 1612 | 35.4727 | 2158 |
| values | 26.3867 | 1663 | 26.6376 | 1632 | 32.8656 | 2033 | 26.9196 | 1630 | 28.2103 | 1628 | 26.8990 | 1621 | 33.3910 | 2052 | 24.2967 | 1477 | 26.7752 | 1639 | 38.9120 | 2413 |





# References

Abdulsalam, S., Zong, Z., Gu, Q., and Qiu, M. (2015). Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency. In *Proceedings of the 6th International Green and Sustainable Computing Conference*, pages 1–8. IEEE.

Abreu, R., Zoeteweij, P., and Gemund, A. J. C. v. (2009). Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99. IEEE Computer Society.

Abreu, R., Zoeteweij, P., and Van Gemund, A. J. (2007). On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE.

Atlassian (2018). Openclover 4.2.1. http://openclover.org.

Banerjee, A., Chong, L. K., Chattopadhyay, S., and Roychoudhury, A. (2014). Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 588–598. ACM.

Banerjee, A. and Roychoudhury, A. (2016). Automated re-factoring of android apps to enhance energy-efficiency. In *International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM*, pages 139–150. IEEE.

Bartenstein, T. W. and Liu, Y. D. (2014). Rate types for stream programs. In *ACM SIGPLAN Notices*, volume 49, pages 213–232. ACM.

Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Mahaux, M., Penzenstadler, B., Rodríguez-Navas, G., Salinesi, C., Seyff, N., Venters, C. C., Calero, C., Koçak, S. A., and Betz, S. (2014). The karlskrona manifesto for sustainability design. *CoRR*, abs/1410.6968.

Bourque, P., Dupuis, R., Abran, A., Moore, J. W., and Tripp, L. (1999). The guide to the software engineering body of knowledge. *IEEE software*, 16(6):35–44.

Brandolese, C., Fornaciari, W., Salice, F., and Sciuto, D. (2002). The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems, and Computers*, 11(05):477–502.

Brown, R. et al. (2008). Report to congress on server and data center energy efficiency: Public law 109-431. *Lawrence Berkeley National Laboratory*.

Bunse, C., Höpfner, H., Mansour, E., and Roychoudhury, S. (2009a). Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. In *Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on*, pages 600–607. IEEE.

Bunse, C., Höpfner, H., Roychoudhury, S., and Mansour, E. (2009b). Choosing the "best" sorting algorithm for optimal energy consumption. In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 199–206.

Bunse, C. and Stiemer, S. (2013). On the energy consumption of design patterns. *Softwaretechnik-Trends*, 33(2).

Chandrakasan, A. P., Sheng, S., and Brodersen, R. W. (1992). Low-power cmos digital design. *Institute of Electronics, Information and Communication Engineers Transactions on Electronics*, 75(4):371–382.

Chowdhury, S. A. and Hindle, A. (2016). Greenoracle: estimating software energy consumption with energy measurement corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR, 2016*, pages 49–60.

Cohen, J. (1988). Statistical power analysis for the behavioral sciences. 1988. *Hillsdale, NJ: Lawrence Earlbaum Associates*, 2.

Cook, T. D. and Campbell, D. T. (1979). *Quasi-experimentation: design & analysis issues for field settings*. Houghton Mifflin.

Couto, M., Borba, P., Cunha, J., Fernandes, J. P., Pereira, R., and Saraiva, J. (2017a). Products go green: Worst-case energy consumption in software product lines. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, pages 84–93. ACM.

Couto, M., Carção, T., Cunha, J., Fernandes, J. P., and Saraiva, J. (2014). Detecting anomalous energy consumption in android applications. In *Proceedings of the 18th Brazilian Symposium on Programming Languages*, SBLP 2014, pages 77–91. Springer International Publishing.

Couto, M., Cunha, J., Fernandes, J. P., Pereira, R., and Saraiva, J. (2015). Greendroid: A tool for analysing power consumption in the android ecosystem. In *Scientific Conference on Informatics, 2015 IEEE 13th International*, pages 73–78. IEEE.

Couto, M., Cunha, J., Fernandes, J. P., Pereira, R., and Saraiva, J. A. (2016). Static energy consumption analysis in variability systems. In *2nd Green in Software Engineering Workshop (GInSEng'16), an event of the 4th International Conference on ICT for Sustainability (ICT4S)*, Amsterdam, The Netherlands.

Couto, M., Pereira, R., Ribeiro, F., Rua, R., and Saraiva, J. a. (2017b). Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, SBLP 2017, pages 7:1–7:8, New York, NY, USA. ACM. Best Paper.

Cruz, L. and Abreu, R. (2017). Performance-based guidelines for energy efficient mobile applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, pages 46–57, Piscataway, NJ, USA. IEEE Press.

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

Deb, K., Mohan, M., and Mishra, S. (2005). Evaluating the $\epsilon$-domination based multiobjective evolutionary algorithm for a quick computation of pareto-optimal solutions. *Evolutionary Computation Journal*, 13(4):501–525.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Transactions Evolutiontionary Computation*, 6(2):182–197.

Delaluz, V., Kandemir, M., Vijaykrishnan, N., Sivasubramaniam, A., and Irwin, M. J. (2001). Dram energy management using software and hardware directed power mode control. In *The Seventh International Symposium on High-Performance Computer Architecture, 2001. HPCA.*, pages 159–169. IEEE.

Dimitrov, M., Strickland, C., Kim, S.-W., Kumar, K., and Doshi, K. (2015). Intel® power governor. `https://software.intel.com/en-us/articles/intel-power-governor`. Accessed: 2017-10-12.

Dombek, P. E., Johnson, L. K., Zimmerley, S. T., and Sadowsky, M. J. (2000). Use of repetitive dna sequences and the pcr to differentiate escherichia coli isolates from human and animal sources. *Applied and Environmental Microbiology*, 66(6):2572–2577.

Douglis, F., Krishnan, P., Bershad, B., et al. (1995). Adaptive disk spin-down policies for mobile computers. *Computing Systems*, 8(4):381–413.

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

Field, A. (2009). *Discovering statistics using SPSS*. Sage publications.

Flautner, K., Reinhardt, S., and Mudge, T. (2001). Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 260–271. ACM.

Fraser, G. and Arcuri, A. (2014). A large scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8:1–8:42.

Gelenbe, E. and Caseau, Y. (2015). The impact of information technology on energy consumption and carbon emissions. *Ubiquity*, 2015(June):1:1–1:15.

Georgiou, S., Kechagia, M., Louridas, P., and Spinellis, D. (2018). What are your programming language's energy-delay implications? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 303–313, New York, NY, USA. ACM.

Gouy, I. (2018). The computer language benchmarks game. `http://benchmarksgame.alioth.debian.org/`.

Grech, N., Georgiou, K., Pallister, J., Kerrison, S., Morse, J., and Eder, K. (2015). Static analysis of energy consumption for llvm ir programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 12–21. ACM.

Guelzim, T. and Obaidat, M. S. (2013). Chapter 8 - Green Computing and Communication Architecture. In *Handbook of Green Information and Communication Systems*, pages 209–227. Academic Press.

Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Performance Evaluation Review*, 40(3):13–17.

Hao, S., Li, D., Halfond, W. G. J., and Govindan, R. (2013). Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101. IEEE Press.

Harmon, R. R. and Auseklis, N. (2009). Sustainable it services: Assessing the impact of green computing practices. In *Management of Engineering & Technology, 2009. PICMET 2009. Portland International Conference on*, pages 1707–1717. IEEE.

Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., and Hindle, A. (2016). Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 225–236. ACM.

Hindle, A. (2015). Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, 20(2):374–409.

Hogg, R. V. and Tanis, E. A. (1977). *Probability and statistical inference*, volume 993. Macmillan New York.

Hooper, A. (2008). Green computing. *Communication of the ACM*, 51(10):11–13.

Hoque, M. A., Siekkinen, M., Khan, K. N., Xiao, Y., and Tarkoma, S. (2015). Modeling, profiling, and debugging the energy consumption of mobile devices. *ACM Comput. Surv.*, 48(3):39:1–39:40.

Iyer, A. and Marculescu, D. (2001). Power aware microarchitecture resource scaling. In *Proceedings of the conference on Design, automation and test in Europe*, pages 190–196. IEEE Press.

Jabbarvand, R., Sadeghi, A., Bagheri, H., and Malek, S. (2016). Energy-aware test-suite minimization for android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 425–436.

Jabbarvand, R., Sadeghi, A., Garcia, J., Malek, S., and Ammann, P. (2015). Ecodroid: An approach for energy-based ranking of android apps. In *Proceedings of 4th International Workshop on Green and Sustainable Software*, GREENS '15, pages 8–14. IEEE Press.

Kambadur, M. and Kim, M. A. (2014). An experimental survey of energy management across the stack. In *ACM SIGPLAN Notices*, volume 49, pages 329–344. ACM.

Kerstens, A. and DuChene, S. A. (2008). Applying green computing to clusters and the data center. In *Linux Symposium*, volume 2008, page 2.

Ko, A. J., Latoza, T. D., and Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141.

Koomey, J. (2011). Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times.*

Kravets, R. and Krishnan, P. (1998). Power management techniques for mobile communication. In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 157–168. ACM.

Lago, P. (2015). Challenges and opportunities for sustainable software. In *Proceedings of the Fifth International Workshop on Product LinE Approaches in Software Engineering*, PLEASE '15, pages 1–2. IEEE Press.

Li, D. and Halfond, W. G. J. (2014). 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. ACM.

Li, D., Hao, S., Halfond, W. G., and Govindan, R. (2013). Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89. ACM.

Li, D., Jin, Y., Sahin, C., Clause, J., and Halfond, W. G. (2014). Integrated energy-directed test suite optimization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 339–350. ACM.

Li, S. and Mishra, S. (2016). Optimizing power consumption in multicore smartphones. *Journal of Parallel and Distributed Computing*, 95:124–137.

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

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. (2014). Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM.

Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. V., and Eder, K. (2013). Energy consumption analysis of programs based on xmos isa-level models. In *Logic-Based Program Synthesis and Transformation*, pages 72–90. Springer.

Liu, K., Pinto, G., and Liu, Y. D. (2015). Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering*, pages 316–331. Springer.

Liu, Y., Xu, C., Cheung, S.-C., and Lü, J. (2014). Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, 40(9):911–940.

Lorenzo, O. G., Pena, T. F., Cabaleiro, J. C., Pichel, J. C., Rivera, F. F., and Nikolopoulos, D. S. (2015). Power and energy implications of the number of threads used on the intel xeon phi. *Annals of Multicore and GPU Programming*, 3(1):55–65.

Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L. K., and Voelker, G. M. (2013). edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *NSDI*, volume 13, pages 57–70.

Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., and Clause, J. (2016). An empirical study of practitioners' perspectives on green software engineering. In *International Conference on Software Engineering (ICSE), 2016 IEEE/ACM 38th*, pages 237–248. IEEE.

Manotas, I., Pollock, L., and Clause, J. (2014). Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, pages 503–514. ACM.

McIntosh, A., Hassan, S., and Hindle, A. (2018). What can android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering*, pages 1–40.

Media, S. (2018). SourceForge. `https://sourceforge.net`.

Melfe, G., Fonseca, A., and Fernandes, J. P. (2018). Helping developers write energy efficient haskell through a data-structure evaluation. In *2018 IEEE/ACM 6th International Workshop on Green And Sustainable Software (GREENS)*, pages 9–15. IEEE.

Monsoon (2018). Monsoon solutions, inc. `http://www.msoon.com/LabEquipment/PowerMonitor/`.

Mouftah, H. T. and Kantarci, B. (2013). Chapter 11 - Energy-Efficient Cloud Computing: A Green Migration of Traditional {IT}. In Obaidat, M. S., Anpalagan, A., and Woungang, I., editors, *Handbook of Green Information and Communication Systems*, pages 295–330. Academic Press.

Nanz, S. and Furia, C. A. (2015). A comparative study of programming languages in rosetta code. In *IEEE International Conference on Software Engineering (ICSE), 2015 IEEE/ACM 37th*, volume 1, pages 778–788. IEEE.

Nobre, R., Reis, L., and Cardoso, J. M. P. (2018). Compiler phase ordering as an orthogonal approach for reducing energy consumption. volume abs/1807.00638.

Noureddine, A., Rouvoy, R., and Seinturier, L. (2015). Monitoring energy hotspots in software. *Automated Software Engineering*, pages 1–42.

Oliner, A. J., Iyer, A. P., Stoica, I., Lagerspetz, E., and Tarkoma, S. (2013). Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on*

*Embedded Networked Sensor Systems, SenSys '13, Roma, Italy, November 11-15, 2013*, pages 10:1–10:14. ACM.

Oliveira, W., Oliveira, R., and Castor, F. (2017). A study on the energy consumption of android app development approaches. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 42–52. IEEE Press.

Pang, C., Hindle, A., Adams, B., and Hassan, A. E. (2016). What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89.

Pankratius, V., Schmidt, F., and Garretón, G. (2012). Combining functional and imperative programming for multicore software: an empirical study evaluating scala and java. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 123–133. IEEE Press.

Park, J. J., Hong, J.-E., and Lee, S.-H. (2014). Investigation for software power consumption of code refactoring techniques. In *SEKE*, pages 717–722.

Passos, L. S., Abreu, R., and Rossetti, R. J. (2015). Spectrum-based fault localisation for multi-agent systems. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 1134–1140.

Pathak, A., Hu, Y. C., and Zhang, M. (2012). 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*, pages 29–42. ACM.

Pereira, R. (2017). Locating energy hotspots in source code. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 88–90. IEEE Press.

Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. a. P., and Saraiva, J. a. (2017a). Helping programmers improve the energy efficiency of source code. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 238–240, Piscataway, NJ, USA. IEEE Press.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. a. P., and Saraiva, J. a. (2017b). 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. ACM.

Pereira, R., Couto, M., Saraiva, J. a., Cunha, J., and Fernandes, J. a. P. (2016). 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. ACM.

Pereira, R., Simão, P., Cunha, J., and Saraiva, J. a. (2018). 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. ACM.

Perez, A. (2018). *Spectrum-based Diagnosis: Measurements, Improvements and Applications*. PhD thesis, Faculdade de Engenharia, Universidade do Porto.

Pering, T., Burd, T., and Brodersen, R. (2000). Voltage scheduling in the iparm microprocessor system. In *Proceedings of the 2000 International Symposium on Low power electronics and design*, pages 96–101. ACM.

Pettis, N., Ridenour, J., and Lu, Y.-H. (2006). Automatic run-time selection of power policies for operating systems. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 508–513. European Design and Automation Association.

Pinto, G. and Castor, F. (2017). Energy efficiency: a new concern for application software developers. *Communications of the ACM*, 60(12):68–75.

Pinto, G., Castor, F., and Liu, Y. D. (2014a). Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31. ACM.

Pinto, G., Castor, F., and Liu, Y. D. (2014b). Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 345–360. ACM.

Pinto, G., Liu, K., Castor, F., and Liu, Y. D. (2016). A comprehensive study on the energy efficiency of java's thread-safe collections. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 20–31.

Rasmussen, K., Wilson, A., and Hindle, A. (2014). Green mining: energy consumption of advertisement blocking methods. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 38–45. ACM.

Real, R. and Vargas, J. M. (1996). The probabilistic basis of jaccard's index of similarity. *Systematic biology*, pages 380–385.

Ribic, H. and Liu, Y. D. (2014). 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. ACM.

Ricciardi, S., Palmieri, F., Torres-Viñals, J., Martino, B. D., Santos-Boada, G., and Solé-Pareta, J. (2013). Chapter 10 - Green Data center Infrastructures in the Cloud Computing Era. In Obaidat, M. S., Anpalagan, A., and Woungang, I., editors, *Handbook of Green Information and Communication Systems*, pages 267–293. Academic Press.

Rosenthal, R. (1991). *Meta-analytic procedures for social research*, volume 6. Sage.

Rosenthal, R., Cooper, H., and Hedges, L. (1994). Parametric measures of effect size. *The handbook of research synthesis*, pages 231–244.

Rosenthal, R. and Rosnow, R. (1984). *Essentials of Behavioral Research: Methods and Data Analysis*. McGraw-Hill series in psychology. McGraw-Hill.

Rotem, E., Naveh, A., Ananthakrishnan, A., Weissmann, E., and Rajwan, D. (2012). Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27.

Rousseau, R. (1998). Jaccard similarity leads to the marczewski-steinhaus topology for information retrieval. *Information processing & management*, 34(1):87–94.

Saborido, R., Morales, R., Khomh, F., Guéhéneuc, Y.-G., and Antoniol, G. (2018). Getting the most from map data structures in android. *Empirical Software Engineering*, pages 1–36.

Sahin, C., Cayci, F., Gutierrez, I. L. M., Clause, J., Kiamilev, F., Pollock, L., and Winbladh, K. (2012). Initial explorations on design pattern energy usage. In *1st International Workshop on Green and Sustainable Software (GREENS), 2012*, pages 55–61. IEEE.

Sahin, C., Pollock, L., and Clause, J. (2014). 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. ACM.

Sahin, C., Wan, M., Tornquist, P., Mckenna, R., Pearson, Z., Halfond, W. G. J., and Clause, J. (2016). How does code obfuscation impact energy usage? *Journal of Software: Evolution and Process*, 28(7):565–588.

Shapiro, S. and Wilk, M. (1965). An analysis of variance test for normality. *Biometrika*, 52(3):591–611.

Shukla, Y. V. (2012). Cloud computing: Make it green computing. *Journal of Advances in Computational Research: An International Journal*, 1(1-2).

St-Amour, V., Tobin-Hochstadt, S., and Felleisen, M. (2012). Optimization coaching: optimizers learn to communicate with programmers. In *ACM SIGPLAN Notices*, volume 47, pages 163–178. ACM.

Standard, R. (2017). GHG Protocol Product Life Cycle Accounting and Reporting Standard ICT Sector Guidance. In *Greenhouse Gas Protocol*, number January, chapter 6 - Guide for assessing GHG emissions related to Software.

Stulova, N., Morales, J. F., and Hermenegildo, M. V. (2016). Reducing the overhead of assertion run-time checks via static analysis. In *PPDP*, pages 90–103.

Su, C.-L., Tsui, C.-Y., and Despain, A. M. (1994). Low power architecture design and compilation techniques for high-performance processors. In *Compcon Spring'94, Digest of Papers.*, pages 489–498. IEEE.

Symantec (2008). Corporate responsibility report. `http://www.symantec.com/content/en/us/about/media/SYM_CR_Report.pdf`.

Thompson, J. (2008). Environmental progress and next steps. *Email to Everyone Symantec (Employees)*.

Tiwari, V., Malik, S., and Wolfe, A. (1994). Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445.

Trefethen, A. E. and Thiyagalingam, J. (2013a). Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444 – 449.

Trefethen, A. E. and Thiyagalingam, J. (2013b). Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, 4(6):444–449.

Tukey, J. W. (1977). *Exploratory data analysis*. Addison-Wesley Pub. Co.

Verdecchia, R., Guldner, A., Becker, Y., and Kern, E. (2018). Code-level energy hotspot localization via naive spectrum based testing. EnviroInfo 2018.

Williams, K., McCandless, J., and Gregg, D. (2010). Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 278–287. ACM.

Wirth, N. (1995). A plea for lean software. *Computer*, 28(2):64–68.

Woodruff, M. and Herman, J. (2013). pareto.py: a $\epsilon - nondomination$ sorting routine. https://github.com/matthewjwoodruff/pareto.py.

Yuan, W. and Nahrstedt, K. (2003). Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 149–163. ACM.

Yuki, T. and Rajopadhye, S. (2014). Folklore confirmed: Compiling for speed= compiling for energy. In *Languages and Compilers for Parallel Computing*, pages 169–184. Springer.

Zwillinger, D. and Kokoska, S. (1999). *CRC standard probability and statistics tables and formulae*. Crc Press.